IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 18, April 2026
Impact Factor: 8.2

A Practical Implementation of Scholarship Management System Using Token System Based on Object-Oriented Programming (Java)

Vidyankshini Vibhute[1], Punashri Patil[1], Shravani Tanksale[1], Aman Umre[1]
Assistant Professor, Department of Information Technology[1]
Under Graduate Student, Department of Information Technology[1,3,4]
AISSMS's Institute of Information Technology, Pune, Maharashtra, India

Abstract: *With the increase in educational aspirants and limited financial resources, scholarship management is becoming a critical concern for institutions. Most scholarship processes involve manual systems which often lead to issues such as incorrect eligibility checks, improper fund allocation, data loss, and lack of effective tracking. Such manual systems cannot handle large volumes of applications effectively. This project aims to create and implement a Scholarship Management System using the token system with Object-Oriented Programming (OOP) using Java as the programming language. The project aims to create classes representing real-world objects, such as applicants, scholarship slots, tokens, and award bills. The project will use essential OOP principles, including encapsulation, constructors, method overloading, static variables, object references, and class interactions. The study illustrates the use of OOP to create effective and efficient software systems. OOP is more effective in creating management systems compared to procedural programming, especially if the system is likely to increase in size. The object-oriented programming approach is more organized and flexible*

Keywords: Object-Oriented Programming, Scholarship Management, Token System, Java, Encapsulation, Modularity, Static Variables, Constructor Overloading.

I. INTRODUCTION

A. Background of the Study

Modern educational institutions rely on structured scholarship management systems. Scholarship infrastructure needs to manage large numbers of applications, track eligibility, generate award information, and record transactions. Manual systems using tokens or logging systems are often inefficient. A software-based solution for managing scholarship infrastructure provides the necessary efficiency. However, a programming paradigm that allows for the clear and efficient modeling of real-world entities is necessary. Object-oriented programming meets this requirement.

Java, as a class-based object-oriented language, allows for the structured modeling of real-world systems using encapsulation, abstraction, inheritance, and polymorphism [1], [2].

B. Statement of the Problem

Traditional scholarship systems face several structural and operational problems: manual eligibility verification errors, inconsistent award systems, lack of modularity in system design, poor data encapsulation, and poor extensibility. A robust and extensible software architecture is necessary to address the following: modeling of real-world scholarship entities, data security and safety, easier modification and extension of the system, and automated token generation. The present research aims to address these issues with the development of a token-based Scholarship Management System using Object-Oriented Programming [3], [4].

DOI: 10.48175/568

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 18, April 2026
Impact Factor: 8.2

II. LITERATURE REVIEW

The evolution of programming paradigms has been widely discussed in the computer science literature. Past research in procedural programming has focused on the importance of sequential programming and function decomposition; however, these models have been found to be limited in terms of scalability and maintainability. Object-Oriented Programming (OOP) introduced a paradigm shift in software development by simulating real-world entities in software systems [5].

A. Historical Foundations

The term "Object-Oriented Programming" was coined and popularized by works of authors such as Tim Rentsch in his early writings and Bjarne Stroustrup in his works on the development of the C++ programming language [6]. These works pointed out the shortcomings of procedural programming in managing complex enterprise applications, especially with respect to code duplication, poor modularity, and difficulties in modifying existing code.

B. Practical Principles (Stroustrup)

Stroustrup's paper "What is Object-Oriented Programming" presents definitions of OOP features such as classes, inheritance, polymorphism, and abstraction [6]. The paper proves the practical applicability of OOP principles in simplifying complex applications by virtue of modularity, code reusability, and flexibility. The paper is highly applicable to scholarship management, as it deals with interactions between scholarship slot, token, and applicant, each with distinct responsibilities.

C. Formal Framework (Wegner)

Peter Wegner's "Concepts and Paradigms of Object-Oriented Programming" presents a formal framework of classifying programming languages as object-based and object-oriented [7]. The framework focuses on scalability and extensibility, which is essential for enterprise applications such as scholarship management. Wegner's differentiation between class-based and inheritance-based programming languages justifies the use of Java as the implementation language.

D. Applied Literature

Recent research works in applied computer science have shown the effectiveness of OOP in areas like banking systems, reservation systems, and student information management systems [8], [9]. These have shown the significance of encapsulation for secure data handling, inheritance for code reuse, and polymorphism for flexible code. Scholarship management systems have also been identified as a suitable case for demonstrating structured class interactions.

E. Gap in Existing Research

Even though eligibility verification and award mechanisms have been explored, the mapping of these implementations with theoretical OOP concepts like constructor overloading, static variables, returning objects, and arrays of objects has not been done comprehensively. This research attempts to bridge the gap by incorporating UNIT-II syllabus concepts into the research [10].

III. METHODOLOGY

A. Research Design

The methodology used in this research is qualitative, analytical, and implementation-oriented. It is a non-experimental study that uses the existing principles of software engineering and the Object-Oriented Programming concepts to create a practical system. The objective of using this methodology is to identify the causes of inefficiency in the traditional scholarship management systems and systematically break down these causes into structural and operational issues [1], [3].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 18, April 2026
Impact Factor: 8.2

B. Data and Information Sources

The sources of the data used for this research are based on academic literature and industry practices. Three categories of sources were identified:

1. Foundational Literature: Literature on the principles of procedural programming and Object-Oriented Programming, definitions of software complexity, metrics of software complexity, and quality attributes such as maintainability and reusability. Classic books like Stroustrup [6] and Wegner [7].
2. Comparative Case Studies: Academic publications on the comparison of procedural programming and Object-Oriented Programming; white papers on the analysis of code metrics to prove efficiency gains using OOP; academic publications on the comparison of the scalability of OOP systems with other programming paradigms [4], [5].
3. Industry Best Practices: Documentation and design patterns from major software ecosystems (Java, C++, Python); case studies of scholarship management and similar institutional systems implemented using OOP; best practices in modular design, token-based identification, and award automation [8], [9].

C. Analytical Framework

The analytical framework of this study is structured into four stages:

1. Identify and isolate specific limitations of the procedural paradigm.
2. Systematically map each core OOP Principle to the problem it was designed to solve (Encapsulation, Abstraction, Inheritance, Polymorphism).
3. Use deductive reasoning and existing literature to compare modularity, management of change, and isolation of faults in OOP-based versus procedural architectures [2].
4. Provide reference code to demonstrate increased clarity, reuse, and extensibility resulting from OOP Principles.

D. Research Tools

The primary means of conducting this research involves analyzing object-oriented design principles and evaluating their practical application within the development of a Scholarship Management System using a token-based mechanism. This study critically explores established OOP principles such as class structuring, encapsulation, construction, method overloading, static members, and object interactions, applied to the real-world scholarship management system [10], [11].

IV. RESULTS AND DISCUSSION

A. Core OOP Principles: Principle-to-Solution Mapping

Encapsulation: Global access to scholarship data causes instabilities and security issues. In the given code, attributes such as applicantName, scholarshipId, and tokenNumber are declared as private and accessed only through getter methods [3].

Abstraction: Excessive internal information regarding the eligibility checking and award mechanism causes complexity. Only the required public interface has been exposed, and internal details have been hidden [2].

Inheritance: Scholarship systems need to accommodate different applicant types such as undergraduate, postgraduate, and doctoral students, which share common attributes like applicant name and course type. Inheritance accommodates common attributes in a parent Applicant class [6], [7].

Polymorphism: Excessive conditional statements for award computation and eligibility checks cause inflexibility. Dynamic method binding through polymorphism enables seamless implementation of the award mechanism for different applicant types [10].

B. Practical Implementation in Java

The system includes the following main classes: Applicant (represents students with attributes like name and course type), Token (generates unique identifiers using static variables), ScholarshipSlot (models slot availability and

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 18, April 2026 Impact Factor: 8.2

occupancy), ScholarshipManager (manages allocation and freeing of slots), and AwardBill (calculates scholarship amount based on eligibility criteria). This implementation incorporates class definitions, default and parameterized constructors, method overloading, the this keyword, static variables for token generation, objects as return types and arguments, and arrays of objects for slot grouping [9], [11].

C. Code Implementation in Java

1) Class Definition and Encapsulation:

Applicant-related attributes are declared as private to preserve internal state integrity. Access occurs exclusively through defined getter methods, ensuring controlled interaction and encapsulation [3].

```
class Applicant {
private String applicantName; private String courseType;
public String getApplicantName() { return applicantName;
}
public String getCourseType() { return courseType;
}
}
```

2) Constructors and the this Keyword:

The parameterized constructor initializes all essential attributes during object creation. The this keyword resolves scope ambiguity between instance variables and constructor parameters, ensuring accurate value assignment [2].

```
public Applicant(String applicantName, String courseType) { this.applicantName = applicantName;
this.courseType = courseType;
}
```

3) Static Variables for Token Generation:

The Token class uses a static variable counter shared across all instances. This ensures every applicant receives a unique token number, demonstrating class-level state management [3].

```
class Token {
private static int counter = 1000; private int tokenNumber;
public Token() { tokenNumber = counter++;
}
public int getTokenNumber() { return tokenNumber;
}
}
```

4) Array of Objects:

The ScholarshipManager class maintains an array of ScholarshipSlot objects, enabling structured representation of multiple scholarship slots. This demonstrates grouping of multiple objects of the same class [9].

```
class ScholarshipManager { private ScholarshipSlot[] slots;
public ScholarshipManager(int totalSlots) { slots = new ScholarshipSlot[totalSlots]; for (int i = 0; i < totalSlots; i++)
slots[i] = new ScholarshipSlot(i + 1);
}
}
```

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 18, April 2026 Impact Factor: 8.2

5) Object as Return Type:
The findScholarship() method returns a ScholarshipSlot object, demonstrating the factory design pattern where a method returns an object reference [8].

```
public ScholarshipSlot findScholarship() { for (ScholarshipSlot slot : slots)
if (slot.isAvailable()) return slot; return null;
}
public ScholarshipSlot getSlot(int slotNumber) { if (slotNumber > 0 && slotNumber <= slots.length)
return slots[slotNumber - 1]; return null;
}
```

6) Object as Argument:
The assignApplicant() method accepts an Applicant object as an argument, demonstrating object passing between classes. Application time is stored using System.currentTimeMillis() [9].

```
public void assignApplicant(Applicant applicant) { this.applicant = applicant;
this.applicationTime = System.currentTimeMillis(); this.occupied = true;
}
```

7) Complete Menu-Driven Main Program:
The Main class integrates all classes through a menu-driven interface using Scanner for runtime input. It demonstrates object creation, method calls, and interaction between all defined classes [11].

```
import java.util.Scanner; public class Main {
public static void main(String[] args) { Scanner sc = new Scanner(System.in);
ScholarshipManager mgr = new ScholarshipManager(5); while (true) {
System.out.println("1. Register Applicant"); System.out.println("2. Process Application"); System.out.println("3. Exit System"); System.out.print("Choose: ");
int choice = sc.nextInt(); if (choice == 1) {
sc.nextLine(); System.out.print("Applicant Name: "); String name = sc.nextLine(); System.out.print("Course Type: "); String course = sc.nextLine();
Applicant a = new Applicant(name, course); ScholarshipSlot slot = mgr.findScholarship();
if (slot != null) { slot.assignApplicant(a); Token token = new Token();
System.out.println("Assigned Slot: " + slot.getSlotNumber()); System.out.println("Token Number: " + token.getTokenNumber());
} else { System.out.println("No Slots Available"); }
} else if (choice == 2) { System.out.print("Enter Slot Number: "); int slotNum = sc.nextInt();
ScholarshipSlot slot = mgr.getSlot(slotNum); if (slot != null && !slot.isAvailable()) {
double award = slot.processApplication(); System.out.println("Application Processed."); System.out.println("Award Amount: Rs." + award);
} else { System.out.println("Invalid Slot or Already Empty."); }
} else { System.out.println("System Closed."); break; }
}
sc.close();
}
}
```

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 18, April 2026
Impact Factor: 8.2

D. Code Explanation

• Classes: A class works like a blueprint that defines what an object should contain and what it can do. In this project, classes like Applicant, Token, ScholarshipSlot, and ScholarshipManager each have their own data and functions, helping organize the program properly [3].

• Creating Objects: To use a class, we create an object using the new keyword. When an applicant registers, an Applicant object is created and its details are stored. When a ScholarshipSlot object is created, memory is assigned for that slot.

• Static Variables: All members of a class share common class variables throughout the program. In the Token class, a static counter ensures all tokens generated are unique for all applicants, providing common state information [3].

• Method Overloading: Method overloading allows different methods with the same name but different parameters. In this system, award computation methods can be overloaded based on eligibility criteria, course type, or slot category. This is compile-time polymorphism [6].

• Method Overriding: Overridden methods are redefined in the child class. For example, if an Undergraduate or Postgraduate class inherits from Applicant, a calculateAward() method can be overridden. This is runtime polymorphism [7].

• Constructors: A constructor is a special method used to initialize the state of an object. Parameterized constructors initialize objects based on certain parameters; default constructors enable dynamic object generation [2].

• The this Keyword: The this keyword refers to the current instance of the class. It distinguishes instance variables from parameters, ensuring proper assignment of values during object construction.

• Object as Return Type: Methods can return references to objects. The findScholarship() method returns a ScholarshipSlot object — an example of the factory design pattern [8].

• Object as Argument: An object can be passed as an argument to other methods. An Applicant object is passed to the assignApplicant() method. This is a fundamental feature of object-oriented programming [9].

• Array of Objects: An array of objects groups multiple objects of the same class. The ScholarshipManager class maintains an array of ScholarshipSlot objects, enabling structured representation of multiple scholarship slots [10].

E. OOP Principle Mapping Summary

The principal OOP coordination elements are summarized in Table I below.

TABLE I: OOP Principle-to-Implementation Mapping

Mechanism	Implementation Context	Functional Role
Encapsulation	private fields + getters	Protect scholarship and applicant data integrity
Abstraction	Public interface only	Hide eligibility checking and award logic
Inheritance	Applicant parent class	Shared attributes for Undergraduate and Postgraduate types
Polymorphism	Overloading / Overriding	Flexible eligibility and award computation
Static Variables	ScholarshipCounter.count	Unique application ID generation
Object Return	findScholarship()	Factory-style scholarship assignment
Array of Objects	Scholarship[]	Structured scholarship record management

IV. CONCLUSION

The Scholarship Management System is an excellent example of the effective use of Object-Oriented Design with the help of the token-based mechanism. The classes, constructors, method overloading, object interactions, static members, and encapsulation make the scholarship management system more maintainable and scalable [3], [9].

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 18, April 2026 Impact Factor: 8.2

The object-oriented approach is better than the procedural approach in the context of extensibility and modularity. This research work has covered the objectives of UNIT-II of the Object-Oriented Programming curriculum and has proven the significance of OOP in the development of management systems in the modern world [1], [2].

Database integration, GUI interface development, and the use of complex design patterns like Singleton and Factory Method for widespread institutional deployment are some of the potential future improvements [8], [10].

V. ACKNOWLEDGMENT

The authors would like to thank the Department of Information Technology, AISSMS's Institute of Information Technology, Pune, for providing the academic environment and resources that supported this work. Special thanks are extended to the faculty members whose guidance and feedback contributed significantly to the development and refinement of this study.

REFERENCES

[1]. I. Sommerville, Software Engineering, 10th ed. Pearson, 2016.
[2]. G. Booch, R. Maksimchuk, M. Engel, B. Young, J. Conallen, and K. Houston, Object-Oriented Analysis and Design with Applications, 3rd ed. Addison-Wesley, 2007.
[3].] J. Bloch, Effective Java, 3rd ed. Addison-Wesley, 2018
[4]. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
[5]. B. Meyer, Object-Oriented Software Construction, 2nd ed. Prentice Hall, 1997.
[6]. B. Stroustrup, "What is Object-Oriented Programming?," AT&T Bell Laboratories Technical Report, 1991
[7]. P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," OOPS Messenger, vol. 1, no. 1, pp. 7-87, 1990.
[8]. B. Eckel, Thinking in Java, 4th ed. Prentice Hall, 2006.
[9]. P. Deitel and H. Deitel, Java How to Program, 11th ed. Pearson, 2018.
[10]. R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.
[11]. C. S. Horstmann, Core Java, Volume I — Fundamentals, 12th ed. Pearson, 2022.

BIOGRAPHY

Ms. Punashri Patil is an Assistant Professor in the Department of Information Technology at AISSMS's Institute of Information Technology, Pune, Maharashtra, India. Her research interests include object-oriented programming, software engineering, and academic software development.

Vidyankshini Vibhute, Shravani Tanksale, and Aman Umre are undergraduate students in the Department of Information Technology at AISSMS's Institute of Information Technology, Pune, Maharashtra, India. Their academic interests include Java programming, object-oriented design, and software construction.