

Why OOP Matters More, Not Less, in the AI Era

*The less code developers write by hand,
the more their OOP judgment shapes what AI produces.*

Jennifer Jin, Soo Dong Kim • October 2026 • Version 1.0

1. OOP Is Everywhere: The De Facto Standard of Programming

Object-oriented programming is the dominant paradigm of modern software development: most widely used languages, libraries, frameworks, and platforms are organized around it. Its emphasis on modularity, abstraction, encapsulation, reusability, and extensibility gives developers a structured way to organize complex systems around objects, classes, interfaces, components, and well-defined interactions. As software systems grow in size, diversity, and interdependency, this structure has become the common foundation of the programming world. The breadth of this foundation is illustrated in Figure 1.



Figure 1. Ubiquity of the Object-Oriented Paradigm

As shown in the figure, the object-oriented paradigm pervades both software development practices and reusable software assets.

Development practices. Many of the most widely used programming languages, including Java, C++, Python, C#, and Swift, are either primarily object-oriented or support object-oriented programming alongside other paradigms. Programming libraries organize reusable functionality as classes and interfaces, as in the Java Collections Framework with its *List*, *Set*, and *Map* interfaces. Frameworks such as Spring, Django, and Angular define extension points that developers specialize by extending framework classes and implementing framework interfaces, and platforms such as the Java Virtual Machine, .NET, Android, and iOS expose their capabilities through object-oriented APIs and class libraries. Software analysis and design follow the same model: object-oriented analysis and design, UML, and domain-driven design describe systems in terms of objects, responsibilities, and collaborations, from requirements through implementation.

Reusable software assets. Object-oriented principles also shape the design knowledge that is reused across systems. Design patterns such as *Factory Method*, *Composite*, and *Strategy* capture recurring solutions as arrangements of classes and objects. Architecture styles such as Model-View-Controller and Publish-Subscribe rely on modular boundaries, explicit interfaces, and clear responsibility assignment. Architecture reference models such as AUTOSAR for automotive software and openEHR for electronic health records define reusable structures built on object-oriented modeling. Cloud and microservice systems apply the same

principles at the scale of services: each service encapsulates a business capability behind a well-defined API, and cloud SDKs expose storage, computation, and machine learning services through client, request, and response objects.

OOP is therefore the de facto standard of the programming world. Its concepts form the shared vocabulary of developers, the structure of the code they maintain, and the contracts they must honor whenever they use a library, framework, platform, or API. This ubiquity extends to AI copilots: code produced by a copilot must fit into object-oriented languages, libraries, and frameworks. Whether code is written by hand or generated by AI, the result is, in most cases, object-oriented software. OOP is not a legacy topic that AI will make obsolete; it is the medium in which AI-generated code lives.

2. Does OOP Knowledge Still Matter?

AI copilots can now generate substantial amounts of program code directly from natural-language prompts. A developer can describe the desired classes, operations, relationships, and constraints, and receive candidate code within seconds. In one controlled experiment, developers using GitHub Copilot completed a programming task 55.8% faster than those working without it [2]. This capability naturally raises a question: **If AI can generate object-oriented code, does OOP knowledge still matter?** This article argues that it matters more, not less.

OOP knowledge still matters in AI-assisted programming for the following seven reasons. The first two concern what is asked of the copilot, the next three concern how its output is judged, and the last two concern how the accepted code is integrated and sustained.

Reason 1. Decomposing the problem into objects is still the developer's job.

A copilot works best on a well-scoped request, not on an entire system. Before anything is generated, the developer must identify the domain objects, divide the system into classes and components, and decide which responsibilities belong where. This object-oriented modeling determines what each prompt asks for, and a copilot cannot do it reliably without the developer's understanding of the application domain.

Reason 2. Design intent must be stated in OOP terms, or the copilot decides.

Every prompt leaves some structural choices open: whether to inherit or compose, which class owns a responsibility, whether to depend on an interface or a concrete class. The copilot fills these gaps by default. Terms such as *interface*, *composition*, *responsibility*, and *cohesion* let a developer close them in a few words. The same holds for review and refactoring requests, where OOP terms specify which dimensions of design quality to examine and which structural weaknesses to correct.

Reason 3. Judging generated code requires object-oriented quality criteria.

Compilers and tests check whether code behaves correctly, not whether its object structure is sound. Whether state is properly encapsulated, whether a subclass is truly substitutable for its superclass, whether a class is cohesive, and whether classes are loosely coupled are object-oriented criteria. Only a developer who has mastered them can apply them to code that already runs.

Reason 4. Validating AI review findings requires object-oriented design knowledge.

An AI review report can be wrong in two directions. It may flag a sound design as a flaw, such as recommending composition in place of a genuinely justified inheritance relationship. It may also miss a real flaw, such as a *Square* class that inherits from *Rectangle* but breaks its expected behavior. Separating

true findings from false positives, and noticing what the copilot overlooked, requires the design knowledge the review was meant to supplement.

Reason 5. Refactoring with AI requires object-oriented design judgment.

Effective refactoring begins with recognizing object-oriented code smells, such as a large class with scattered responsibilities, feature envy, or repeated type-based conditional logic. It continues with choosing appropriate transformations, such as *Extract Class*, *Move Method*, or *Replace Conditional with Polymorphism*. A copilot can carry out these transformations, but whether the object structure has actually improved is an object-oriented judgment.

Reason 6. Generated code must fit into object-oriented systems.

As the previous section showed, new code must extend framework classes, implement platform interfaces, and conform to the object structure of an existing codebase. A copilot that sees only part of the system may produce code that works in isolation but weakens the design it joins.

Reason 7. Evolving AI-generated code requires understanding its object structure.

A copilot produces the first version, but the developer must extend, debug, and maintain the code over its lifetime. Adding a new variant means knowing which class hierarchy or interface it belongs to and how polymorphic dispatch will reach it; diagnosing a defect means tracing dynamic binding and message flows among collaborating objects. Without object-oriented expertise, a developer can run AI-generated code but cannot safely change it.

Taken together, these reasons show that OOP knowledge serves the developer at every stage of AI-assisted work: it shapes what is asked of the copilot, provides the criteria for judging what the copilot returns, and sustains the code after it is accepted. The rest of this article develops this argument, beginning with how AI shifts the developer's role.

3. AI Shifts Developers from Writing to Judging Code

Conventional programming places most implementation effort on the developer. The developer interprets requirements, designs a solution, writes source code, tests it, corrects defects, and progressively improves the implementation.

AI copilots change this workflow because part of the design and coding effort can now be delegated. Boilerplate code, classes, constructors, methods, inheritance structures, interfaces, tests, and alternative implementations can often be generated much faster than through manual coding.

What remains, and grows, is the need to determine whether the resulting implementation is appropriate. A generated program can compile, run, and appear to work correctly, yet still be poorly structured: internal state may be exposed, inheritance misused, or classes tightly coupled. It may also rest on assumptions that conflict with the application domain.

The change becomes clear when the two workflows are compared, as shown in Figure 2.

Conventional Programming*AI-Assisted Programming***Figure 2. Conventional and AI-Assisted Programming Workflows**

As shown in the figure, both workflows consist of three steps, and the developer performs the first and the last in each. The difference lies in the first two steps. In conventional programming, the developer designs the solution in detail, deciding the classes, their attributes and methods, and their relationships, and then writes the code. In AI-assisted programming, the developer specifies the design intent in a prompt, namely the objects, their responsibilities, and their key relationships, and the copilot works out the remaining details, such as attributes, method signatures, and internal structure, and generates the code. Two things therefore change hands: the detailed design and the writing of code.

The third step is the same in both workflows, but what is evaluated differs. In conventional programming, the developer evaluates code built on the developer's own design decisions. In AI-assisted programming, the developer evaluates code whose detailed design was decided by the copilot.

In practice, the AI-assisted workflow proceeds as a human-in-the-loop cycle of four activities. The first step in the figure divides into two activities, a decision and a prompt, and each of the other two steps corresponds to one activity.

Human decision. The developer establishes the programming objective and decides how the copilot will be used. Routine tasks, such as generating boilerplate code or drafting a simple class, may be delegated. More critical tasks, such as defining core domain rules or deciding the class structure of the domain model, require stronger human control.

Prompt formulation. The developer specifies the task for the copilot: the desired output, target language, relevant context, constraints, and expected format. In object-oriented programming, a prompt may specify class names, attributes, methods, visibility rules, inheritance relationships, interface requirements, or a design pattern to apply.

Copilot output. The copilot produces candidate artifacts, such as code, alternative designs, review comments, refactoring suggestions, or test cases. These artifacts can accelerate development, but they are drafts that may need correction, restructuring, or further validation.

Human evaluation. The developer examines the output against the requirements, object-oriented principles, and coding standards, and checks that it is readable, maintainable, and secure. The developer then accepts, revises, or rejects it, or formulates a new prompt.

The cycle is iterative. A developer may add missing context, request alternative implementations, or ask the copilot to improve its output, refining an initial draft into a reliable software artifact over several rounds. Notably, three of the four activities belong to the developer, and the two that open and close the cycle, deciding what should be built and judging what was produced, are precisely where object-oriented expertise is exercised. A later article in this series (ART-AIOOP-03) examines this cycle in detail.

The developer is no longer necessarily the author of every line, or of every design detail, but remains the author of the **programming intent**, supplying domain knowledge, OOP foundations, and constraints, and the final judge who validates and integrates what the copilot returns.

This shift increases, rather than reduces, the value of OOP expertise. When a copilot generates a complete class hierarchy at once, dozens of structural decisions arrive already made, and recognizing which of them are sound requires the same object-oriented judgment the copilot was expected to replace. The same holds, with greater force, for coding agents that read an entire repository, run tests, and apply changes across many files: the more work is delegated, the more decisions arrive already made.

4. Where OOP Judgment Enters: Code Generation, Review, and Refactoring

AI copilots support three major object-oriented programming activities: **code generation**, **code review**, and **code refactoring**, as illustrated in Figure 3.

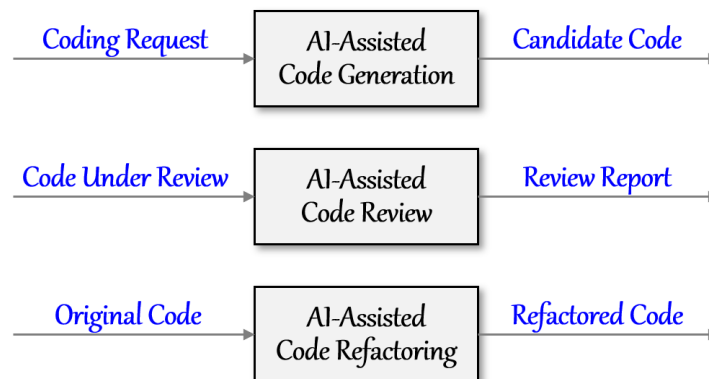


Figure 3. AI-Assisted Code Generation, Review, and Refactoring

As shown in the figure, the three activities share a common pattern. The developer supplies an input (a coding request, code under review, or original code), and the copilot returns a candidate artifact (candidate code, a review report, or refactored code) that the developer must evaluate before accepting it. Object-oriented foundations enter at both ends of this exchange, as summarized in Table 1. In the book, each activity is supported by reusable prompt templates, which the next section introduces.

Table 1. OOP Foundations in AI-Assisted Programming Activities

Activity	Developer Provides	Copilot Returns	OOP Judgment Required
Code generation	Prompt with requirements, design description, partial code, and target language	Candidate code: classes, interfaces, hierarchies, and client code	Whether responsibilities, relationships, and constructs match the intended design

Activity	Developer Provides	Copilot Returns	OOP Judgment Required
Code review	Code under review with review criteria and concerns	Review report: identified issues and suggested improvements	Whether findings are valid, and which corrections should be applied
Code refactoring	Existing code with improvement needs or refactoring guidance	Candidate refactored code	Whether behavior is preserved and structure has actually improved

Code generation. Copilots can identify candidate objects, define classes and responsibilities, generate attributes and methods, implement inheritance and polymorphism, and construct client code. Structural decisions left unstated in the prompt, such as inheritance versus composition, the assignment of responsibilities, and the use of interfaces, are made by the copilot by default. For example, a simple request to process payments with *CreditCardPayment* and *DigitalWalletPayment* objects may yield client code of the following kind, even though a common *PaymentMethod* interface already exists:

```
public class CheckoutService {
    public void payByCard(CreditCardPayment card, double amount) {
        card.pay(amount);
    }
    public void payByWallet(DigitalWalletPayment wallet, double amount) {
        wallet.pay(amount);
    }
}
```

The code works, but it depends directly on both concrete classes and must be modified whenever a payment type is added. A prompt based on the GEN-INT-04 template [1] brings the decision into the request:

Generate client code in Java that uses references of the interface type PaymentMethod to manipulate instances of the implementing classes CreditCardPayment and DigitalWalletPayment. Show the references in the following usage contexts: a method parameter. The client code must invoke only methods declared by the interface.

A representative response depends only on the interface:

```
public class CheckoutService {
    public void pay(PaymentMethod method, double amount) {
        method.pay(amount);
    }
}
```

A new payment type can now be added without modifying the client. Even so, what the copilot returns must still be judged as candidate code rather than a finished implementation. ART-AIOOP-05 examines AI-assisted code generation in detail.

Code review. Copilots can examine existing or generated code for defects and structural weaknesses, and recent research has investigated prompt engineering and model adaptation for automating code review with large language models [3]. A focused prompt, such as one based on the REV-FP-01 template [1], names the dimensions of object-oriented quality to examine: encapsulation, cohesion, coupling, inheritance, and so on. The resulting report, however, is itself AI-generated. A copilot reviewing an *Account* hierarchy may recommend replacing the *SavingsAccount* subclass with composition to “reduce coupling,” even though a

savings account is genuinely a kind of account and the inheritance is sound. Recognizing such a false positive is the developer's task, not the copilot's. ART-AIOOP-07 examines AI-assisted code review in detail.

Code refactoring. Copilots can improve the internal structure of existing code while preserving its externally observable behavior. The book organizes this activity into four steps, each supported by a dedicated family of prompt templates [1]: identify code smells, define a refactoring strategy, apply refactoring transformations, and validate the refactored code. Code that passes all tests is not necessarily successfully refactored; success requires two judgments: behavior is preserved, and structural quality has improved. ART-AIOOP-08 examines AI-assisted code refactoring in detail.

Across all three activities, the copilot extends what the developer can produce and inspect, but it does not replace the developer's object-oriented judgment. How effectively that judgment reaches the copilot depends largely on the prompt, the subject of the next section.

5. Using AI Effectively: Prompt Templates with OOP Built In

In AI-assisted programming, the **prompt** is the principal means by which developers convey their intent to a copilot, and the copilot's response is shaped by what the prompt contains, as illustrated in Figure 4.



Figure 4. Prompt and Response in AI Copilot Interaction

As shown in the figure, whatever the developer knows about the application domain, the intended design, and the applicable constraints reaches the copilot only if it is stated in the prompt; anything left unstated must be inferred, and the copilot's inference may not match the developer's intent. The quality of the resulting code, review report, or refactored code therefore depends directly on the quality of the prompt. Prompts composed ad hoc for each task, however, tend to vary in completeness and focus, and essential information is easily omitted.

One practical mechanism for addressing this problem is the **prompt template**. A **prompt template** is a reusable structure for formulating a recurring type of AI-assisted programming request. Rather than composing each request from scratch, a developer uses a template that systematically specifies important information such as the programming task, target OOP construct, relevant context, responsibilities, relationships, constraints, quality concerns, and expected output.

What makes prompt templates effective for object-oriented development is that they embed the object-oriented paradigm in their structure. Each template targets a specific construct or task, such as a class definition, an inheritance hierarchy, or client code that uses an interface, and its placeholders ask for the information that construct requires; across the catalog, these cover responsibilities, attributes and methods, relationships, interfaces, and constraints. Filling in a template therefore leads the developer to express intent in object-oriented terms, rather than describing functionality alone and leaving the structure to the copilot. The resulting prompts are more **systematic, complete, focused, and consistent** than prompts composed ad hoc.

For example, the book defines **GEN-CLS-02**, a template for generating a class definition from an object's name together with a description of its responsibilities [1]:

Prompt Template GEN-CLS-02: Class Definition from Object Description

Intent	To generate a class definition from an object name together with a description of its role and responsibilities.
Precondition	The name of the object, a description of its role and responsibilities, and the target programming language.
Prompt Template	Generate a class definition in <i><language></i> for the object named 'X', which is responsible for <i><description of responsibilities></i> . Include attributes, methods, and a constructor consistent with these responsibilities.
Expected Output	A class definition in the specified language whose attributes and methods reflect the described responsibilities.

Filled in for a *Reservation* object in a car rental system, the template becomes the following prompt [1]:

Generate a class definition in Java for the object named Reservation, which is responsible for recording a customer's request to rent a specific car type for a given period, tracking the reservation status, and supporting confirmation and cancellation. Include attributes, methods, and a constructor consistent with these responsibilities.

Because the template requires the developer to state the object's responsibilities, it builds a central object-oriented concept, responsibility assignment, directly into the request. As the book's usage note for this template observes, the clearer and more specific the description of responsibilities, the closer the generated class will be to the intended design; vague descriptions tend to produce generic classes.

Prompt templates do not, however, replace object-oriented judgment. A template tells the developer which information to supply, but not what that information should be. Describing an object's responsibilities well, choosing between inheritance and composition, and deciding which interface a client should depend on still require object-oriented foundations. The template provides the structure of the request; the developer's expertise provides its substance.

The book provides such templates for all three activities, including GEN-INT-04 for interface-based client code, REV-FP-01 for focused object-oriented code review, and the REF families for refactoring [1]. ART-AIOOP-06 presents the generation templates in detail, and ART-AIOOP-07 and ART-AIOOP-08 present those for review and refactoring, with complete examples.

Conclusion: Two Capabilities for the AI Era

AI copilots now produce a growing share of program code. The question for developers is therefore no longer whether to use AI, but what they must bring to the collaboration. The discussion in this article points to two complementary capabilities.

Solid object-oriented foundations. Developers need a working command of the principles of object-oriented programming (encapsulation and information hiding, abstraction, inheritance and polymorphism, and composition), the constructs that realize them in C++, Java, and Python, and the design guidelines that turn them into criteria for judging a design. These foundations allow a developer to state design intent precisely and to recognize whether what a copilot returns is sound. Without them, a developer can accept AI-generated code but cannot evaluate it.

Effective use of prompt templates for generation, review, and refactoring. Developers must also be able to bring these foundations to the copilot, and prompt templates are an effective means of doing so. Because templates embed the object-oriented paradigm, they turn each request for generation, review, or refactoring into a structured statement of design intent. Used consistently, they make interactions with the copilot systematic, complete, and repeatable, and they make its output easier to evaluate against the intent that was stated.

Neither capability is sufficient alone. Templates without foundations produce well-structured requests with weak content; foundations without templates leave design intent to be restated, and often partly omitted, in every ad hoc prompt. Together, the two capabilities define the developer’s role in the AI era: the author of programming intent and the final judge of what the copilot produces. Yet they are not equal partners. A template is only as good as the object-oriented judgment used to fill it, and that is why OOP matters more, not less, in the AI era.

About the Book

AI-Assisted Object-Oriented Programming: Generation, Review, and Refactoring in C++, Java, and Python (forthcoming) covers the core object-oriented constructs—objects, classes, inheritance, polymorphism, dynamic binding, abstract classes, interfaces, generic classes, and reflection—and applies them to AI-assisted code generation, review, and refactoring. It includes a catalog of reusable prompt templates, such as GEN-INT-04, REV-FP-01, and the REF template families, with examples in all three languages. Learn more at handsonse.org.

References

- [1] S. D. Kim, J. Jin, and M. Kim, *AI-Assisted Object-Oriented Programming: Generation, Review, and Refactoring in C++, Java, and Python*. Forthcoming, 2027.
- [2] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot,” arXiv preprint arXiv:2302.06590, 2023.
- [3] C. Pornprasit and C. Tantithamthavorn, “Fine-tuning and Prompt Engineering for Large Language Models-Based Code Review Automation,” *Information and Software Technology*, vol. 175, Art. no. 107523, 2024.

Article ID: ART-AIOOP-01 • Published: October 2026 • Version: 1.0 • handsonse.org