Foundations of Historic and Modern Object-Orientation

Simon Hamilton
simpham@email.ecok.edu
East Central University
Mathematics and Computer Science
Ada, OK, USA

I. INTRODUCTION

Object-Oriented Programming (OOP) has, over the past forty years, become a prominent paradigm for creating software systems. By modeling software using intractable "objects," OOP brings simplified structures into code, fostering developer speed and adaptability.

OOP is a collection of core mechanisms—inheritance, data abstraction, encapsulation, polymorphism, class design, and composition—that are designed for smooth and simple programming. Together, they define how objects are specified, related, and evolved. Each mechanism has been the subject of extensive academic study and scrutiny, yielding a rich body of results: formal type-theoretic treatments of sub-typing, empirical evaluations of interface versus implementation inheritance, language-level support for information hiding, patterns for behavioral polymorphism, and best practices for class cohesion and coupling.

In this paper, a survey of several academic research papers will be analyzed. I first traced the historical development of each principle, then compared alternative formalisms, language features, and finally identify open challenges and trade-offs that still constrain large-scale OOP designs today. By synthesizing insights across decades of research, my goal is to illuminate both the enduring strengths of OOP and the areas where new abstractions or tooling might push the paradigm forward.

II. FUNDAMENTAL PRINCIPALS OF OBJECT-ORIENTATION

A. Inheritance

Inheritance is the mechanism by which a class (the *subclass* or *derived class*) takes on the properties and behavior of another class (the *base class*). Defining a subclass inherits all public and private members of its parent automatically, encouraging code reuse and natural taxonomies (e.g. `Animal` → `Dog`). Single and multiple inheritance are allowed: in single inheritance, each class has exactly one immediate parent, but in multiple inheritance a class may have multiple parents (with resolution of name conflicts and the "diamond problem"). Proper use of inheritance eliminates duplication but must be balanced against tight coupling between base and derived classes.

B. Encapsulation

Encapsulation is bundling an object's internal state (its data fields) with the operations (methods) that change that state, and restricting direct external access. By making attributes `private` or `protected`, and only providing a well-defined *public* interface, a class hides implementation details and ensures its invariants. This information hiding prevents clients from putting objects into illegal states, makes maintenance easier (you can change internals without breaking callers), and makes the intended use of each member explicit. Common techniques include getters/setters, immutability, and carefully chosen visibility modifiers.

C. Data Abstraction

Data abstraction separates the *interface* (accessible operations) of an abstract data type from its *representation* (implementation of something). An *interface* or an *abstract class* offers a contract of methods (say, `enqueue`, `dequeue` for a queue ADT) without committing to linked-list, array, or other internal implementations. Clients are designed to the interface, rather than the implementation, so various concrete variants (e.g. bounded vs. unbounded queues) can be interchanged with no client changes. This layering supports modular reasoning, formal verification of behavior contracts, and performance tuning in front of a stable API.

D. Polymorphism

Polymorphism lets one interface stand for multiple underlying forms. There are three main styles:

- *Ad hoc polymorphism* (overloading): same function name but different signatures (e.g. `abs(int)` vs. `abs(double)`).
- *Parametric polymorphism* (generics/templates): code written without specific types, instantiated later (e.g. `List<T>`).
- *Subtype polymorphism* (dynamic dispatch): a base-class pointer/reference can invoke overridden methods on any subclass instance, with the actual method chosen at runtime.

Subtype polymorphism enables designs like `Drawable` collections of `Circle`, `Square`, etc., all sharing a common `draw()` method.

III. ORIGINS OF OBJECTIVE PROGRAMMING

A. First Instances of OOP

In 1962, computer scientists Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo, developed Simula 1, the first OOP language. Simula 1 introduced principles of objects, inheritance, classes and subclasses, and other forms of sub-typing. As the name implies, Simula was created for the creation of computer simulations. In 1967, an updated version with more expansive integrated functions was released under the name, Simula 67.

B. Governmental Usage

During the late 1970s the U.S. Department of Defense was in need of a mission-critical programming language. In response, the DoD commissioned a new language, standardized in 1983 as Ada 83, which unified syntax and semantics across all projects. Ada's *package* mechanism cleanly separates interface (specification) from implementation (body), enforcing data abstraction and encapsulation at the module level.

With the introduction of Ada 95, the language gained full object-orientation:

- **Tagged types** and **type extension** provide single-inheritance hierarchies and allow derived types to inherit fields and methods from a parent ("base") type.
- **Class-wide types** (e.g. 'T'Class) enable subtype polymorphism via dynamic dispatch, so that calls to an overridden primitive procedure resolve at runtime to the correct implementation.
- **Generic units** deliver parametric polymorphism, permitting algorithms and data structures to be defined abstractly over any type that meets specific interface requirements.

Composition is natively supported through record types containing objects, favoring "has-a" relationships over deep hierarchies when appropriate. Subsequent revisions (Ada 2005 and Ada 2012) further extended OOP with interface types, multiple inheritance of interfaces, and contract-based programming ("pre"/"post" conditions), bringing formal design-by-contract into the language's core.

Because of its strong typing, midday run-time checks, exception handling, built-in tasking (for concurrency), and potential for integrated garbage collection, Ada became—and remains—a benchmark for reliability and maintainability in avionics, defense, and other high-integrity fields where the full suite of inheritance, encapsulation, data abstraction, polymorphism, and composition is essential.

IV. MODERN PROGRAMMING LANGUAGES

A. Python

Today, Python is statistically the most popular programming language in the world. It is an interpreted language meaning the written code does not compile down into a binary, but instead runs directly line-by-line from an interpreter. Although convenient, Python has an exponentially large execution time. This makes it ideal for quick data science operations, scripts, automation, etc. but not ideal for memory intense loads, or large scale production software.

B. C++

C++ has been around far longer than the vast majority of other languages. Yet, it has endured and remains the fifth most popular programming language in the world for 2025. C++ is primarily used in memory intensive software such as game development, academia, research, embedded systems, and enterprise SaaS deployments. The reason it has stayed so relevant is partially due to its native backwards compatibility with native C code. This allows C programs to be quickly migrated over to more modern systems across the world.

C. Java

The Java programming language remains the second most popular language for 2025. Java (not related to JavaScript) is has become ubiquitous in large-scale enterprise backends, Android mobile development, server-side web services, and game engines. Its "write once, run anywhere" philosophy—backed by the Java Virtual Machine—combined with strong static typing, automatic garbage collection, and a pure object-oriented model (classes, interfaces, inheritance and polymorphism) streamlines development, promotes code reuse, and helps teams build robust, systems with minimal ceremony.

V. FUTURE USE CASES & ARTIFICIAL INTELLIGENCE

Artificial Intelligence (AI) has become a staple in Computer and Data Science, and modern OOP languages are evolving to leverage AI's capabilities at every layer of the stack. Below are several emerging trends and future use cases that illustrate how AI and OOP will continue to intertwine:

1. AI-Augmented Development Environments

- **Intelligent Code Completion and Refactoring:** IDEs powered by large language models can suggest entire class hierarchies, method implementations, and design patterns in real time—reducing boilerplate and accelerating the prototyping cycle. CoPilot within Visual Studio Code is an example of this.

2. Autonomous Microservices and Distributed Systems

- **Predictive Scaling and Cost Optimization:** Machine learning models embedded in each microservice class predict traffic spikes and proactively adjust resources, ensuring both high availability and cost efficiency in cloud deployments. This can be compared to automated rapid elasticity.

3. Embedded and Edge Computing

- **On-Device Inference:** Object-oriented languages like C++ and Rust are gaining built-in libraries for quantized neural networks. Classes encapsulate sensor data pipelines, allowing constrained devices (e.g. drones,

IoT sensors) to perform real-time object detection or anomaly detection without cloud latency.

- **Adaptive Control Systems:** AI-driven PID controllers and reinforcement-learning agents can be implemented as polymorphic classes that learn optimal control policies on the fly, enabling robotics and industrial automation.

4. Intelligent Operating System Services

- **Predictive Task Scheduling:** Future OS kernels written in object-oriented or object-capable languages may include scheduler classes that leverage AI to predict process runtimes and I/O patterns, optimizing throughput and power consumption.
- **Context-Aware Security:** Security-manager classes will incorporate anomaly detection models that continuously learn "normal" system behavior, rapidly quarantining or rolling back suspicious processes before they can cause harm.

5. AI-Driven Domain-Specific Frameworks

- **Neural DSLs in OOP:** New domain-specific languages embedded in Java, Python, or C# allow developers to define neural architectures as class hierarchies. These DSL classes generate optimized computation graphs and handle distributed training under the hood. Neural Networks, in this context, can be trained for nearly infinite circumstances.

By melding traditional OOP principles—encapsulation, inheritance, polymorphism—with the adaptivity and predictive power of AI, software systems will become more autonomous. As AI libraries mature, developers will routinely compose neural-network components alongside classic business logic classes, creating hybrid applications that learn, adapt, and optimize themselves within production.

VI. CONCLUSION

Over the past six decades, object-oriented programming has matured from Simula—where the very notions of classes and objects were first forged—to today's AI-driven ecosystems in Python. This survey has dissected the four pillars of OOP—inheritance, encapsulation, data abstraction, and polymorphism—showing how each principle first emerged, how they were formalized in successive languages (including Ada, C++, Java, and C#), and how modern languages continue to refine them with generics, interfaces, and formal contracts.

We have also traced the evolution of best practices in class design and composition, illustrating why composition often supersedes deep inheritance hierarchies, and how design patterns and SOLID principles guide developers toward high cohesion and low coupling. With the department-wide adoption of Ada in the 1980s, the "write once, verify everywhere" approach set new benchmarks for reliability; Java's ubiquitous JVM and pure OOP model cemented portability and large-scale maintainability; and today's hybrid paradigms in Python seamlessly blend procedural, functional, and object-oriented styles into production.

Looking forward, the integration of machine/deep learning and AI into OOP frameworks promises to reshape how we build, test, and deploy software: from intelligent refactoring assistants in our IDEs (CoPilot, etc.) to self-optimizing microservices and AI-infused domain-specific languages. As class hierarchies begin to encapsulate neural components alongside classical business logic, tomorrow's systems will model our problems—learn, adapt, and refactor.

Ultimately, while languages and libraries will continue to evolve, the core tenets of object orientation—modularity, reuse, and clear abstractions—remain constant today as they were in the earliest Simula 1 simulation software systems. By understanding both the historical roots and the modern ideology of OOP, developers and researchers alike can better harness these principles to construct the resilient, extensible, and intelligent software systems of the future.

REFERENCES

[1] Kristen Nygaard and Ole-Johan Dahl. 1978. The development of the SIMULA languages. History of programming languages. Association for Computing Machinery, New York, NY, USA, 439–480. https://doi.org/10.1145/800025.1198392

[2] William E. Carlson, Larry E. Druffel, David A. Fisher, and William A. Whitaker. 1980. Introducing Ada. In Proceedings of the ACM 1980 annual conference (ACM '80). Association for Computing Machinery, New York, NY, USA, 263–271. https://doi.org/10.1145/800176.809976

[3] Peter Wegner. 1990. Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess. 1, 1 (Aug. 1990), 7–87. https://doi.org/10.1145/382192.383004

[4] A. Toni Cohen. 1984. Data abstraction, data encapsulation and object-oriented programming. SIGPLAN Not. 19, 1 (January 1984), 31–35. https://doi.org/10.1145/948415.948418

[5] Alan Snyder. 1986. Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not. 21, 11 (Nov. 1986), 38–45. https://doi.org/10.1145/960112.28702

[6] Tim Rentsch. 1982. Object oriented programming. SIGPLAN Not. 17, 9 (September 1982), 51–57. https://doi.org/10.1145/947955.947961

[7] Bertrand Meyer, Alisa Arkadova, and Alexander Kogtenkov. 2024. The Concept of Class Invariant in Object-oriented Programming. Form. Asp. Comput. 36, 1, Article 5 (March 2024), 38 pages. https://doi.org/10.1145/3626201

[8] Carlos E. C. Dantas and Marcelo de A. Maia. 2017. On the actual use of inheritance and interface in Java projects: evolution and implications. In Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (CASCON '17). IBM Corp., USA, 151–160.

[9] Peter Wegner. 2003. Object-oriented programming (OOP). Encyclopedia of Computer Science. John Wiley and Sons Ltd., GBR, 1279–1284.

[10] Adam Ipsen, "Top 10 programming languages for 2025, PluralSight, https://www.pluralsight.com/resources/blog/upskilling/top-programming-languages-2025.

[11] H. Wechsler and D. Rine, "Object oriented programming (OOP) and its relevance to designing intelligent software systems," in Proceedings 1988 International Conference on Computer Languages, Miami Beach, FL, USA, 1988, pp. 242,243,244,245,246,247,248, doi: 10.1109/ICCL.1988.13070.