Object Model

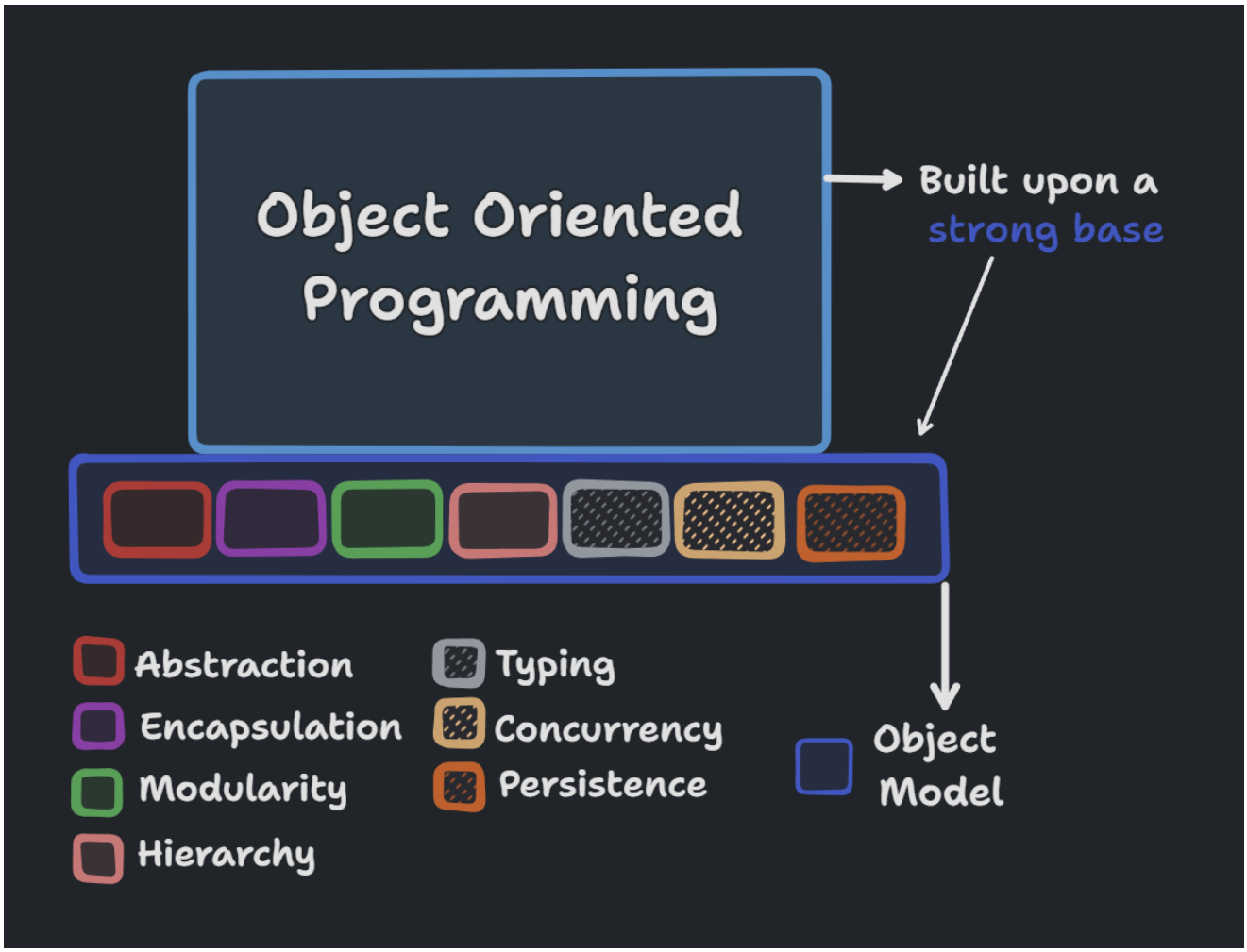
What is Object Model?

Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call the object model of development or simply the object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. By themselves, none of these principles are new. What is important about the object model is that these elements are brought together in a synergistic way.

More Simply,
Object-oriented technology is like a well-built house, with its solid foundation called the *object model*. This model is made up of different parts, such as abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. Each part is like a brick that isn't new on its own, but when they're put together, they create something special.

Think of abstraction as simplifying things, like how a remote control simplifies using a TV. Encapsulation is like putting things in boxes to keep them safe and organized. Modularity is breaking things into smaller, manageable pieces, like building blocks. Hierarchy is like a family tree, where things are organized from general to specific. Typing is like sorting things into categories, such as separating toys from books. Concurrency is doing multiple things at once, like cooking while also cleaning the kitchen. And persistence is like saving your game progress so you can pick up where you left off later.

What makes the object model important is how all these parts work together smoothly, like gears in a machine. It's this combination that makes object-oriented technology so powerful and useful for building complex software systems.



Evolution of Object Model:

Evolution of languages:

1. **First-generation languages**: These languages were primarily used for scientific and engineering applications and were heavily mathematical in nature. Languages like FORTRAN I allowed programmers to write mathematical formulas, abstracting away some intricacies of assembly or machine language. This generation brought programming closer to the problem space and further from the machine.
2. **Second-generation languages**: With the focus on algorithmic abstractions, these languages emerged as machines became more powerful and automation expanded, particularly in business applications. They instructed the machine on what to do in a step-by-step manner, moving programming closer to the problem space.
3. **Third-generation languages**: These languages, such as ALGOL 60 and Pascal, introduced support for data abstraction. Programmers could describe the meaning of related kinds of data (their type), letting the language enforce these design decisions. This further brought programming closer to the problem domain and away from the underlying machine.
4. **Late third-generation languages**: Modules became an important structuring mechanism to address programming-in-the-large. However, support for data

abstraction and strong typing was often lacking, leading to errors detected only during execution.

5. **Object-based and object-oriented programming languages**: These languages, including Smalltalk, Object Pascal, C++, and Java, emerged to support the object-oriented decomposition of software. They organized programs around classes and objects, providing a more natural way to model complex systems. The physical structure of applications in these languages appears as a graph, with classes and objects as fundamental building blocks. These languages scaled up to handle programming-in-the-colossal, with clusters of abstractions built in layers on top of one another, following the organization of complexity.

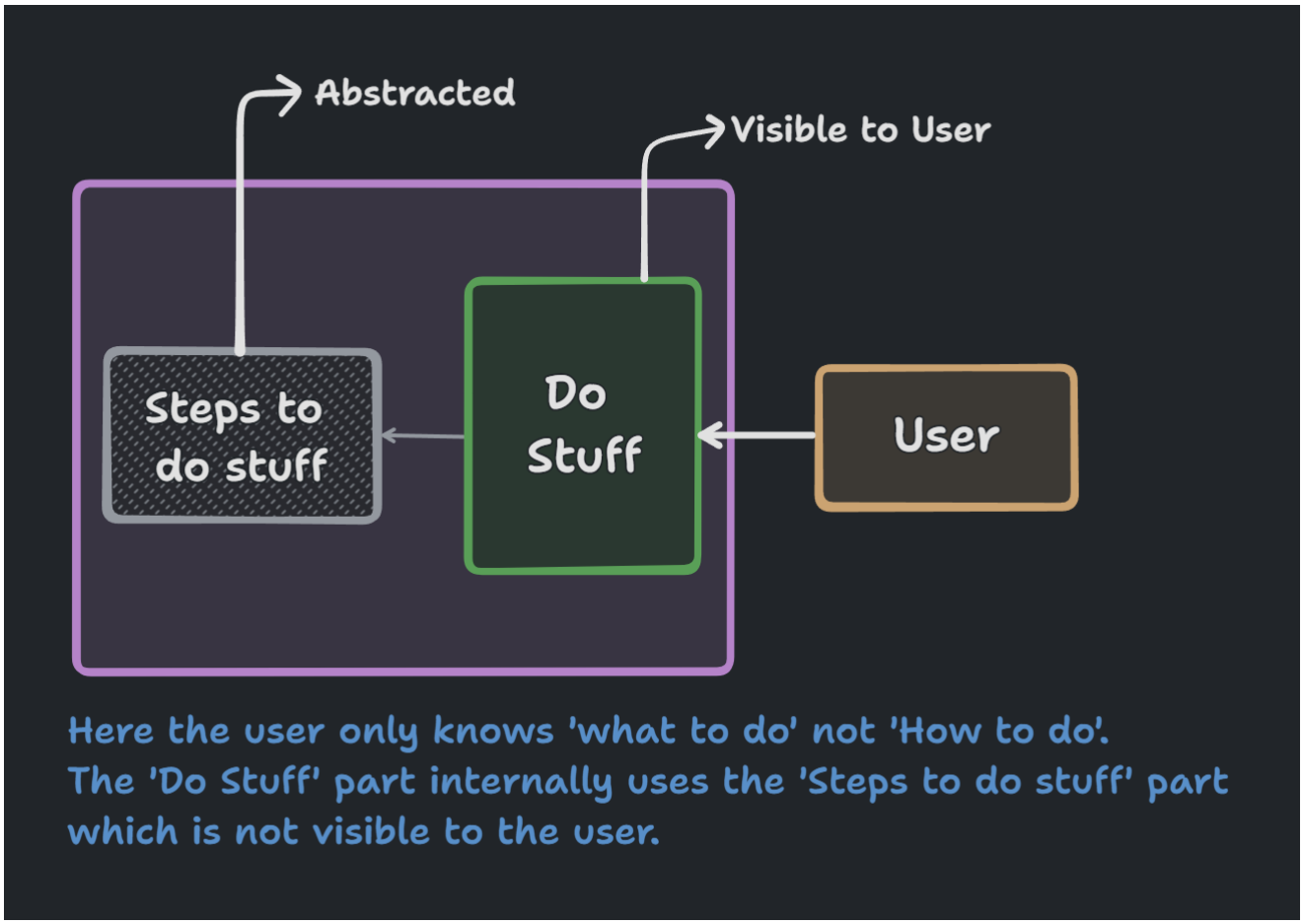
The Path to Object Oriented Language:

1. **First- and early second-generation languages**: These languages, like FORTRAN and COBOL, had a relatively flat physical structure. The basic building block was the subprogram (or paragraph in COBOL), and applications consisted mainly of global data and subprograms. Dependencies between subprograms and data were evident, but there was little enforcement of design decisions, leading to potential errors and maintenance challenges.
2. **Late second- and early third-generation languages**: Building upon earlier generations, these languages introduced structured programming concepts and supported greater control over algorithmic abstractions. They addressed some inadequacies of earlier languages but still struggled with programming-in-the-large and data design issues.
3. **Late third-generation languages**: These languages introduced separately compiled modules to address programming-in-the-large. However, modules were initially used primarily as containers for data and subprograms, lacking strong mechanisms for semantic consistency among module interfaces. Errors related to data assumptions and manipulations could only be detected during program execution.
4. **Object-based and object-oriented programming languages**: These languages, including Smalltalk, C++, and Java, revolutionized the programming landscape by organizing programs around classes and objects. The physical structure of applications in these languages resembled graphs rather than trees, with classes and objects as the fundamental building blocks. They provided better support for data abstraction and strong typing, paving the way for scalable and maintainable software development.

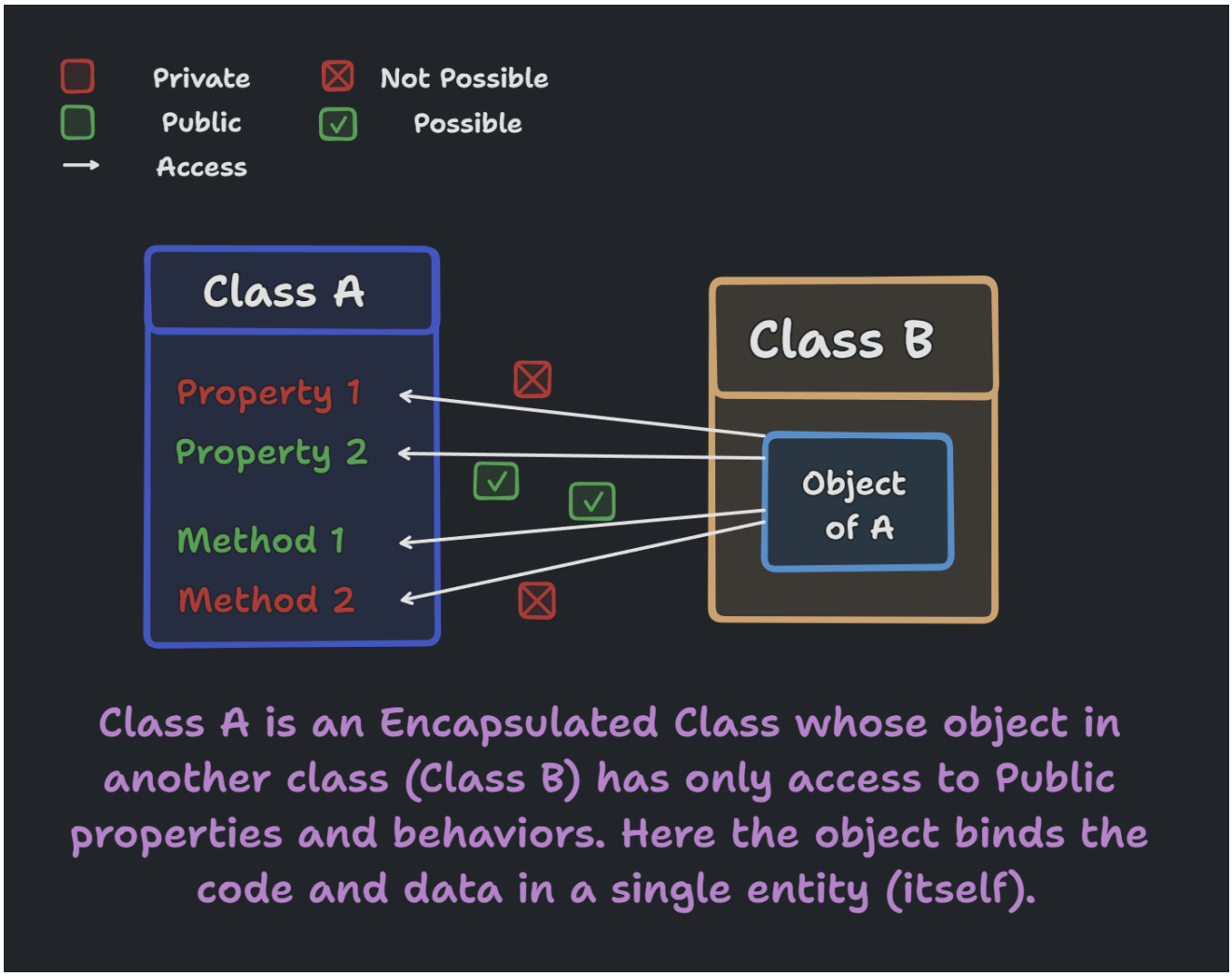
"Bricks" or Elements of Object Model and their Applications:

The Elements:

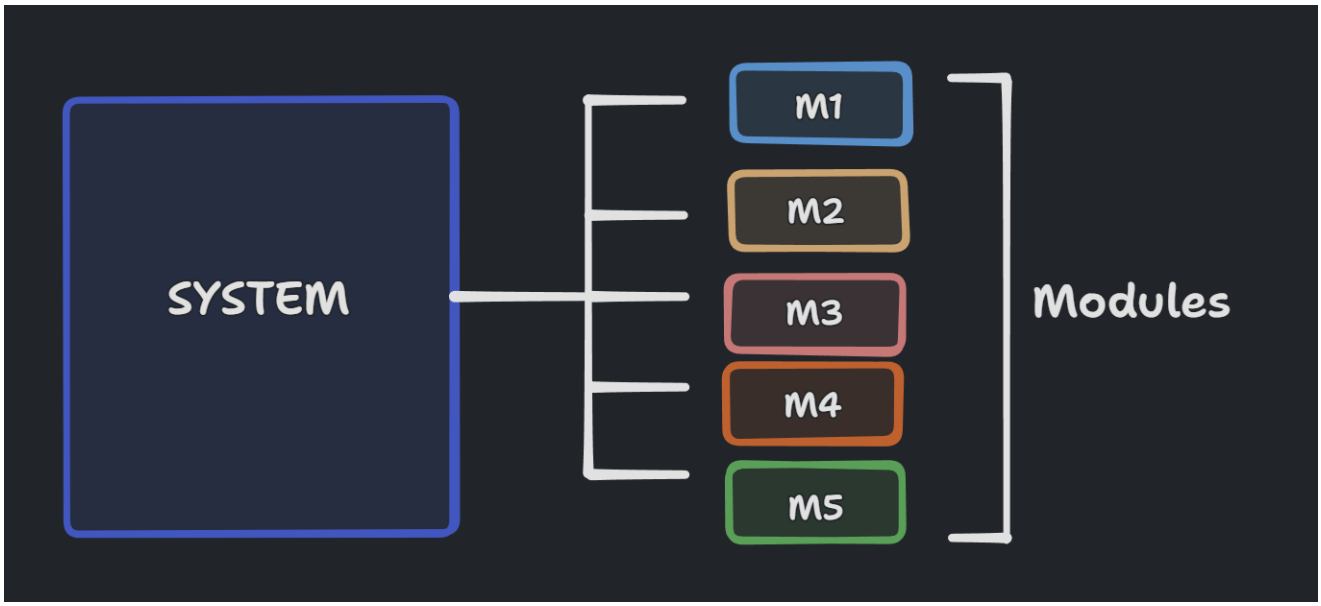
1. **Abstraction**: Abstraction involves simplifying complex reality by modeling classes appropriate to the problem, hiding unnecessary details while highlighting essential features. This allows developers to focus on relevant aspects of a system.



2. **Encapsulation**: Encapsulation involves bundling the data (attributes) and methods (functions or procedures) that operate on the data into a single unit or class. This unit provides a controlled interface for interacting with the data, ensuring that the internal state of an object is protected and accessed only through well-defined methods.

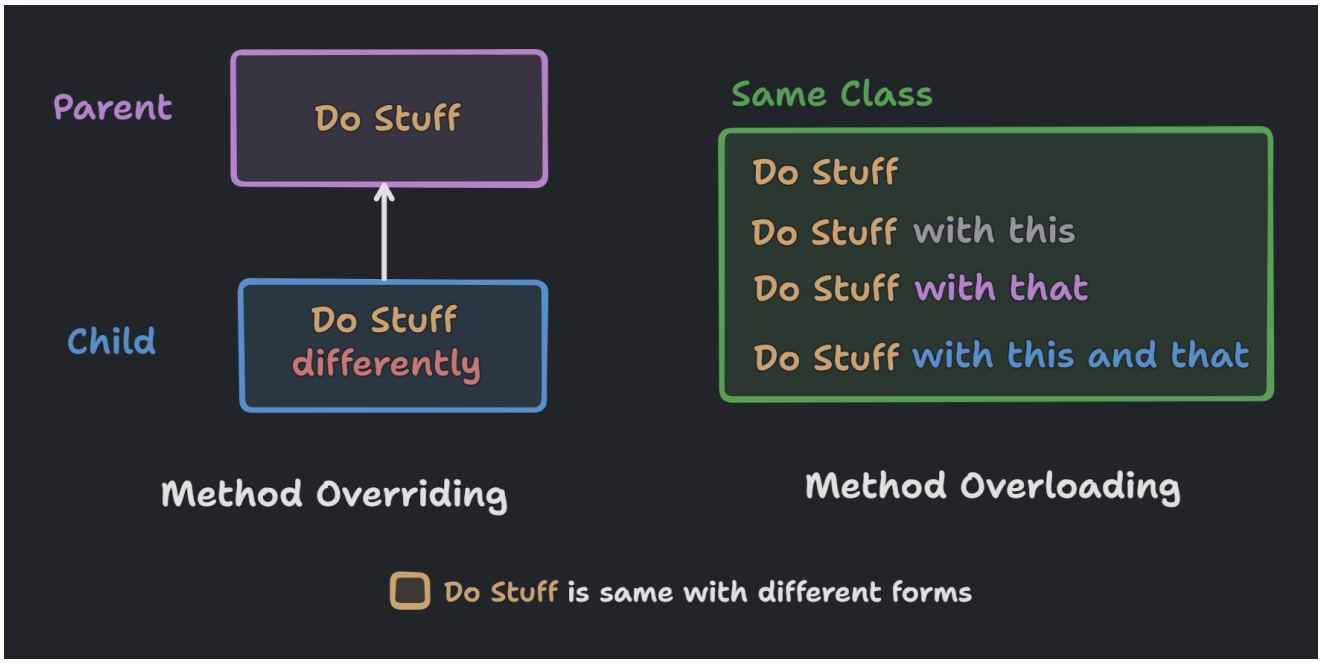


3. **Modularity**: Modularity refers to the decomposition of a system into smaller, manageable, and reusable units or modules. Each module encapsulates a specific functionality, promoting code organization, maintainability, and reusability.

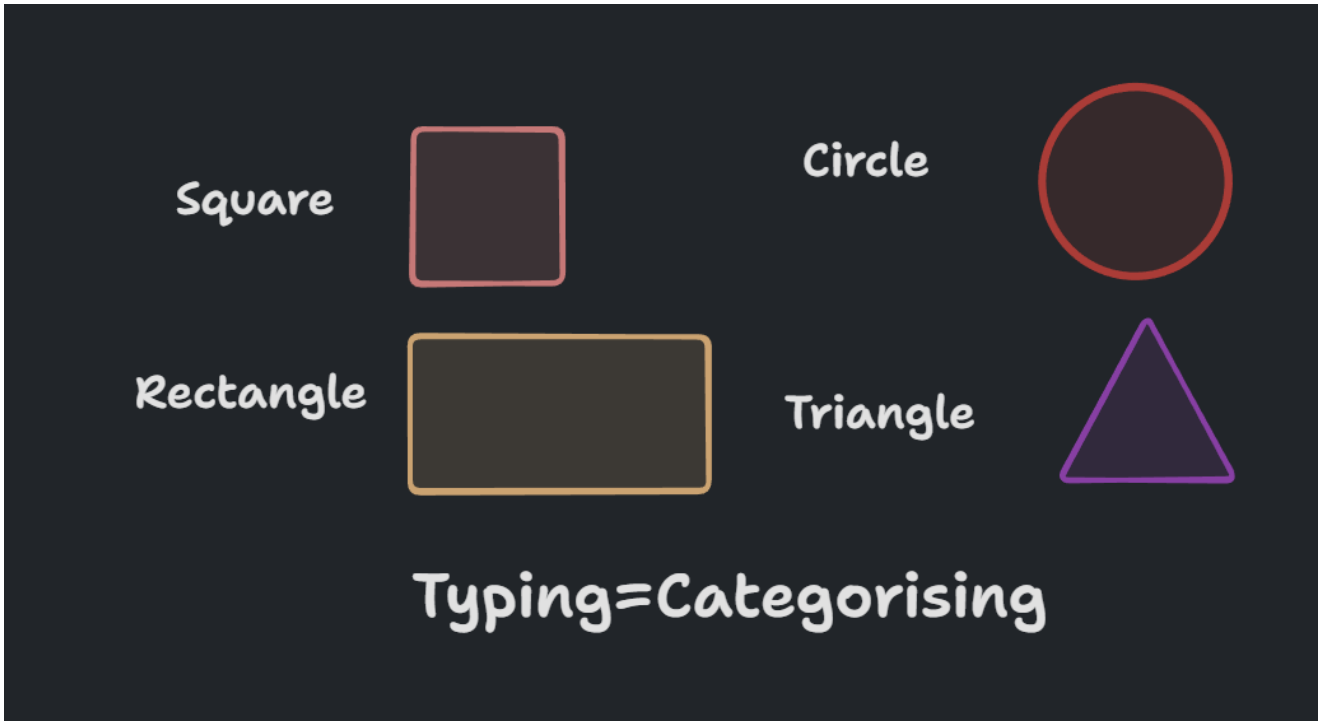


4. **Hierarchy**: Hierarchy establishes relationships between classes through inheritance, allowing the creation of specialized classes (subclasses) that inherit properties and

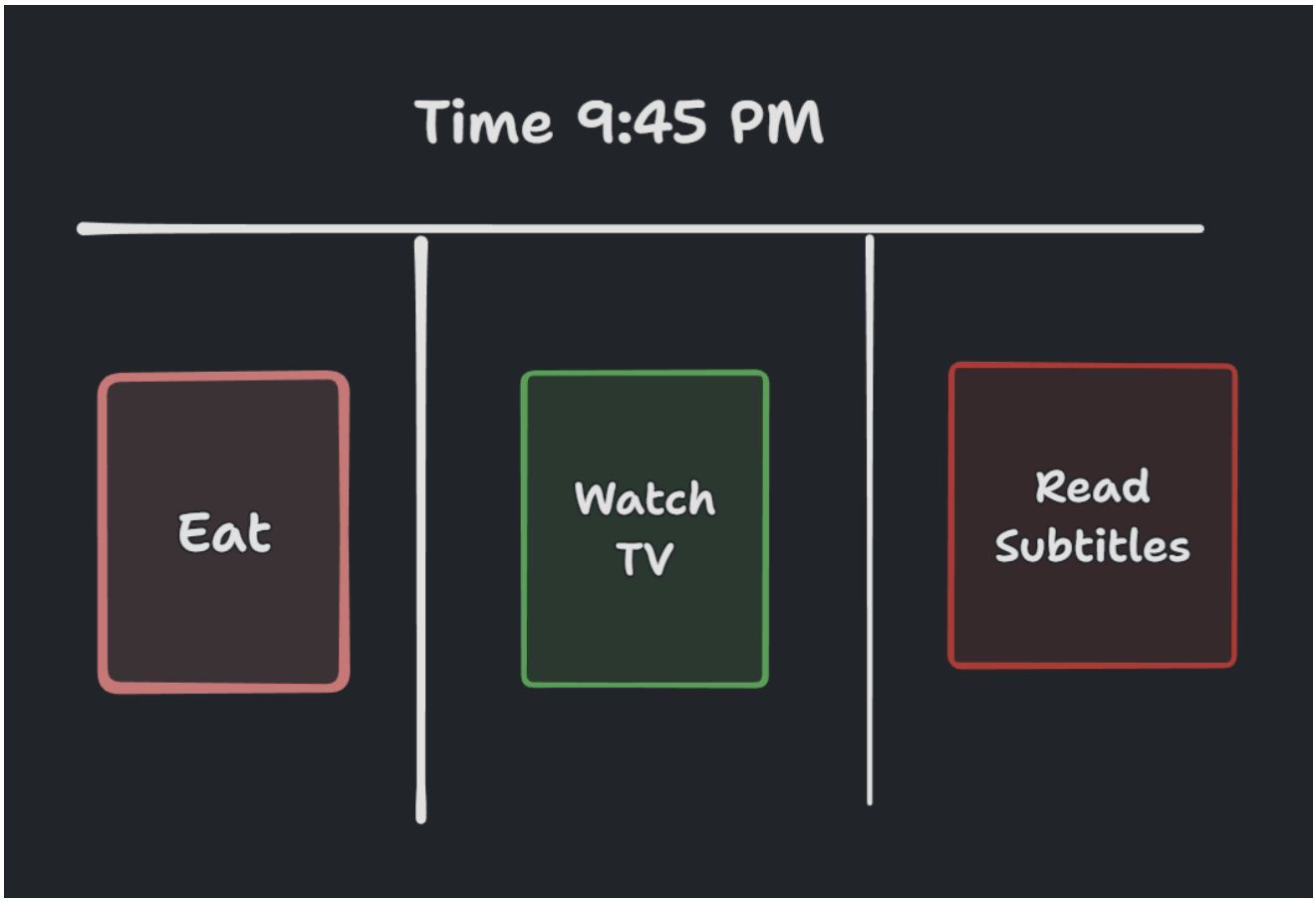
behaviors from more general classes (superclasses). This hierarchical structure promotes code reuse, extensibility, and conceptual clarity.



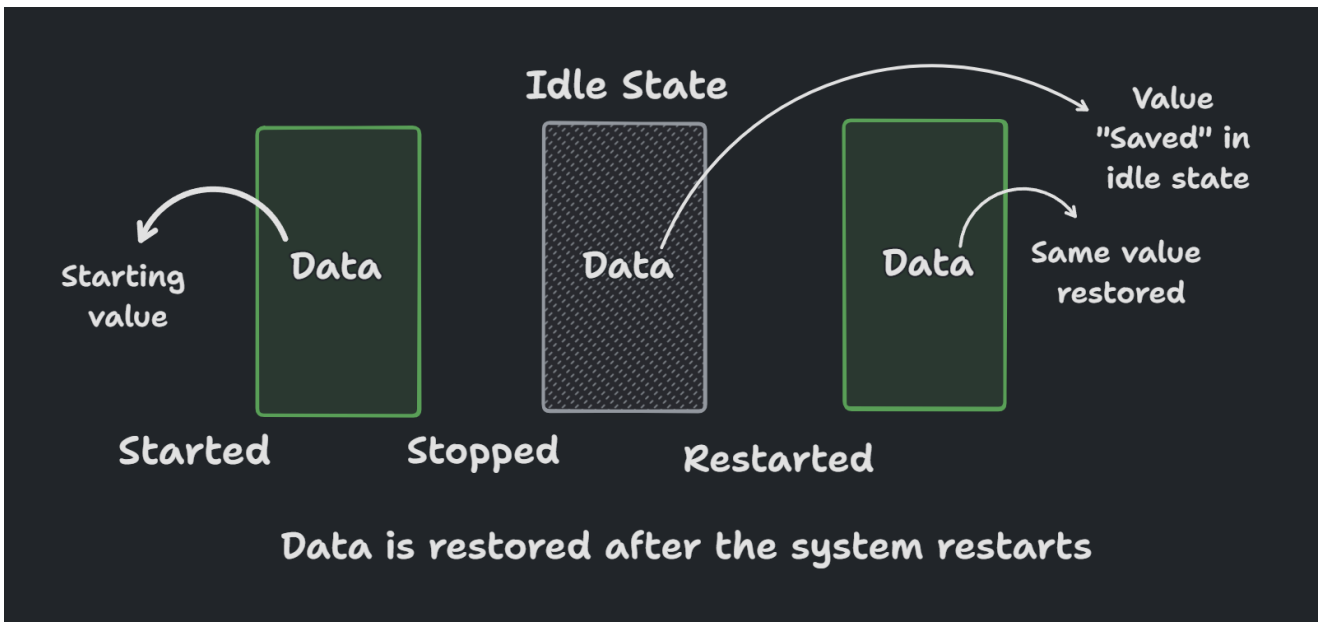
5. **Typing**: Typing involves specifying the types of objects that can be manipulated within a system. Strong typing ensures type safety and helps prevent errors by enforcing constraints on the interactions between objects.



6. **Concurrency**: Concurrency enables multiple tasks or processes to execute simultaneously, enhancing system responsiveness and resource utilization. Object-oriented concurrency mechanisms facilitate the creation, coordination, and synchronization of concurrent activities within a system.



7. **Persistence**: Persistence involves the ability to store and retrieve object state from a persistent storage medium, such as a database or file system. Object-oriented persistence mechanisms enable objects to maintain their state across different sessions or executions, supporting data persistence and application state management.



Their Applications:

1. **Abstraction**:

- **Application**: Abstraction allows developers to create models that capture essential aspects of a system while hiding unnecessary details. For example, in a banking application, a `BankAccount` class abstracts away the complexities of managing financial transactions, focusing on essential operations like deposits, withdrawals, and balance inquiries.

2. **Encapsulation**:
   - **Application**: Encapsulation ensures data integrity and promotes modular design by encapsulating related data and behavior within a class. For instance, in a payroll system, the `Employee` class encapsulates employee data (e.g., name, salary) and related methods (e.g., calculatePay, updateDetails), protecting the internal state from unintended modifications.
3. **Modularity**:
   - **Application**: Modularity facilitates code organization, maintenance, and reuse by breaking down a system into smaller, cohesive units or modules. In a web application, modules like authentication, user management, and content delivery can be developed independently, promoting scalability and flexibility.
4. **Hierarchy**:
   - **Application**: Hierarchy enables the creation of class relationships, fostering code reuse and extensibility. For example, in a geometric shapes library, a `Shape` superclass can have subclasses like `Rectangle`, `Circle`, and `Triangle`, inheriting common properties and behaviors while allowing specialization for specific shapes.
5. **Typing**:
   - **Application**: Typing helps enforce constraints on object interactions, promoting code correctness and robustness. In a type-safe language, such as Java or C#, specifying types ensures that operations are performed on compatible objects, reducing the risk of runtime errors and improving code reliability.
6. **Concurrency**:
   - **Application**: Concurrency enables the concurrent execution of tasks, improving system responsiveness and resource utilization. In a multi-user application, concurrency mechanisms allow multiple users to interact with the system simultaneously without conflicts, enhancing scalability and user experience.
7. **Persistence**:
   - **Application**: Persistence enables the storage and retrieval of object state, supporting data management and application state preservation. For example, in an e-commerce platform, object-oriented persistence mechanisms facilitate the storage of product information, user preferences, and order history, ensuring data consistency across sessions and interactions.

These elements collectively contribute to the development of robust, maintainable, and scalable software systems by promoting code organization, encapsulation, reuse, and flexibility.

Applying the Object Model:

The object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming. This does not mean that the object model abandons all of the sound principles and experiences of these older methods. Rather, it introduces several novel elements that build on these earlier models. Thus, by applying the object model we can get a number of significant benefits that other models simply do not provide. Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems: hierarchy, relative primitives (i.e., multiple levels of abstraction), separation of concerns, patterns, and stable intermediate forms. In our experience, there are five other practical benefits to be derived from the application of the object model.

Sources:

Object Oriented Analysis and Design by Grady Booch, Addison-Wesley.