ICRIE 2023 4TH INTERNATIONAL CONFERENCE ON RECENT INNOVATIONS IN ENGINEERING

Proceedings of the 4th International Conference on Recent Innovation in Engineering
ICRIE 2023, University of Duhok, College of Engineering, 13th – 14th September 2023
(Special issue for Passer journal of basic and applied sciences)
Paper No. 33

Exploring Polymorphism: Flexibility and Code Reusability in Object-Oriented Programming

Hussein Mohammed Ali[1], Mahmood Yashar Hamza[1], Tarik Ahmed Rashid[2]
[1] Department of Computer Engineering at Tishk International University, Iraq
[2] Computer Science and Engineering at University of Kurdistan Hewler, Iraq
hussein.mohammed@tiu.edu.iq; mahmood.yashar@tiu.edu.iq; tarik.ahmed@ukh.edu.krd;

ABSTRACT

A key concept in object-oriented programming (OOP) is polymorphism, which enables objects or references to take on different forms depending on the situation. To allow for the creation of numerous sorts of objects, variables, and methods, it integrates ideas like virtual functions, function overriding, and function overloading. By enabling several implementations of the same function within parent classes, polymorphism promotes scalability and improves code clarity. Understanding how polymorphism is used in programming and how class objects may access and use it is crucial. This paper covers the idea of polymorphism, examines its uses, benefits, and drawbacks, and offers examples to show how it is used in OOP. Polymorphism allows for quick development, reuse of code, and flexibility in object-oriented programming paradigms.

KEYWORDS: Child Class, Code Reusability, OOP, Parent Class, Polymorphism.

1 INTRODUCTION

Polymorphism is a crucial aspect of object-oriented programming (OOP) that enhances code readability and reusability. It allows objects or methods to operate in multiple forms and adapt to different conditions within a program's code. Polymorphism uses fundamental concepts like virtual functions, function overloading, and function overriding to allow for a wide range of variables, objects, and methods. In OOP languages, an interface serves as the control for both classes and objects. Because of polymorphism, this approach may be used in a variety of situations. Because of this versatility, programmers may write solutions that handle a variety of data types and attitudes, which improves code readability and scalability. Programmers may efficiently build and use objects from several classes while adjusting properly based on the distinct characteristics of each object by using polymorphism. This makes it possible for objects to interact and communicate dynamically, ensuring that each object behaves in accordance with its class. Additionally, polymorphism has several advantages for software developers. By enabling programmers to use pre-existing classes, functions, and methods in a range of contexts, it allows code reuse. This time-saving method increases productivity and provides strong programming abilities [1] [2] [3]. Additionally, polymorphism facilitates software testing and speeds up runtime; many previous software initiatives have addressed this procedure [4] [5]. The following sections of this research paper are arranged as follows: in section 2, it will delves into the background of polymorphism, in section 3, it will address its limitations and problem statements, in section 4, it will discuss its numerous applications in related articles, in section 5, it will explain its potential drawbacks, and finally, in section 6, it will provide code samples that illustrate how polymorphism can be used in practical applications.

2 BACKGROUND OF POLYMORPHISM

One of the key points of object-oriented programming (OOP) is polymorphism. It gives programmers the option to combine methods from many classes, which encourages code structure and reuse. A parent class (or superclass), several child classes (or subclasses), and at least one common method are used to accomplish the behaviour of polymorphism. As illustrated in Figure 1, the hypothetical classes "Helicopter" and "Airplane," for instance, are both subclasses of the parent class "Aircraft." The parent class defines a method named "fly," which is used by these classes. Although they use the same function, each class varies in how it is implemented to fit its unique qualities [6]. Polymorphism in OOP enables the interchangeability of objects from several classes that derive from a single parent class. This implies that objects may be handled consistently regardless of the class type they belong to. Polymorphism is a powerful feature since it allows for the replacement of one object with another while still preserving anticipated behaviours and functionality. Most often, subclasses inherit attributes and methods from a parent class to implement polymorphism. This creates a hierarchy between classes, with shared traits and behaviours being transmitted down along the line of inheritance [7].

```
//a second child class.
public class Airplane extends Aircraft {
    Body of the class
    }
    public void fly () {
//here the changes to the method
defined in the parent class are
implemented.
    }
}
```

Figure 1: *A second child class.*

This will create a new class named “Airplane” that's also a child of the parent class “Aircraft” and has its own modified “fly” method. Polymorphism is an effective way to organize multiple classes and have them all fall under separate categories defined by the parent classes they belong to [7]. Polymorphism enables programmers to build code that works with generic types, enhancing flexibility and extensibility. Polymorphism ensures that changes in particular implementations do not affect the general operation of the program by having programmers write to interfaces or abstract classes rather than individual implementations. Developers may create flexible, scalable, and maintainable code by utilizing polymorphism. It encourages code reuse, streamlines system design, and improves the program's general structure.

3 PROBLEM STATEMENT

In the field of modern software development, programmers are confronted with numerous challenges that need creative resolutions to optimize code efficiency and ensure its long-term maintainability. The issues discussed are related to the attainment of enhanced flexibility in the realm of object-oriented programming, which facilitates seamless code migrations across different contexts. In the pursuit of achieving a harmonious equilibrium between optimizing performance and managing code complexity, it becomes crucial to explore approaches that enable the smooth integration of various object kinds while maintaining the clarity and comprehensibility of the code. Investigating various approaches to address these difficulties might result in improved software quality and more efficient development procedures [8].

Classification and creation of objects has one potential limitation of this study is the difficulty in accurately identifying and constructing suitable objects that represent real-world ideas. One significant challenge is in determining the appropriate level of granularity for objects and their properties, which may prove to be a complex task, perhaps resulting in objects that are too detailed or excessively broad in scope. This factor may influence the maintainability and versatility of code.

4 RELATED WORK

This section will demonstrate which applications use polymorphism and the types of usage of polymorphism. Polymorphism is used in a variety of scenarios in Object-Oriented Programming (OOP), and there are two sorts of polymorphism: compile-time or static polymorphism and runtime polymorphism. These types of polymorphism cover a variety of behaviours, such as method overloading and method overriding. Compile-time polymorphism is accomplished by method overloading, which allows the same function to be executed with alternative argument sets and return types. The compiler in languages such as Java identifies the right method to invoke based on matching method signatures. This type of polymorphism is used when an object's capabilities must be limited as presented in the article [9].

4.1 Compile-time polymorphism is advantageous in the following situations

1. Each argument should have its own type.
2. The order of the parameters is adaptable.
3. The techniques should differ in terms of the number of arguments and parameters.

4.2 Static binding polymorphism invokes

Static binding polymorphism, which happens during compilation, calls the appropriate overloaded functions based on argument type and number compatibility. By accessing this information during compilation, the compiler selects which function to run. This type of polymorphism is achieved using static binding, often known as early binding and function overloading.

1. Overloaded functions when the parameter type and number are compatible.
2. The compiler can choose the correct function because it can access all this data during compilation.
3. Static binding (sometimes called early binding) and function overloading (also called operator overloading) which is represented in articles [10] [11].

4.3 Dynamic polymorphism or Runtime

At runtime, runtime polymorphism, also known as dynamic polymorphism, resolves a single overridden method call. Method overriding is a well-known application of runtime polymorphism, which occurs when a subclass implements a method inherited from the parent class, as presented in the article [12].

Runtime polymorphism has the following properties:

1. A subclass replaces a parent class method with its implementation.
2. The class transfers its method to another inherited function during runtime.
3. It is critical to remember the following polymorphism insights.
4. The overriding method's name, return type, and parameters (including order, type, and length) must all be the same.
5. Overriding a method using final or static modifications is not possible.
6. Access modifiers should be either more limited or kept the same, but not made more liberal.
7. The overridden method may generate fewer, more specific, or no checked exceptions.

Polymorphism is a beneficial technique in object-oriented programming languages, whether it is runtime polymorphism through method overriding or static polymorphism through method overloading. It enables the development of numerous variations of a system, each categorized to different needs. This adaptability improves program adaptability, encourages code reuse, and makes software development more productive, as has been shown in the article [12].

4.4 Static polymorphism or compile time

Method Overloading is a feature that can be found in Object-Oriented Programming languages. This feature allows the programmer to create several variants of a single method. These languages can offer static polymorphism because of this feature since it makes it viable. Even though they have the same names, each method has different capabilities. The presence of static polymorphism is more likely to take place when the following criteria are met:

1. All parameters should have different types.
2. The order of the parameters is flexible.
3. One method's argument count should differ from the other method's parameter count.

The static binding polymorphism will invoke the overloaded functions when there is a match between the type of arguments and the number of arguments. As a result, any of this information may be easily accessed at the relevant function chosen by the compiler when the build is being performed. Both methods, overloading and overloading, often referred to as static binding or early binding, are responsible for producing this effect, the example has been illustrated in the following article [13].

5 ADVANTAGES AND DISADVANTAGES OF POLYMORPHISM

Polymorphism is an essential field of OOP, as shown in Table 1, there are several benefits and problems of applying polymorphism, which have been briefly explained in the table below [13], also the code examples for advantages and disadvantages of polymorphism are presented in Table 2.

Table 1: *Advantages and disadvantages of polymorphism*

ADVANTAGES	DISADVANTAGES
Polymorphism, in object-oriented programming (OOP), offers the advantage of code reuse allowing developers to save time and create programs. With the use of classes and shared interfaces, programmers can test, validate, and repurpose existing code efficiently. This encourages modular programming techniques by building upon established frameworks. Furthermore, polymorphism plays a role, in enhancing the readability and organization of code. In writing implementations developers can create code that is more adaptable and easier to maintain by programming to interfaces or abstract classes. This approach brings about dependencies and increased modularity in the code fostering connections, between components. Consequently, designs become simpler, more modular, and easier to understand modify and maintain. Furthermore, polymorphism empowers developers to build applications that employ an approach, in situations. Polymorphism enhances the flexibility and adaptability of code by enabling programmers to write code that can handle types of objects through a shared interface. This approach promotes the development of scalable software architectures and simplifies system design.	While polymorphism offers advantages it is important to consider the drawbacks as well. One of these downsides is the added complexity that comes with code. As the number of classes and their interactions grow, understanding the execution flow and behaviour of objects might become more challenging. To address this complexity and ensure comprehension of the software meticulous design and documentation are essential. Another aspect to consider is the impact of polymorphism, on performance. When it comes to runtime polymorphism dynamic dispatch is introduced, which adds a layer of indirection and might result in a decrease, in overall execution speed. Modern programming languages and compilers have incorporated functionalities to mitigate this impact and the difference, in performance is usually negligible. When utilizing polymorphism, it becomes vital to assess the performance requirements of an application and carefully balance the trade-off, between flexibility and performance.

In the illustration of the code in Table 2, a hierarchical structure of classes is shown, showcasing the concept of code reusability via the use of polymorphism. The basic class Vehicle has a universally shared

method, startEngine(), which applies to all types of vehicles. The subclasses Car and Bike inherit the StartEngine () function from the Vehicle class as they extend it. The primary aspect to consider is that the method is first established in the parent class and may thereafter be used by all child classes, hence eliminating the need for repetitive coding. Within the main function, we instantiate objects of both the Car and Bike classes. The demonstration of polymorphism is achieved by using the StartEngine () function on various instances, illustrating the ability to reuse the same method across objects belonging to different classes. This example illustrates the enhanced efficiency achieved using a shared interface, namely a common method, across several object kinds. This instance serves as an illustration of the possible intricacy that might arise from the implementation of polymorphism. A hierarchical structure has been developed to represent animals and the noises they produce in a class-based system. The Animal basic class contains a polymorphic makeSound() function. The subclasses "Dog" and "Cat" override the function to offer their vocalizations. Within the main function, we instantiate objects of the Dog and Cat classes. By invoking the makeSound() method, we initiate the execution of the corresponding override methods. This is where the intricacy arises. Polymorphism facilitates the smooth transition between distinct subclasses. However, it is important to note that the behaviour of a method might change greatly, which may result in possible confusion and complexity when comprehending the flow of code, particularly as the class hierarchy expands.

Table 2: *Code examples for advantages and disadvantages*

ADVANTAGES CODE EXAMPLE

```
// Advantage: Code Reusability
class Vehicle {
   public void StartEngine(){
         // code to start the engine
   }
}
class Car extends Vehicle {
   // Additional Car-specific functionalities
}

class Bike extends Vehicle {
   // Additional Bike-specific functionalities
}
public class CodeReuseExample{
   public static void main(String [] args)
   {
      Car car = new Car();
      Bike bike = new Bike();

      car.startEngine();
      // Reusing the startEngine()
      // method from Vehicle class
      bike.startEngine();

      // Reusing the StartEngine()
      // method from Vehicle class
   }
}
```

DISADVANTAGES CODE EXAMPLE

```
// Disadvantage: Increased Complexity
class Animal {
   public void makeSound(){
         // code to make a generic animal sound
   }
}
class Dog extends Animal {
   public void makeSound(){
         // code to make a dog-specific sound
   }
}
class Cat extends Animal {
   public void makeSound(){
         // code to make a cat-specific sound
   }
}
public class ComplexityExample{
   public static void main(String [] args)
   {
      Animal animal1 = new Dog();
      Animal animal2 = new Cat();
      animal1.makeSound();
      // Output : Dog-specific sound
      animal2.makeSound();
      // Output : Cat-specific sound
   }
}
```

6 EXAMPLE OF POLYMORPHISM IN ACTION

Different programs contain complex relationships between groups of collaborating objects from different classes. To improve these interactions within object-oriented software, polymorphism is essential. It makes use of dynamic method search to postpone the method selection until runtime and enables an object reference to communicate with objects belonging to other classes. Subclass methods can override and replace behaviour inherited from the superclass through method overriding. Instead of relying exclusively on inheritance, this separation between subclasses uses polymorphism, which promotes the flexibility and extensibility of the code [14]. As shown in Figure 2 of polymorphic behaviour. Consider a situation where there is a parent class called "Fruit" and two child classes called "Apple" and "Orange." Both the Apple and Orange classes override the "toString()" method and derive from the Fruit class.

The override toString() function of the Fruit class, the illustration below provides a general description. The toString() method is replaced with an Apple-specific implementation in the Apple class and an Orange-specific implementation in the Orange class. Instead of using the defined type, the appropriate version of the method is dynamically bound at runtime based on the actual object type. The program's output shows how polymorphic behaviour works, with each object exhibiting the distinctive behaviour specified in its corresponding overridden methods. Through the interchangeable usage of objects from other classes, polymorphism enables flexibility, code reuse, and extensibility, as shown in this code example [15].

```
class Fruit {
   // Common method overridden by subclasses
   public String toString() {
      return "I am a Fruit.";
   }
}

class Apple extends Fruit {
   // Override method specific to Apple
   public String toString() {
      return "I am an Apple.";
   }
}

class Orange extends Fruit {
   // Override method specific to Orange
   public String toString() {
      return "I am an Orange.";
   }
}

public class PolymorphismExample {
   public static void main(String[] args) {
      Fruit fruit1 = new Apple();
      Fruit fruit2 = new Orange();

      System.out.println(fruit1.toString()); // Output: "I am an Apple."
      System.out.println(fruit2.toString()); // Output: "I am an Orange."
   }
}
```

Figure 2: *An example code for polymorphism.*

6.1 Dynamic Binding

Runtime polymorphism, which determines the right method to execute based on the actual type of the object rather than the declared type, uses dynamic binding as a major component. It allows polymorphic objects to be switched out for one another while a program is running, encouraging

substitutability and adaptability in object-oriented systems. Imagine a situation where there is a superclass called "Fruit," as well as several subclasses like "Apple" and "Orange." Consider that the Fruit class has a toString() method that we will override in the subclasses, as represented in Figure 3.

```
class Fruit {
    public String toString() {
        return "I am a Fruit.";
    }
}

class Apple extends Fruit {
    public String toString() {
        return "I am an Apple.";
    }
}

class Orange extends Fruit {
    public String toString() {
        return "I am an Orange.";
    }
}

public class DynamicBindingExample {
    public static void main(String[] args) {
        Fruit fruit1 = new Apple();  // Dynamic binding
        Fruit fruit2 = new Orange(); // Dynamic binding

        System.out.println(fruit1.toString()); // Output: "I am an Apple."
        System.out.println(fruit2.toString()); // Output: "I am an Orange."
    }
}
```

Figure 3: *Dynamic binding example code.*

In this illustration, we define two variables of type Fruit, fruit1 and fruit2. But we provide them instances of the Apple and Orange classes, respectively, at runtime. In this example of dynamic binding, the choice of whether to use a particular definition of the toString() method is left up to runtime. When fruit1.toString() is used during program execution, the real object assigned to fruit1 is an instance of the Apple class. The outcome of the dynamically bound method call, which is toString(), which is defined in the Apple class, is "I am an Apple." [16]. Similar to this, when fruit2.toString() is used, an instance of the orange class is given to fruit2. As a result, the dynamically bound method call calls the toString () method specified in the Orange class, which outputs the phrase "I am an Orange." Late method binding is made possible via dynamic binding, which makes sure that the right method implementation is selected based on the object's runtime type. This adaptability makes it possible to use objects from other subclasses interchangeably, enabling polymorphic behavior and encouraging code expansion [17].

6.2 Abstract Classes and Methods

Instead of adding a new method definition for Fruit objects, say one that returns the constant price that the Fruit object implements, you wanted to introduce a new method rather than overriding an existing one like toString(). To do this, we may create a getPrice() function for each Fruit subclass. This method could be applied to the Apple class, as seen in Figure 4.

```
public String getPrice(){

return("2.00$");

}
```

Figure 4: *Public string type.*

This may be necessitated by defining a particular getPrice() method in the Fruit class, which the subclasses could override as desired (we've seen this technique work with toString()), but Java offers an alternate strategy based on abstract methods for such circumstances. A method is declared but not defined by an abstract method. The abstract modifier is used to specify it, as seen in Figure 5.

The getPrice() function shown in Figure 4 can be implemented via abstract methods. The Fruit and Apple classes can be modified as seen in Figure 5 and Figure 6.

Fruit.java

```
abstract class Fruit {

public abstract String getPrice(); }
```

Figure 5: *Fruit class.*

```
abstract String getPrice(parameterList);
```

Figure 6: *Abstract string type.*

The relationship of this abstract class example is displayed in Figure *7*.

Apple.java

```
class Apple extends Fruit {

public String getPrice() { return "2.00$";
}}
```

Figure 7: *Apple class.*

The relationship between this Fruit abstract class and apple class example is displayed in Figure ***8***.

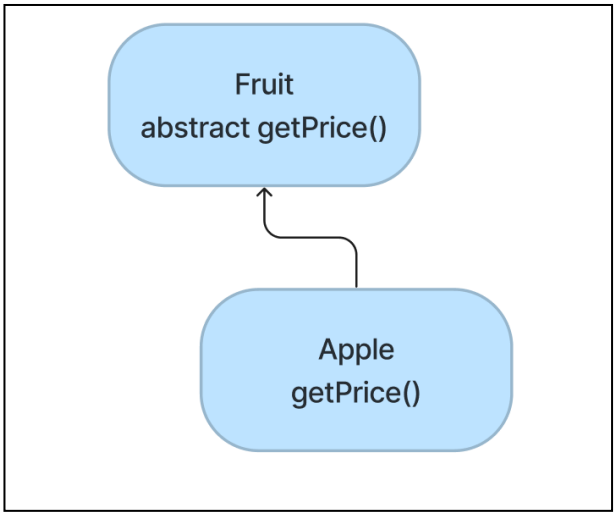

Figure 8: *Relationship of Fruit and Apples class.*

7 CONCLUSION

To summarize polymorphism, it allows code reuse, flexibility, and expansion in OOP. Developers may create versatile and scalable systems using polymorphism. This article covered polymorphism fundamentals, applications, pros, and cons. Polymorphism allows numerous objects, variables, and methods via virtual functions, method overriding, and method overloading. Polymorphism encourages the reuse of codes. Developers may reuse code and classes through inheritance, abstract classes, and common interfaces. This method improves efficiency, consistency, and modular programming. Polymorphism also simplifies coding. Programming to interfaces or abstract classes simplifies codebase expansion and maintenance. Polymorphism reduces dependencies, loosens coupling, and simplifies structures.

Polymorphism lets programs manage several object types with a single interface. This approach simplifies system design, scales, and lets developers adjust to changing demands. Polymorphism increases program complexity, requiring careful design and documentation to preserve comprehensibility. In dynamic dispatch and method lookup scenarios, performance should be considered. Method overriding and overloading are fundamental characteristics within the paradigm of object-oriented programming, which play a crucial role in enhancing the adaptability and expandability of software systems. The use of method overriding in object-oriented programming allows subclasses to implement unique behaviour while inheriting from a superclass. This approach facilitates the reuse of code and contributes to the overall maintainability of the software system. This ability enables software developers to generate more precise implementations while maintaining a uniform interface. Method overloading provides a technique for the creation of several methods that possess comparable functionality but vary in their parameter lists. Improving code readability and using simpler method names may contribute to increasing the user experience. Nevertheless, it is crucial to consider the possible confusion that may develop due to overloading, resulting in unintentional method resolution in some circumstances. The strategic use of method overriding and overloading enables programmers to create sophisticated and adaptable software solutions. Developers may achieve a balance between code readability, maintainability, and performance by selecting a suitable method that aligns with the unique objectives of a project. Both method overriding and overloading are essential components in achieving some of the objectives of polymorphism and object-oriented programming, despite their own merits and drawbacks. Finally, Polymorphism plays a role, in object-oriented programming serving as an aspect of software development. It enhances the ability to reuse code provides flexibility and readability and facilitates the creation of easily maintainable systems. By enabling binding and promoting cohesion polymorphism simplifies intricate code structures and closely resembles real-world situations. Consequently, it becomes an asset, for building resilient software solutions.

REFERENCES

[1] R. MILNER, "A Theory of Type Polymorphism in Programming," *JOURNAL OF COMPUTER AND SYSTEM SCIENCES,* vol. 17, no. 03, pp. 348-375, 1978.

[2] M. Loy, P. Niemeyer and D. Leuck, Java An Introduction to Real-World Programming, O'Reilly Media, 2020.

[3] S. K. Omer, S. D. Mirkhan, N. N. Hussein, A. Z. Ali, T. A. Rashid, H. M. Ali, M. Y. Hamza and P. Nedunchezhian, "Comparative Analysis of Encapsulation in Java And Python: Syntax And Implementation Differences," *Journal of University of Duhok,* vol. 26, no. 2, pp. 379-389, 23 12 2023.

[4] H. M. Ali, M. Y. Hamza and T. A. Rashid, "A Comprehensive Study on Automated Testing with The Software Lifecycle," *Journal of University of Duhok,* vol. 26, no. 2, pp. 613 -620, 24 12 2023.

[5] S. D. Mirkhan, S. K. Omer, T. A. Rashid, H. M. Ali, M. Y. Hamza and P. Nedunchezhian, "Effective Delegation And Leadership in Software Management," *Journal of University of Duhok,* vol. 26, no. 2, pp. 407-415, 23 12 2023.

[6] J. H. Solorzano and S. Ala, "Parametric Polymorphism for JavaTM:," *In Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* vol. 6, pp. 216-225, 1998.

[7] D. Umbarger, "An Introduction to Polymorphism in Java," in *AP Computer Science Curriculum Module: An Introduction to Polymorphism in Java*, Dallas, Texas, 2008, pp. 1-20.

[8] N. Singh, S. S. Chouhan and K. Verma, "Object oriented programming: Concepts, limitations and application trends," *5th International Conference on Information Systems and Computer Networks (ISCON),* pp. 1-4, 2021.

[9] A. Rountev, A. Milanova and B. G. Ryder, "Fragment Class Analysis for Testing of Polymorphism in Java Software," *IEEE Transactions on Software Engineering,* vol. 30, no. 6, pp. 372-387, 2004.

[10] J. L. Knudsen, "ECOOP 2001 - Object-Oriented Programming," *15th European Conference,* vol. 2072, 2001.

[11] S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram and T. D. Singh, "Comparative Analysis of Python and Java for Beginners," *International Research Journal of Engineering and Technology (IRJET),* vol. 7, no. 08, pp. 4384-4407, 2020.

[12] P. Canning, W. Cook, W. Hill, W. Olthoff and J. C. Mitchell, "F-Bounded Polymorphism for Object-Oriented Programming," *Proceedings of the fourth international conference on functional programming languages and computer architecture,* pp. 273-280, 1989.

[13] N. Milojkovic, A. Caracciolo, M. F. Lungu and O. Nierstrasz, "Polymorphism in the Spotlight: Studying Its Prevalence in Java and Smalltalk," *IEEE 23rd International Conference on Program Comprehension,* pp. 186-195, 2015.

[14] E. Ernst, "Family Polymorphism," *European Conference on Object-Oriented Programming,* pp. 303-326, 2001.

[15] P. Deitel and H. Deitel, Java How to Program, 11/e, Early Objects, Pearson; 11th edition, 2017.

[16] N. Liberman, C. Beeri and Y. B.-D. Kolikant, "Difficulties in Learning Inheritance and Polymorphism," *ACM Transactions on Computing Education (TOCE),* vol. 11, no. 1, pp. 1-23, 2011.

[17] J. Gross, K. Coogan, S. Heckman and G. S. de Oliveira, "Building a model of polymorphism comprehension," *ASEE Annual Conference & Exposition,* 2022.