

**Critical Thinking Assignment 5: Polymorphism - Using UML Class Diagrams and  
Pseudocode**

Alexander Ricciardi

Colorado State University Global

CSC470: Software Engineering

Dr. Vanessa Cooper

January 19, 2025

## **Critical Thinking Assignment 5: Polymorphism - Using UML Class Diagrams and Pseudocode**

In object-oriented software development, complex dependencies can exist between classes or objects (the instantiation of those classes) due to inheritance, association, and aggregation relationships (Labiche et al., 2000). Inheritance, in particular, often creates complex dependencies. However, polymorphism can be used to manage these dependencies particularly those related to inheritance. This essay explores the concept of polymorphism within computer science, more specifically within object-oriented software development, through the use of pseudocode and UML class diagrams.

### **Overview of Polymorphism in Computer Science**

The term polymorphism is derived from the Greek words "poly," meaning "many," and "morph," meaning "form" translating in English to many many-shape or many-forms. In computer science, polymorphism refers to the ability of data or a system to be processed in more than one form. Within computer science, polymorphism is an essential part of Object-Oriented Programming (OOP), it allows objects to respond to the same message (or method call) in their own way (Gupta, 2024). This polymorphic behavior can be achieved at compile-time, for example, through parameter overloading (static polymorphism), or at run-time through operator overloading and method overriding (dynamic polymorphism). Compile-time polymorphism, or static polymorphism, is when multiple methods have the same name but have different parameter numbers, types, or both (a method can handle values of many types without needing to be overridden). Whereas, compile-time polymorphism, or dynamic polymorphism, is when the functionality of a method is determined at runtime by the object invoking it. This type of polymorphism, method overrering, is typically associated with the inheritance relationship,

interfaces, and abstract class. It is important to understand that the relation between an interface and a class is a contract that specifies what the class should do based on a set of methods declared in the interface. Inheritance is a parent-child relationship or "is-a" relationship where the child class inherits the methods of the parent class. An abstract class is a class that can not be instantiated on its own, it is a superclass that serves as a blueprint for subclasses by defining methods, the relation between an abstract class and its subclass is a parent-child relationship.

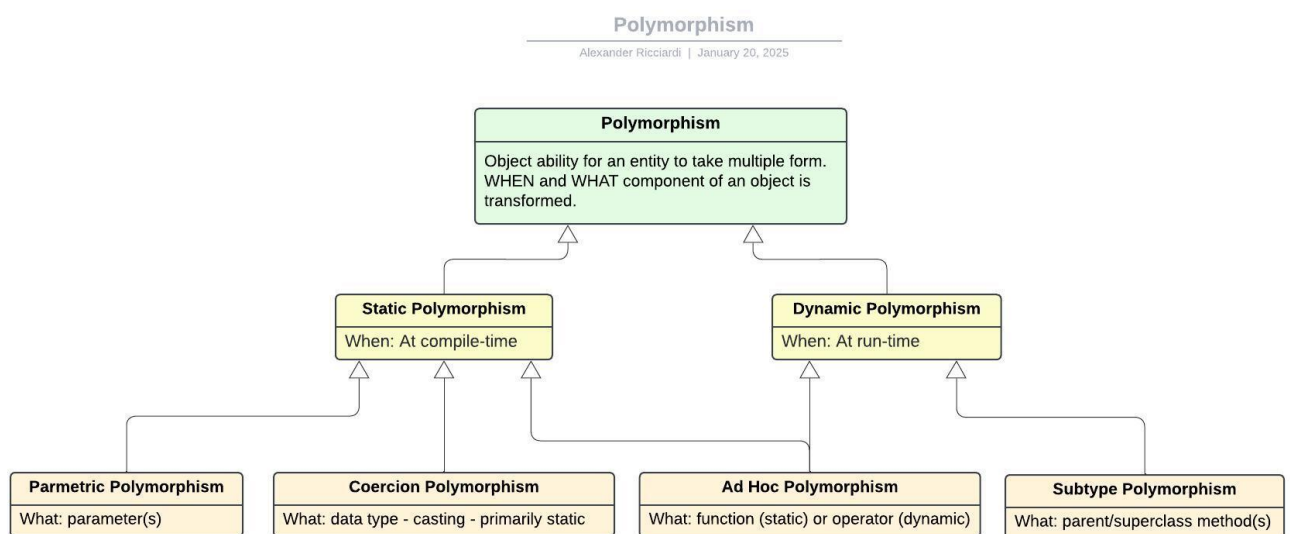
### Types of Polymorphism in OPP

Compile-time polymorphism and run-time polymorphism describe ‘*when*’ an object is transformed, but they also define ‘*what*’ component of an object is transformed. The ‘*when*’ and ‘*what*’ of an object transformation can be categorized into four different Polymorphism types.

Figure 1 illustrates these four types and how they are related.

**Figure 1**

*Types of Polymorphism*



*Note:* The relationships illustrated in the diagram by the arrows are parent-child relationships.

The inheritance relationships illustrated in Figure 1 are generalized and simplified illustrations of the relationships that exist between the different types of polymorphism. These relationships can be more complex depending on the programming language used to develop a particular software.

Table 1, below, shows a more detailed description of the different types.

**Table 1**

*Types of Polymorphism Descriptions*

Polymorphism Type	Description	When It Transforms	What Transforms	Mechanism/Example
<b>Subtype Polymorphism – Inheritance</b>	The object is polymorphic. It is the most used type in OPP.  A child's class overrides the parent class method behavior.	Run-time. Method overriding is performed when the program is running.	The object itself (i.e., its concrete subtype)	An interface or a parent class can have multiple subclasses that each implement or override methods differently.
<b>Ad Hoc Polymorphism</b>	One function or operator name can behave differently for different types, such as the + operator for int vs. String.	Run-time for operator (e. g. '+' '-' ) overloading and compile time for functions - early binding	The function or operator (act differently for different data type)	Operator overloading in C++ (e.g., + for integers or strings); certain function/method overloading patterns in Java, C++, etc.
<b>Parametric Polymorphism - Method - parameter overloading.</b>	A single piece of code (e.g., a generic function/method or data structure) can handle values of many types without needing to be re-written.	Typically compile-time, though may be later bound (depending on language)	The function/method in a more generic sense (type parameters or generics)	Generics in Java (List<T>), templates in C++, or universal functions in functional languages.

<b>Coercion Polymorphism - Casting</b>	An object or value is coerced (converted) from one type into another—essentially, the type itself changes.	Occurs at compile-time or sometimes at run-time for implicit casts	The type of the data (int → double, etc.)	Casting primitives (e.g., int to double), or upcasting/downcasting objects
--	--	--	---	--

*Note:* Data from several sources (Johnson, 2020; Gupta, 2024; Umbarger, 2008)

As shown in Table 1 defining the specific type of polymorphisms used by a relationship can be challenging and is often related to the programming language used to develop the software.






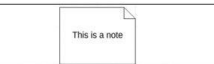
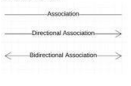
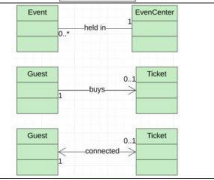
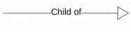
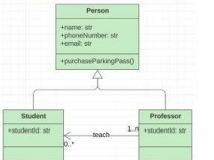
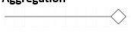
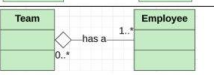
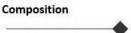
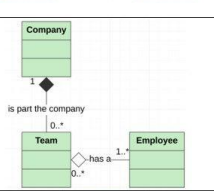
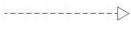
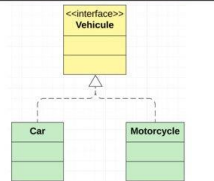

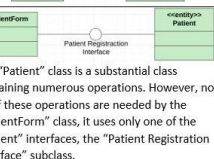
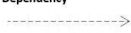
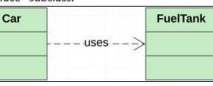
### **Polymorphism in Software Development**

In software engineering, polymorphism is a fundamental concept that is defined as “the ability of an instantiated object (at runtime) to understand and interpret the message sent from a calling object. This interpretation of a message by an object depends on its own characteristics and definition” (Unhelkar, 2018 b, p. 182). When incorporating polymorphism in software design tools such as pseudocode and Unified Modeling Language (UML) class diagrams are often used to visualize and document polymorphic relationships. Pseudocode is defined as a step-by-step description of an algorithm (Kartik, 2024). It helps programmers to design easy-to-understand and easy-to-read code solutions to the problems.

### **UML Class Diagram Overview**

UML class diagram is one of six types of structural diagrams used in software engineering (IBM, 2021). They are one of the most important diagrams as they can be used to model a software system's static structure (Shmallo & Shrot, 2020). In other words, they are the blueprints of systems and their subsystems (Ricciardi 2025). Moreover, in software engineering, they can be used within the problem space and the solution space. The table below describes the different components found in UML class diagrams.

**Table 1***UML Class Diagram Notation*

Component	Definition	Example:
<b>Class</b> 	Represents an object or objects that share a common structure and behavior.	Person, Doctor, Department.
<b>Attributes</b> 	Represent properties of a class that describe a range of data values that instances of the class may hold.	+name: str +phoneNumber: str +email: str
<b>Operations</b> 	Represent Functions or methods that can be applied to or by objects of a class.	+purchaseItems()
<b>Multiplicity</b> 	Indicates the number of instances of one class linked to one instance of another class.	1..*, 0..*
<b>Notes</b> 	Represents annotations, it is used to add comments or explanations to a diagram.	
<b>Association</b> 	Represent a relationship between two classes where objects of one class are connected to or use the services of objects of another class.	
<b>Inheritance</b> 	Represents a relationship where one class (subclass) inherits the attributes, operations, and relationships of another class (superclass). Also known as generalization.	
<b>Aggregation</b> 	Represent a specialized form of association representing a "whole-part" or "has-a" relationship. Also referred to as "by reference" implying that the relationship is only a reference (pointer) to an object	
<b>Composition</b> 	Represent a stronger form of aggregation where the "part" cannot exist independently of the "whole."	
<b>Realization</b> 	Represents a relationship between an interface and the class that realizes or implements it.	
<b>Interface</b> 	Represents a collection of operation signatures without implementation details.	 <p>The "Patient" class is a substantial class containing numerous operations. However, not all of these operations are needed by the "PatientForm" class, it uses only one of the "Patient" interfaces, the "Patient Registration Interface" subclass.</p>
<b>Dependency</b> 	Represents a relationship indicating that one class (the client) depends on another class (the supplier). Changes in the supplier may affect the client.	

*Note:* From “UML Class Diagrams: Modeling Systems from Problem Space to Solution Space”

by Ricciardi (2025).

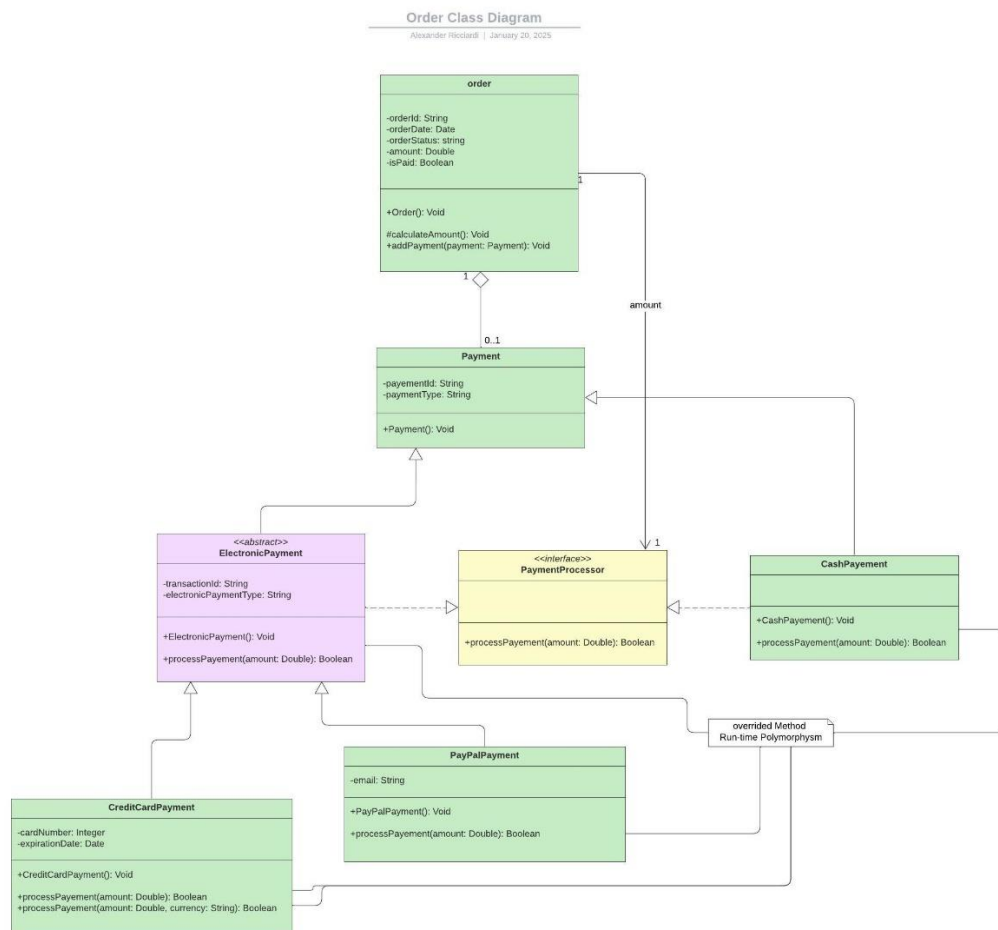
As shown in Table 1 UML class programs provide an extensive set of notations that can be used to represent classes (objects/entities), their attributes, their operations, and various types of relationships between them.

## Polymorphic Class Diagrams

Polymorphic class diagrams are used within the solution space to illustrate polymorphic relationships. These diagrams must contain inheritance relationships, interfaces, abstract classes, or a partial or full combination of the three. The figure below, Figure 2, is a UML class diagram of a simple order system.

**Figure 2**

*Order UML Class Diagram*



As shown in Figure 2, class diagrams do not show directly polymorphism details, they imply polymorphism such as subtype polymorphism by restating a parent method in a child class. For example, the child class CreditCarPayment restates the method processPayment(Double: amount) inherited from its parent abstract class ElectronicPayment implying that the method is overridden by it. To illustrate polymorphism in detail, developers often use pseudocode, which allows them to demonstrate the specific logic and behavior of each method.

### Example Pseudocode

Pseudocode is unusually used in combination with class diagrams. Depending on the developers' preference, pseudocode can be generated before, after, or during the creation of class diagrams; however, both tools are used to develop the software source code in the programming language chosen for the project. Below is an example of the simple order system pseudocode.

This pseudocode example focuses on illustrating, using mostly comments, where subtype polymorphism is implemented, rather than detailing the functionality of each method.

### Code Snippet 1

#### *Simple Order System Pseudocode*

```
//-----
// Order Class
//-----
class Order:
    private String orderId
    private Date orderDate
    private String orderStatus
    private Double amount
    private Boolean isPaid := false
    private Payment payment := Payment() // an object of the class Payment is instantiated
    // payment is declared as the parent type,
    // Subtype Polymorphism potential - Run-time Polymorphism

    // Constructor
    Order()
        //... Constructor logic
        calculateAmount() // Runs methods

    protected calculateAmount():
        amount := // ... Logic to calculate the amount

    public addPayment(String type, Payment paymentToAdd):
        payment := paymentToAdd(type, amount)
```



```

        isPaid := true

//-----
// Payment Class
//-----
class Payment:
    private String paymentId
    private String paymentType

    // Constructor
    Payment()
        //... Constructor logic

//-----
// PaymentProcessor Interface
//-----
Interface Class PaymentProcessor:
    /* The abstract keyword means the methods have no functionality implemented
       Subtype Polymorphism - potential (compile-time) */
    public processPayment(Double amount):
        //... No logic implementation

//-----
// ElectronicPayment Abstract Class
//-----
abstract class ElectronicPayment extends Payment implements PaymentProcessor:
    private String transactionId
    private String electronicPaymentType

    // Constructor
    ElectronicPayment()
        //... Constructor logic

    // Subtype Polymorphism - Run-time Polymorphism
    @override
    public processPayment(Double amount)
        //... Logic to process electronic payment
        returns true

//-----
// CashPayment Class
//-----
class CashPayment extends Payment implements PaymentProcessor:

    // Constructor
    CashPayment()
        //... Constructor logic

    // Subtype Polymorphism - Run-time Polymorphism
    @override
    public processPayment(Double amount)
        //.... Logic to process a cash payment
        return true

//-----
// CreditCardPayment Class
//-----
class CreditCardPayment extends ElectronicPayment:
    private Integer cardNumber
    private Date expirationDate

    // Constructor
    CreditCardPayment(I)
        //... Constructor logic

    // Subtype Polymorphism - Run-time Polymorphism
    @override

```

```

public processPayment(Double amount)
    /* ... Logic to process electronic payment
       ... Adds logic for to process credit card payment using cardNumber, expiryDate,
       and Amount */
    return true

/* Subtype Polymorphism - Run-time Polymorphism
   and Parametric Polymorphism (method-parameter overloading) - Compile-time
   Polymorphism */
@Override
public processPayment(Double amount, String currency)
    /* ... Logic to process electronic payment
       ... Adds logic for processing credit card payments using cardNumber, expiryDate,
       amount, and currency type */
    return true

//-----
// PayPalPayment Class
//-----
class PayPalPayment extends ElectronicPayment:
    private String email

    // Constructor
    PayPalPayment()
        //... Constructor logic

    // Subtype Polymorphism - Run-time Polymorphism
    @Override
    public processPayment(Double amount) : Boolean
        /* ... Logic to process electronic payment
           ... Adds logic to process PayPal payment using email */
        return true

//-----
// Main class - Example usage of the code
//-----
class Main:
    public static main():
        // 1. Create Orders (instantiate Order objects)
        Order myOrder1
        Order myOrder2
        Order myOrder3

        // 2. Create Payments (instantiate Payment objects)
        Payment payment1
        Payment payment2
        Payment payment3

        /* 3. Let's assign electronic and cash payments to payment objects
           The compiler will check for these payment objects' relationships Payment class
           and the PaymentProcessor interface. It will ensure that the objects inherit
           all the methods defined in the abstract class (ElectronicPayment)
           and the methods declared in the interface (PaymentProcessor)
           As the inherited methods are transformed by the ElectronicPayment class,
           This can be considered Subtype Polymorphism (Run-time Polymorphism)
           Note that the electronic payment objects are instantiated as
           Payment/ElectronicPayment
           Objects with ElectronicPayment class functionality, as ElectronicPayment is an
           abstract class and it cannot be instantiated on its own. */
        Payment1 := new ElectronicPayment()
        Payment2 := new ElectronicPayment()
        Payment3 := new CashPayment()

        /* 4. Let's assign credit card payments to electronic payments.
           This object inherited all the methods from Payment/ElectronicPayment objects,
           As the inherited methods are transformed by the CreditCardPayment class,

```

```

    this is can be considered a Subtype Polymorphism (Run-time Polymorphism) */
    Payment1 := new CreditCardPayment()
    Payment2 := new CreditCardPayment()

    // 5. Let make a payment
    payment2.processPayment(myOrder3.amount) // cash
    /* => subtype of ElectronicPayment using the interface PaymentProcessor (Subtype
        Polymorphism - run time) */
    payment1.processPayment(myOrder1.amount)
    /* => subtype of ElectronicPayment using the interface PaymentProcessor (Subtype
        Polymorphism - run time) */
    // Use the overloaded processPayment
    payment2.processPayment(myOrder2.amount, "Euro") // currency
    /* => subtype of ElectronicPayment using the interface PaymentProcessor (Subtype
        Polymorphism - run time)
        => and method - parameter overloading (Compile-time Polymorphism) */

    // 6. Assigned payment to order (aggregation)
    myOrder1.addPayment(payment1)
    myOrder2.addPayment(payment2)
    myOrder3.addPayment(payment3)

```

### Polymorphism at Run-time

Polymorphism at run-time is achieved through dynamic polymorphism, where the specific method to be executed is determined during program execution rather than at compile time. As shown by the Code Snippet 1 and Figure 1, in the context of SE, both tools are needed to show polymorphism at run-time. The class diagrams illustrate the relationships between classes, especially inheritance, a child-parent relationship, which is a fundamental concept in OOP and object-oriented design allowing classes to be derived from others, gaining their attributes and methods (Fallucchi & Gozzi, 2024). The inherited methods can be overridden by the child class at run-time, demonstrating subtype polymorphism (run-time polymorphism a type of dynamic polymorphism). This is also applicable to interfaces such as the interface classes in the Java programming languages. In the programming language C++, for example, “dynamic polymorphism is implemented by adding the virtual keyword to the function that is intended to be overridden” (Masri et al., 2018, p. 241). Additionally, class diagrams can imply subtype polymorphism by restating a parent method in a child class. On the other hand, pseudocode can show polymorphism by illustrating the detailed behavior of a piece of code or a method.

Therefore, in object-oriented design, both tools are essential to completely represent and fully model the polymorphic structures of a system.

### **Conclusion**

In object-oriented software development, the classes' inheritance, association, and aggregation relationships create complex dependencies and polymorphism can be used to manage these dependencies particularly those related to inheritance. In computer science, polymorphism is the ability of a system to offer the same interface for accessing different underlying data, objects, processes, or functionalities forms; it is a cornerstone of object-oriented programming. Therefore, it is essential for software developers to understand the role polymorphism plays in OOP, to be capable of identifying and utilizing different types, such as subtype, Ad hoc, parametric, and coercion polymorphism. Furthermore, it is crucial to understand the difference between static and dynamic polymorphism and how to use tools like class diagrams and pseudocode to illustrate polymorphic relationships within systems. Ultimately, a firm understanding of polymorphism, coupled with the use of modeling tools like UML class diagrams and pseudocode, allows developers to analyze complex systems and tackle challenging software design projects.

## References

- Fallucchi, F., & Gozzi, M. (2024, June 11). Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in Object-Oriented Programming. *Applied Sciences*, 14(12). <https://doi.org/10.3390/app14125083>
- Gupta, E. (2024, March 11). Difference between inheritance and polymorphism. Shiksha online. <https://www.shiksha.com/online-courses/articles/difference-between-inheritance-and-polymorphism-blogId-153349>
- IBM (2021, March 5). *Class Diagrams. Rational software modeler*. IBM Documentation. <https://www.ibm.com/docs/en/rsm/7.5.0?topic=structure-class-diagrams>
- Johnson, J. (2020, October 23). *Polymorphism in programming*. BMC Blogs. <https://www.bmc.com/blogs/polymorphism-programming/>
- Kartik, (2024, September 12). What is pseudocode: A complete tutorial. GeeksForGeeks. <https://www.geeksforgeeks.org/what-is-pseudocode-a-complete-tutorial/>
- Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., & Durand, M.-H. (2000). Testing levels for object-oriented software. *Proceedings of the 22nd International Conference on Software Engineering*, 136–145. <https://doi.org/10.1145/337180.337197>
- Masri, S. A., Nadi, S., Gaudet, M., Liang, X., & Young, R. W. (2018). Using static analysis to support variability implementation decisions in C++. *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*, 236–245. <https://doi.org/10.1145/3233027.3233043>
- Ricciardi, A. (2025, January 18). UML class diagrams: Modeling systems from problem space to solution space. Omegapy. <https://www.alexomegapy.com/post/uml-class-diagrams-modeling-systems-from-problem-space-to-solution-space>

- Shmallo, R. S., & Shrot, T. (2020). Constructive Use of Errors in Teaching the UML Class Diagram in an IS Engineering Course. *Journal of Information Systems Education*, 31(4), 282–293. <https://doi.org/https://www.jise.org/Volume31/n4/JISEv31n4p282.html>
- Umbarger, D. (2008). *An Introduction to Polymorphism in Java* [PDF]. Computer Science Teacher. The College Board.  
[https://apcentral.collegeboard.org/media/pdf/Intro\\_to\\_Polymorphism\\_in\\_Umbarger.pdf](https://apcentral.collegeboard.org/media/pdf/Intro_to_Polymorphism_in_Umbarger.pdf)
- Unhelkar, B. (2018 a). Chapter 1 — Software engineering fundamentals with object orientation. *Software engineering with UML*. CRC Press. ISBN 9781138297432
- Unhelkar, B. (2018 b). Chapter 11 — Class model-3: Advanced class designs. *Software engineering with UML*. CRC Press. ISBN 9781138297432