A CASE STUDY REPORT

ON

[INHERITANCE]

Submitted by

23RH1A1240 M VENNELA

23RH1A1241 M VAISHNAVI

23RH1A1242 M MOUNIKA

Under the Esteemed Guide

Dr.AR.SIVA KUMARAN

Assistant Professor

In partial fulfillment of the academic Requirements for the Degree of

BACHELOR OF TECHNOLOGY

IT

DepartmentofHumanitiesandSciences

MALLA REDDY ENGINEERING COLLEGE FOR WOMEN

(Autonomous Institution-UGC, Govt. of India)

Accredited by NBA & NAAC with 'A+' Grade

National Ranking by NIRF Innovation – Rank band (151-300), MHRD, Govt. of India

AAAA Rated by Careers 360 Magazine in India's Best Engineering Colleges

Approved by AICTE, Affiliated to JNTUH, ISO 9001:2015 Certified Institution

Maisammaguda, Dhulapally, Secunderabad -500100.

www.mallareddyecw.com

FEBRUARY 2025

INDEX

CASESTUDY

A company has different types of employees, such as Full-time employees and Part-time employees. However, all employees share some common attributes like name, employee ID, and salary.

To avoid redundancy, we use inheritance where a base class (Employee) stores common properties, and two derived classes (Full Time Employee and Part Time Employee) extend the base class.

Explanation of Inheritance Usage\Base Class (Employee)Contains common attributes: name and employee Id. Has a constructor to initialize these attributes.Includes a display Details() method to print basic details.Derived Classes (Full Time Employee and Part Time Employee)Extend the Employee class.

Add specific attributes (salary for full-time, hours Worked & hourly Rate for part-time).Override the display Details() method and call super.display Details() to reuse the base class method.Main Class (Employee Management)Creates objects of both Full Time Employee and Part Time Employee.Calls display Details() to print information.

ABSTRACT

Inheritance is a fundamental concept with applications in both legal and programming domains. In a legal context, inheritance refers to the transfer of assets, rights, and obligations from a deceased person to their heirs. This process is governed by wills, succession laws, and legal frameworks that ensure a fair and lawful distribution of property. Inheritance disputes often arise due to unclear wills, competing claims, or complex family structures, requiring judicial intervention.

In programming, inheritance is a key principle of object-oriented programming (OOP), allowing a class (subclass) to derive properties and behaviors from another class (super class). This promotes code reliability, molecularity, and maintainability. Inheritance supports concepts like method overriding and isomorphism, enabling more efficient and structured software development. However, improper use of inheritance can lead to code complexity and reduced flexibility.

This study explores real-world inheritance cases in both domains, analyzing legal precedents and software development practices. It highlights the advantages and challenges of inheritance, emphasizing best practices to optimize its implementation.

PROBLEM IDENTIFICATION

In Object-Oriented Programming (OOP), inheritance allows one class (child) to acquire properties and behaviors from another class (parent). However, improper use of inheritance can lead to several problems. Here are some common inheritance-related issues:

1. Tight Coupling

The child class is tightly dependent on the parent class, making changes difficult.
If the parent class changes, all child classes must be updated.
Solution: Use composition instead of inheritance where possible.

2. Fragile Base Class Problem
Changes in the parent class may unintentionally break child classes.
Solution: Minimize modifications to base classes or use composition.

3. Improper Use of Inheritance (Is-A vs. Has-A Relationship)
Inheritance should be used only when there is a clear "is-a" relationship, not "has-a".
Example: A Car should not inherit from Engine; instead, it should have an Engine.
Solution: Use composition when "has-a" is more appropriate.

4. Diamond Problem (Multiple Inheritance Issue)
Occurs when a class inherits from two classes that both inherit from a common ancestor, leading to ambiguity.

Example:
```
class A { };
class B : public A { };
class C : public A { };
class D : public B, public C { }; // Diamond Problem
```
Solution: Use interfaces (in Java) or virtual inheritance (in C++).

5. Deep Inheritance Hierarchies
Having too many levels of inheritance makes code hard to read, maintain, and debug.
Solution: Flatten the hierarchy by using composition or interfaces.

6. Overriding and Hiding Issues
If a method in the parent class is overridden in a child class but not called correctly, it may lead to unexpected behavior.
Solution: Always use super.method-name() (Java) or base.method Name() (C#) where necessary.

7. Liskov Substitution Principle (LSP) Violation
A subclass should be sustainable for its super class without altering program behavior.
Example: If Bird has a method fly(), but Penguin (which extends Bird) cannot fly, this violates LSP.
Solution: Use interfaces and avoid forcing sub classes to implement inappropriate methods.
Would you like help with a specific problem related to inheritance in your code?

OBJECTIVE SETTING

Inheritance in Object-Oriented Programming (OOP) is used to promote code reuse, hierarchical classification, and extensibility. When designing inheritance in a system, the following objectives should be considered:

1. Code Re usability
Avoid code duplication by defining common attributes and behaviors in a base class.
Example: A Vehicle class with properties like speed and color, inherited by Car and Bike.

2. Establishing a Hierarchical Relationship
Ensure a proper "is-a" relationship.
Example: A Dog class inheriting from an Animal class because a dog is an animal.

3. Extensibility and Maintainability
Design the base class to be extendable for future modifications.
Example: If new types of animals need to be added, they should fit naturally into the hierarchy.

4. Method Overriding for Custom Behavior
Allow sub classes to modify inherited methods without changing the parent class.
Example: A Bird class has a method move(), but Penguin overrides it to walk instead of fly.

5. Promote Code Organization and Readability
A well-structured inheritance hierarchy improves readability and maintainability.
Example: Grouping common functionalities in a base class reduces complexity.

6. Implementation of Isomorphism
Enable method overriding and dynamic method dispatch.
Example: A Shape class with a draw() method can be overridden in Circle and Rectangle classes.

7. Adherence to OOP Principles

Follow SOLID principles, especially the Obelisk Substitution Principle (LSP) and Open-Closed Principle (OCP).
Example: A subclass should be replaceable for its super class without breaking the system.

KEY WORDS

1.extends: The primary keyword used to establish inheritance between classes.

2.super: A keyword used within a child class to access members of the parent class.

3.subclass: The class that inherits from another class (also called a child class).

4.superclass: The class that is being inherited from (also called a parent class).

5.Code reusability: Inheritance allows you to reuse properties and methods from a parent class in a child class, promoting code organization and efficiency.

6. this – Refers to the current instance of the class and can be used to differentiate between instance variables and parameters or to call another constructor in the same class.

7. final – Used to prevent inheritance and method overriding:

8. abstract – Defines an abstract class or method. An abstract class cannot be instantiated and may contain abstract methods (methods without a body).

INTRODUCTION

What is Inheritance?

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a child class (subclass) to acquire properties and behaviors (methods) from a parent class (super class). It enables code reliability, hierarchical classification, and polymorphism.

Key Features of Inheritance

1. Code Re-usability – Common functionality is defined once in a base class and reused in derived classes.

2. Hierarchical Relationship – Represents real-world relationships (e.g., a Car is a type of Vehicle).

3. Extensibility – A subclass can extend the functionality of a super class by adding new features.

4. Method Overriding – A subclass can redefine a method from the parent class to provide specialized behavior.

5. Isomorphism – A subclass can be treated as an instance of its super class, enabling dynamic method calls.

Types of Inheritance

1. Single Inheritance – A subclass inherits from one super class.

```
class Animal { }
class Dog extends Animal { }  // Dog inherits from Animal
```

2. Multiple Inheritance (Supported in C++ but not Java) – A subclass inherits from multiple super-classes.

```
class A { };
class B { };
class C : public A, public B { }; // Multiple inheritance
```

3. Multilevel Inheritance – A subclass inherits from another subclass.

```
class Animal { }
class Mammal extends Animal { }
class Dog extends Mammal { }  // Multilevel inheritance
```

4. Hierarchical Inheritance – Multiple classes inherit from the same super class.

```
class Vehicle { }
class Car extends Vehicle { }
class Bike extends Vehicle { } // Both inherit from Vehicle
```

5. Hybrid Inheritance (Combination of multiple types, supported in C++ via virtual inheritance).
Example in Different Languages

DISCUSSION

Discussion on Inheritance in OOP

Inheritance is a powerful feature in Object-Oriented Programming (OOP) that enables code reuse, extensibility, and maintainability. However, while it offers several benefits, improper use can lead to design issues.

Advantages of Inheritance

1. Code Re-usability – Reduces duplication by allowing a subclass to reuse methods and attributes from a parent class.

2. Hierarchical Organization – Represents real-world relationships (e.g., a Bird is-a Animal).

3. Extensibility – New functionalities can be added to the child class without modifying the parent class.

4. Polymorphic Support – A parent class reference can be used to child class methods dynamically.

5. Reduces Code Complexity – Encourages modular and structured coding.

Challenges and Limitations of Inheritance

1. Tight Coupling – Subclasses are strongly dependent on the base class, making changes risky.

2. Inheritance Depth Issues – Deep inheritance hierarchies (too many levels) make debugging difficult.

3. Improper Use (Is-A vs. Has-A Relationship) – Overusing inheritance where composition is better. Example: A Car has-a Engine rather than is-a Engine, so composition is better.

Best Practices for Using Inheritance

- Use inheritance only when there is a clear "is-a" relationship (e.g., Dog is-a Animal).
- Prefer composition over inheritance if a class "has-a" relationship (e.g., Car has-a Engine).
- Keep inheritance hierarchies shallow to avoid complexity.
- Follow SOLID principles, especially Liskov Substitution Principle (LSP) – subclasses should be replaceable for their base class without breaking the system.
- Use method overriding carefully to avoid unexpected behavior.

Open Discussion Points

1. When should we use composition instead of inheritance?
2. How can we avoid tight coupling in an inheritance hierarchy?
3. What are real-world examples where inheritance is the best choice?
4. How do different programming languages handle multiple inheritance.

OUT COMES

By implementing inheritance effectively in Object-Oriented Programming (OOP), several key outcomes can be achieved:

1. Code Re-usability

Reduces redundancy by allowing child classes to reuse parent class attributes and methods.
Example: A Vehicle class can define common properties (speed, color), and sub-classes (Car, Bike) inherit them instead of redefining.

2. Better Code Organization and Maintainability

Encourages modular programming by structuring code in a hierarchical manner.
Makes debugging and modifications easier, as changes in the parent class reflect in all child classes.
3. Scalability and Extensibility

New features can be added to a system without modifying existing code.
Example: If a new type of Electric-car is introduced, it can extend Car without affecting Vehicle.

4. Polymorphism Implementation

Enables method overriding, allowing different classes to provide their own implementations of a method.
Example: A Shape class with a draw() method, overridden by Circle and Rectangle.

```
class Shape {
   void draw() { System.out.println("Drawing a shape"); }
}
class Circle extends Shape {
   void draw() { System.out.println("Drawing a circle"); }
}
```

5. Hierarchical Representation of Real-World Objects

Helps in modeling real-world relationships like "is-a" (e.g., Dog is-a Animal).
Improves conceptual clarity in large software projects.

6. Faster Development and Reduced Effort

Speeds up development by allowing developers to reuse existing code.
Reduces the need for rewriting similar functionalities across multiple classes.

Tight Coupling – Overuse of inheritance can make classes overly dependent on each other.
Deep Inheritance Hierarchies – Too many levels of inheritance can make debugging difficult.
Improper Use (Is-A vs. Has-A Relationship) – Using inheritance when composition is better can lead to design .

CASE REPORT

Case Title: Efficient Code Reliability and Extensibility Using Inheritance in a Library Management System

Background:

A software development team was tasked with building a Library Management System (LMS) for a university. The system needed to manage different types of books, including Printed Books, E-Books, and Reference Books. The team decided to use inheritance to avoid redundancy and improve maintainability.

Problem Statement:

Initially, the developers created separate classes for different book types. However, they noticed a lot of duplicate code across classes. This led to higher maintenance efforts, inconsistent behavior, and difficulty in adding new book types.

Solution: Implementing Inheritance

To solve this, they implemented a base class Book with common attributes and methods. Different book types inherited from this base class.

Class Design Using Inheritance

```
// Parent class
class Book {
    String title;
    String author;
    int publicationYear;

    Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.publication Year = year;
    }

    void displayDetails() {
        System.out.println("Title: " + title + ", Author: " + author + ", Year: " + publicationYear);
    }
}

// Child class for Printed Books
class PrintedBook extends Book {
    int numOfPages;

    PrintedBook(String title, String author, int year, int pages) {
        super(title, author, year);
        this. num= pages;
```

```
    }

    void printBookDetails() {
        displayDetails();
        System.out.println("Pages: " + numOfPages);
    }
}

// Child class for E-Books
class EBook extends Book {
    double fileSizeMB;

    EBook(String title, String author, int year, double fileSize) {
        super(title, author, year);
        this.file Size MB = file Size;
    }

    void displayEBookDetails() {
        displayDetails();
        System.out.println("File Size: " + fileSizeMB + "MB");
    }
}

// Testing the classes
public class LibrarySystem {
    public static void main(String[] args) {
        Printed Book printed = new Printe dBook("Java Programming", "John Doe", 2020, 500);
        EBook ebook = new EBook("Python Basics", "Jane Doe", 2021, 2.5);

        printed.print Book Details();
        System.out.println("----------------");
        ebook.displayEBookDetails();
    }
}
```

Outcomes of Using Inheritance:

Code Re-usability – Common properties (title, author, publication Year) were defined once in the base class and inherited.

Scalability – New book types (e.g., Audio-book) could be added easily by extending Book. Simplified Maintenance – Fixing a bug or updating a method in Book automatically updated all sub classes.

Better Organization – The system had a clear "is-a" relationship (Printed Book is-a Book).

Lessons Learned & Best Practices

Use inheritance only when there is a clear "is-a" relationship.

Avoid deep inheritance hierarchies to prevent complexity.

Use super() to call parent class constructors and avoid redundant initialization.

Follow the Open-Closed Principle (OCP) – The system should be open for extension but closed for modification.

REFERENCES

1. Books & Academic Sources:

Grady Bo-och, James Rumba-ugh, and I-var Jacobson. Object-Oriented Analysis and Design with Applications, 3rd Edition, Addison-Wesley, 2007.

Bertrand Meyer. Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.

Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.

2. Online Documentation & Tutorials:

Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

Python Official Documentation: https://docs.python.org/3/tutorial/classes.html#inheritance

C++ Inheritance Guide (Geeks for Geeks): https://www. geeksforgeeks.org/inheritance-in-c/

3. Research Papers & Articles:

Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Lisk ov, Barbara, and Jeannette Wing. A Behavioral Notion of Sub typing. ACM Transactions on Programming Languages and Systems, 1994.

4. Programming Websites & Blogs:

Stack Overflow Discussions on Inheritance Best Practices: https: //stackoverflow.com

Medium Articles on OOP & Inheritance: https://medium.com/tag/oop