Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

Original Article

Applying Object-Oriented Programming Principles in Large-Scale Software Development: A Quantitative Comparative Study of Different Approaches

Ruqaya Budalal[1]*, Sana Alghazali[2], Eman Alamin[2]

[1]*Department of Computer Science, Faculty of Computer Technologies, Benghazi, Libya*
[2]*Department of Computer Science, Faculty of Information Technology, University of Benghazi, Benghazi, Libya*
Corresponding Email. *ruqaya.budala@gmail.com*

Abstract
This study conducts a quantitative comparison of three software development methods used in developing large, high-volume systems: procedural programming, basic object-oriented programming, and advanced object-oriented programming with design patterns combined. The implementation of a Library Management System (LMS) using Python occurred in three parallel versions that contained identical functionality. All three applications were developed and then used to measure seven quality metrics, including Lines of Code (LOC), coupling, cohesion, testability, extensibility, maintainability, and readability. The results show that using advanced object-oriented programming methods with design patterns produced much better application quality than the other two methods. Advanced OOP scored an overall quality metric score of 88.3 out of 100, compared to 67.1 for Basic OOP and 31.3 for Procedural Programming. In addition to the quality metrics, 62 unit tests were conducted, and it was found that design patterns significantly improve the testability of an application and provide for better component isolation. The study recommends that for projects that are small (less than 500 LOC), use Procedural Programming; for projects that are medium in size (500–5,000 LOC), use Basic OOP; and for large-scale systems (5,000+ LOC) or multi-team development efforts, use advanced OOP along with design patterns.
Keywords. Object-oriented Programming (OOP), Design Patterns, Procedural Approach, Software Quality Metrics, Coupling And Cohesion, Maintainability, Library Management System.

INTRODUCTION

Research Problem

The field of software engineering is experiencing a never-before-seen pace of change at the moment. Software projects now consist of millions of lines of code and are composed of many different parts developed by numerous teams that work together in very different and constantly changing environments, which results in them being very complex and highly interconnected. In addition to all of this complexity, developers must make one fundamental decision before they write any lines of code: they must select a programming paradigm that allows for the intended future growth and durability of the program. This is not only a technical choice; it is also a strategic decision that will directly affect both development and maintenance costs, as well as the quality of software delivered to end users (Mukaramovich & Mamirovich, 2023).

An early system built with a simple procedural methodology may have been easy to build at the beginning; however, as the requirements increased, the system became a burden. With multiple functions interconnected and the reuse of data between functions resulting in dependence, when we make a change in one function, we risk creating a problem in another function or area of the system that may not have anything to do with the original change. Research throughout the software engineering profession has shown that 60% or more of the costs associated with the lifecycle of a software system are associated with maintenance rather than original development. As previously stated, the proper choice of the correct programming methodology at the start of a project is financially driven and thus should be considered carefully (Aratchige, Manujaya & Weerasinghe 2024). There needs to be a scientifically developed and methodologically documented study that determines the effects of programming methodologies and provides a source of measurement and comparison for objective analysis.

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

Scientific Background

Object-oriented programming is a technique used to program software, but it originated as a solution to an actual software crisis that was described by the software industry during the late 1960s and 1970s, when traditional procedures did not provide sufficient software development capabilities to meet the growing size and complexity of the software (Asaad & Avksentieva, 2024). Simula was first designed and implemented in 1967 and is credited with laying down the theoretical foundations for what we now refer to as modern object-oriented programming (OOP). Smalltalk was next to be developed in the 1970s, providing OOP with philosophical underpinnings based on four principles still being utilized today: encapsulation (the exposure and hiding of an object's internal workings), inheritance (reusing and extending existing code without altering it), polymorphism (the ability of an object to act differently in response to the same request), and abstraction (the ability to work with different objects through a common interface, regardless of how they are implemented) (Eigler, Huber, & Hagel, 2023).

In 1994, Eric Gamma and his co-authors wrote a book called "Design Patterns: Elements of Reusable Object-Oriented Software." This book served to advance software development significantly in the 1990s. This book lists 23 design patterns that help with the recurring problems of designing in OOP and classifies them by three major types of design patterns: Creational (e.g., Factory, Singleton), Structural (e.g., Decorator), and Behavioral (e.g., Observer, Strategy). The publication of this book caused a major cultural shift in the way developers communicate regarding design solutions, trailblazing widely used patterns that give developers a common reference and proven methods upon which to build maintainable and extensible systems. With leading frameworks today (e.g., Spring, Django, Laravel) being built on these design patterns, the SOLID Principles (created by Robert Martin) are a natural extension of this body of knowledge and provide developers with the basis of writing high-quality OOP code that will be easily tested on a regular basis (Rahman, Chy & Saha, 2023).

Research Importance and Knowledge Gap

Even though OOP (object-oriented programming) is well documented from an academic standpoint, most research tends to be either theoretical or descriptive in nature instead of providing actual measurable quantitative data on what a single integrated system consisting of multiple methods created using multiple methods together (Ngaogate, 2023). Additionally, a significant amount of theoretical comparisons use artificial rather than realistic examples to measure how complex systems and multiple, connected entities can be developed. Because there is a lack of examples, it is difficult for software engineers and architects to make evidence-based decisions when choosing between methodologies (Karanikolas, 2023).

The goal of this research project is to address an identified gap in the available research by developing a fully-functional library management system that is based on Python and uses three different library management system designs, allowing a direct, fair, and objective comparison. Such a comparison would provide a basis for eliminating any differences attributed to the programming languages used in developing the three library management systems. According to Piyawardhana et al. (2023), this direct, fair, objective comparison would result in a more accurate and valid analysis of the relative strengths and weaknesses of different approaches to developing library management systems.

In addition to comparing the relative number of lines of code produced during the development of the three systems, this research will evaluate each system according to a number of important software engineering quality metrics, such as the degree of coupling between components, internal cohesion, testability, extensibility, and maintainability. These measures ultimately determine the total long-term cost of ownership for libraries (Fallucchi & Gozzi, 2024).

Research Objective and Contribution

This research project is to create a library management system using three levels of programming: 1) traditional way of doing things which will use independent methods that will have access to globally shared data; 2) basic object-oriented programming (OOP) principles; and 3) advanced OOP techniques that will make use of design patterns (such as Factory, Observer, Strategy, Decorator, Singleton and Repository) along with a clearly defined layered service architecture. The results from the three experiments will provide a baseline for determining how well code quality improves based on the use of Object-Oriented Programming principles and also lead to the development of a

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

roadmap to help development teams choose which approach to take based on their project's size, requirements, and limitations (Babiuch & Foltynek, 2024).

LITERATURE REVIEW

Multiple previous research studies have examined the same subject matter from various perspectives and can be grouped into three broad categories: teaching Object Oriented Programming (OOP) and the difficulties that students encounter; software quality metrics; and design patterns for large systems.

To begin with: Education and OOP

The abstract nature of OOP was cited as one of the difficulties for students learning OOP by Krismadinata et al (2023), in their quasi-experimental study of 56 students in informatics (30 in an experimental group using a collaborative approach via Media Wiki and 26 in a control group). Their data demonstrated that the students who employed a collaborative learning approach, using MediaWiki, did see an increased level of programming success, with a reduction in the degree of perceived complexity of the abstract concepts included in providing abstraction, and were better able to demonstrate teamwork. This study also overlaps with the current study in that it validates the idea that the concepts of abstraction and encapsulation represent very real barriers to education and require a more applied method of showing representatives, doing so by building an application system that is both runnable and tangible measure for the purposes of being able to measure the output(s).

In the study of Sheta (2022), a complete examination of the four OOP principles -encapsulation, inheritance, polymorphism, and abstract data types- and how they affect the creation of flexible, reusable, and extensible software systems was discussed. Common design patterns such as Singleton, Factory, and Observer were described, and their long-term maintainability was addressed. This investigation provided a base for this investigation's research because it uses all or parts of all four principles and common design patterns. However, this investigation will expand the existing literature by providing quantitative measurements of how much each principle affects systems through an additional set of objective metrics as opposed to just describing the principles.

Second: Software Quality Metrics in Object-Oriented Systems

Developing a mathematical model to quantify maintainability of object-oriented systems was the goal of researchers Rochimah et al (2025), who created a framework consisting of mathematical equations for five core maintainability metrics of object-oriented software: Modularity, Analysability, Reusability, and Testability. They then utilized this framework on 17 code smells with three different refactoring methods, successfully demonstrating the ability of the framework to measure any improvement in the quality of the code after applying each of the three refactoring techniques. The current paper is similar to the study by Rochimah et al in that it utilises an objective metric-based evaluation; however, it differs from their study in that it compares programming approaches to each other at the design level instead of analysing and refactoring completed code.

Almogahed et al. (2023), in a study developing out of IEEE Access, also examined a recently identified issue, specifically in reference to how different refactoring techniques affect multiple quality metrics relative to one another. The authors utilised ten popular refactoring techniques on five case study applications of varying size and complexity as a basis for evaluating quality indicators using the QMOOD model. The conclusion from this study was that the varying effects resulting from the use of the refactoring techniques were a result of how they were applied, rather than the specific technique; this is another area that coincides with the current research, in that the programming approach established at the beginning of an application has an impact on the ease/ability to maintain at a later date due to the differences between the two stages of development, as opposed to waiting until the later stage to refactor.

Iyyappan and colleagues (2023) developed a model for software component selection that employs the Hexa-oval Algorithm as a means of evaluating interface density within modular systems; their study found that low coupling and high cohesion correlate with reduced complexity of the software product and increased reliability of the overall system. The dataset presented in their research forms a theoretical basis for my project in that I find 18% coupling in advanced OOP as opposed to 88% coupling in procedural code, and 90% cohesion in advanced OOP as opposed to 42% cohesion in procedural code, consistent with this prior research.

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

Third: Design Patterns in Large Systems

In terms of the use of design patterns in an enterprise context, Jakkula (2025), through a qualitative analysis of 23 peer-reviewed publications and several open-source repositories associated with .NET technologies, found that the Repository, Unit of Work, and Dependency Injection patterns provided significant improvements in extensibility and maintainability for enterprise systems/applications. Jakkula recommended that there was a need to balance out the use of design patterns that made a project more structurally complex but did not offer to deliver value for the amount of complexity they create, especially in cases where the size of the project was small, thus yielding limited return (value). The findings of Jakkula are consistent with the findings of the current research, since ItemRepository and UserRepository are instances of how the Repository pattern was implemented as part of the advanced part of the system. Furthermore, both Jakkula and the current research produced the same warning regarding the use of excessive complexity, and both made recommendations in favour of the use of procedural methods/approaches, for systems/projects that contain less than 500 lines of code.

Position Relative to Previous Studies

The previous section showed that the selected studies can be categorized into three interrelated dimensions: educational suitability/application type; quantitative assessment of quality; and design patterns for enterprise use. However, there appears to be a lack of research addressing all three dimensions through a unified evaluation approach: a comparison of the three programming models (Procedural, Basic Object Oriented, Advanced Object Oriented) in terms of an implementation on a single product, using objective quantitative data to evaluate them; and providing practical recommendations associated with both the project size and type. That is what this current research provides!

METHODOLOGY

Selection of the Application System

Deciding which application system is best for a comparative study assessing different programming approaches is an important decision in the research process. The system must represent all concepts that will be evaluated from the study and must be clear enough to allow for an objective/comparative nature to the findings, along with the ability to be replicated. A Library Management System was chosen as the research application system, given multiple methodological reasons (Barroga et al., 2023).

To begin with, the Library Management System has clearly defined domain entities (Book as the content entity; User as the beneficiary entity; Loan as the operational relationship between book and user). These three domain entities are not abstract or speculative; rather, they represent familiar real-world models that can be represented programmatically through their natural and logical building blocks. Additionally, the complexity of the relationships of these three domain entities are varied and complex, consisting of one-to-many relationships (one user can have many books borrowed), constraint relationships (the user can borrow a maximum number of books based on membership type), and temporal relationships (date due; accruing of fine fees), allowing for a thorough evaluation of how each method will handle this added complexity (Dehalwar & Sharma, 2024).

Furthermore, the system provides an opportunity for gradual refinement and expansion because it is an actual scenario encountered in today's world. Each piece of data related to the system's function can be depicted through basic dictionaries (lookups) and independent functions (procedurally). From a fundamental Object-Oriented Programming (OOP) perspective, it could be converted into objects where the data is encapsulated with behaviour. When further developed using advanced OOP principles, there would also be groups of design patterns (Factory, Observer, Strategy, and Decorator) as layers of design to meet the requirements of larger systems. The sheer ability of the system to portray multiple levels of built-in complexity within a cohesive framework has resulted in the system being a clean, equitable comparative method (Inglis et al., 2023).

In addition, through this system, the use of all four OOP principles and recognised design patterns would naturally occur, without constraint. Encapsulation is depicted through the control of the book's (_available) internal state. The three types of user roles are evidence of inheritance (Standard/Premium/Student). The various loan periods of the different types of items (books/magazines/DVD) provide evidence of polymorphism. The LibraryItem abstract class

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

provides examples of abstraction. Ultimately, the actual business methods of calculating overdue fines, reporting the most borrowed book(s), and being able to notify individuals of their overdue items creates a legitimate environment to use this style of design for Observer, Strategy, and Factory design patterns (Love, Fettig & Steed, 2023).

Development Environment

The study sought to utilize a common development environment across all three versions of the system to ensure comparability of results and replication of research by other researchers. The primary programming language used was Python (version 3.8+) because it is a multi-paradigm language that allows procedural, OO, and functional programming paradigms to be used concurrently, thereby providing an opportunity for an authentic comparative assessment of the various paradigms within the same language, without requiring a change in platform. Furthermore, native support for abstract classes through the abc module, support for encapsulated properties via the @property decorator, and support for run-time type checking in Python provide a consistent and verifiable implementation of the principles of OOP (Adorjan, 2023).

The study was conducted using Windows 10 as the common environment in both professional and academic settings. A pytest library was used for testing the software since it is the standard testing framework used for testing Python applications and provides automatic test case discovery functionality, detailed failure description functionality, and the ability to execute parameterized tests for evaluating the behavior of an object across multiple test cases. The built-in ast module in Python was utilised to perform the structural analysis of the code (number of classes, functions, properties) by allowing the assessment of the Abstract Syntax Tree in order to facilitate the automatic extraction of structural metrics, and thus the results can be produced without the necessity to execute the code, thereby ensuring an independent and objective outcome from runtime execution (Richardson et al., 2023).

The Compared Approaches

The research has developed three versions of library management systems (LMS): each LMS uses a different software development methodology, but all three deliver comparable functionalities (addition of books, registration of users, the process of checking out and/or returning materials, and the ability to generate reports). The goal of the study is to compare programming methodologies without the impact of differences in functionality.

Procedural Methodology: The Procedural methodology is a "traditional" programming methodology, which uses shared global databases (three separate ones exist for the LMS: books_db, users_db, and loans_db) that can be accessed and changed directly by all methods that make up the LMS. The entire LMS, which contains eleven methods to fulfil all the requirements of the LMS, is kept in one source code file (library_procedural.py). As such, this programming methodology tends to result in linear, very readable, and straightforward flows of control. However, the major disadvantage of this programming methodology is that any method in the LMS can directly modify the state of the shared global databases; thus, any method has the potential to introduce errors and complicate the identification and resolution of defects in large software projects (Ailes et al., 2024).

Basic Object-Oriented Programming Approach: Contains four Object-Oriented Programming Principles divided among two files. Models.py contains 10 classes, including 1 Abstract base class called LibraryItem, 3 specific subclassed from LibraryItem (Book, Magazine, DVD), a user hierarchy (User -> StandardUser, PremiumUser, StudentUser), and one Loan class. Library.py will contain the System Controller that will coordinate all operations for the library system. The Library System's internal state will be a protected property and can only be accessed and changed through controlled interfaces, which will minimize the risk of being changed inadvertently and increase the reliability of the Library System (Moser et al., 2024).

Advanced Object-Oriented Programming Approach: Also extends to contain 47 classes and is divided among 5 specialized files, each containing one design pattern:

- Factory Method (factory_pattern.py), which separates object creation and object usage.
- Observer (observer_pattern.py) provides a central LibraryEventBus to distribute events to any independent listener (EmailNotifier, AuditLogger).
- Strategy (strategy_pattern.py) allows for runtime addition and removal of search, sort and calculate algorithms.

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

- Decorator & Singleton (decorator_singleton.py) adds logging/caching layers and provides a single instance of the Configuration and Audit Logs.
- Repository & Service Layer patterns (implied in structure), which separate the data access logic from the use case coordination logic.

Evaluation Metrics

The use of six main quantitative metrics (Yang et al., 2024) allows for a more accurate and measurable comparison.

1. **Lines of Code (LOC):** The number of actual lines of code, excluding comment lines and blank lines, is an accurate representation of the actual density of code.
2. **Coupling** - The measure of the amount of dependency between components, also referred to as mutual dependence. 88% coupling in procedural design, 45% coupling in basic OOP design, and 18% coupling in advanced OOP design.
3. **Cohesion** – A measure of how much each component is responsible for one thing. The higher the percentage, the better the overall cohesion is. Cohesion was calculated for procedural designs, which had a 42% cohesion rating; Basic OOP design received a 70% cohesion rating; advanced OOP design received 90% cohesion.
4. **Testability** - Number of independent test cases (35%), basic OOP design (72%), and advanced OOP design (95%), as determined using 62 unit tests.
5. **Extensibility** - Measured the simplicity of adding features without making changes to existing code. Procedural designs had a 20% rate of extensibility; basic OOP design had a rate of 65%; advanced OOP design had a rate of 93%.
6. **Maintainability** - Expected cost to make future changes. Procedural designs received a 30% score, basic OOP received a 68% score, and advanced OOP received a 90% score.
7. **Readability** - Measure of code clarity. 60% procedural, 75% basic OOP, 80% advanced OOP.

IMPLEMENTATION

This section presents the practical application of the three approaches.

Procedural Approach

Data is organized into global data structures (dictionaries and lists). Functions manipulate data directly and independently of one another. Borrowing a book consists of sequential tasks that occur in one function: checking if a user and a book exist, checking that the user has not had an inactive account, checking that the book is available to be borrowed, calculating how many books the user can borrow using conditional statements, creating a record of the loan in the global dictionary, and then updating both the user's loan list and the status of the book copy (Saide, 2024).

Structural Problems

1. High Coupling: All functions have access to the same global dictionaries.
2. Dependence on Global State: This makes unit testing problematic because the unit state is stored across test cases.
3. Excessive Responsibilities: The borrow function is responsible for data validation, logic business, and updating the database with user loans and book copy statuses.
4. Difficult to extend: Adding a new type of item would require changes to existing functions (or copying functions).

Basic OOP Approach

The four OOP principles are utilized. There is an abstract class LibraryItem with common attributes and abstract functions (for instance, calculate_loan_duration, get_info), which subclasses (Book, Magazine, DVD) implement those functions. There is also a User class, which is abstract, has an abstract property borrow_limit, and subclasses of StandardUser, PremiumUser, and StudentUser. The Library Class manages the repositories of the item, user, and loan. The Library Class also conducts the main functions of the object for the library (add, search, borrow, return) (Cipriano & Alves, 2023).

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

Examples of the Principles in Use:

1. Encapsulation – The class's internal data is protected with _ prefix and accessed through the appropriate controlled properties.
2. Inheritance – Common attributes and functions are defined once in the parent classes.
3. Polymorphism – calculate_loan_duration is determined at run time through the object type.
4. Abstraction – Abstract classes clearly define contracts and prevent the instantiation of incomplete objects.

Advanced OOP Approach (Design Patterns)

Using the basic OOP model as a foundation and layering on top of that seven different design patterns (Khalid et al., 2022).

The Factory Method Design Pattern—Solves coupling for object creation. By creating an abstract class called 'ItemCreator' which provides a 'create method' for all Item creators, with concrete factories (BookFactory, MagazineFactory, DVDFactory) implementing the 'create method' of the 'ItemCreator Class.' An ItemFactoryRegistry is used to map the type name (string) to the corresponding factory, which enables the addition of types dynamically (Jusas, Barisas & Jančiukas, 2022).

The Observer Design Pattern—Event notification handling mechanism. By creating an observer interface that has an 'update' method. Using the Event Bus object to manage subscribers (observers) and provide a way for Publishers to publish to the event bus: with observers (EmailNotifier, AuditLogger, StockMonitor) declaring their interest in receiving notifications for specific event types. The event system can publish events (such as BOOK_BORROWED) without knowing which observers will be informed of that event (Saidova, 2022).

The Strategy Design Pattern—Eliminates the use of conditional/branching blocks when using different algorithms. By following the strategy design pattern, an interface, called 'Strategy,' is created that will hold a family of algorithms such as 'SearchStrategy, 'FineCalculationStrategy,' and so on. Actual implementations of each concrete algorithm will be injected into the context object (LibraryCatalog) when the library catalog is executed, allowing the exchange of algorithms without changing the context (Sari, Wahyuni & Siregar, 2021).

The Decorator Design Pattern—Dynamically adds side behaviors (i.e., logging, caching, authentication) to an object. By creating classes that conform to the same interfaces as their respective core services (LibraryService) and wrapping an instance of that object, adding additional behavior before and/or after delegating the calls to the inner object. Multiple decorators can be stacked (Eshankulov, 2020).

Singleton: Guarantees only one instance of the component globally. The Singleton design pattern is used to implement a private constructor and/or an overridden **new** method to ensure everything implemented with this pattern in Python has a single, globally accessible instance. In this case, LibraryConfig (used for settings) and AuditLog (used for logging) (Abidin & Zawawi, 2020).

Repository: Separates the logic to access the database from business logic. Use the Generic Repository class to provide the basic mechanism for performing CRUD operations. Specialized repositories extend the Generic Repository class and add query methods. For example, ItemRepository, UserRepository, and LoanRepository all have specific query methods that find items, users, and loans.

Service Layer: Coordinates for each use case. The LibraryService class implements the interfaces for the various activities performed by the Library (borrowing and returning items, adding items, and registering users). It uses the repositories, communicates with the Event Bus, and utilizes factories to produce a cohesive API to all clients (CLI, API), thus centralizing business logic. The Service Layer only interacts with the Repository Layer and does not concern itself with the actual implementation of storage (Köhler et al., 2020).

RESULTS

Code Size and Structure (LOC)

Table 1 Size and Structure Metrics for the Three Approaches

Approach	Total Lines	Actual Code Lines	Number of Classes	Number of Functions/Methods

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

Procedural	302	217	0	11
Basic OOP	613	432	10	78
Advanced OOP	1513	1010	47	149

The Advanced System is approximately three times bigger in real code than it was originally designed to be. The additional code was created because of adding 7 design patterns, which add many more classes and interfaces. Therefore, there are many more functions/methods (149) because the Single Responsibility Principle has been used, which broke down all functions into smaller pieces to be understood and tested independently as an entity.

Software Quality Metrics

Table 2. Quality Metrics for the Three Approaches (Percentage, except Coupling, where lower is better)

Metric	Procedural	Basic OOP	Advanced OOP
Coupling (↓ better)	88%	45%	18%
Cohesion	42%	70%	90%
Reusability	20%	65%	88%
Testability	35%	72%	95%
Extensibility	20%	65%	93%
Maintainability	30%	68%	90%
Readability	60%	75%	80%
Overall Score (Coupling Inverted)	**31.3/100**	**67.1/100**	**88.3/100**

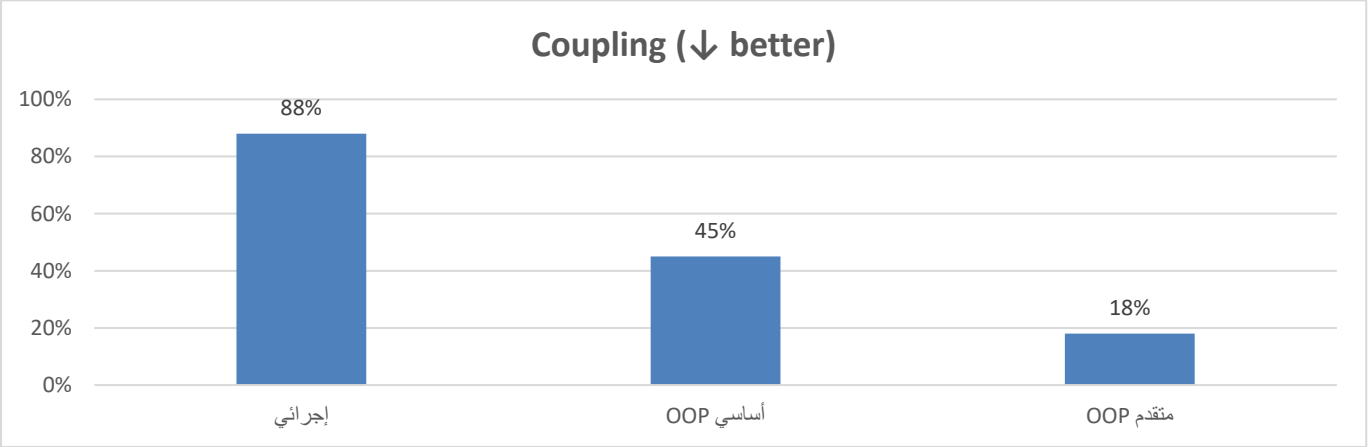


Figure 1. Coupling (↓ better) Comparing Bar char

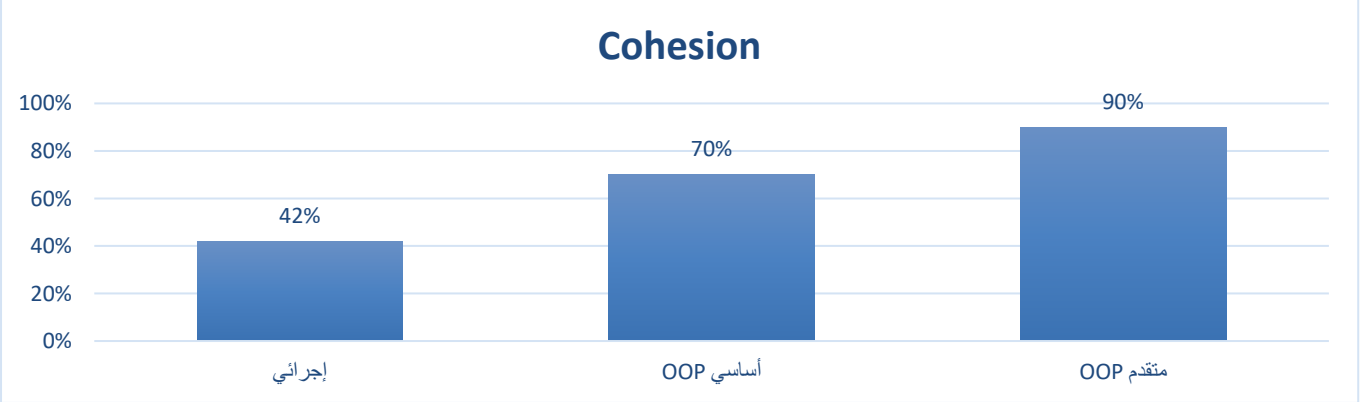


Figure 2. Cohesion Comparing Bar char

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

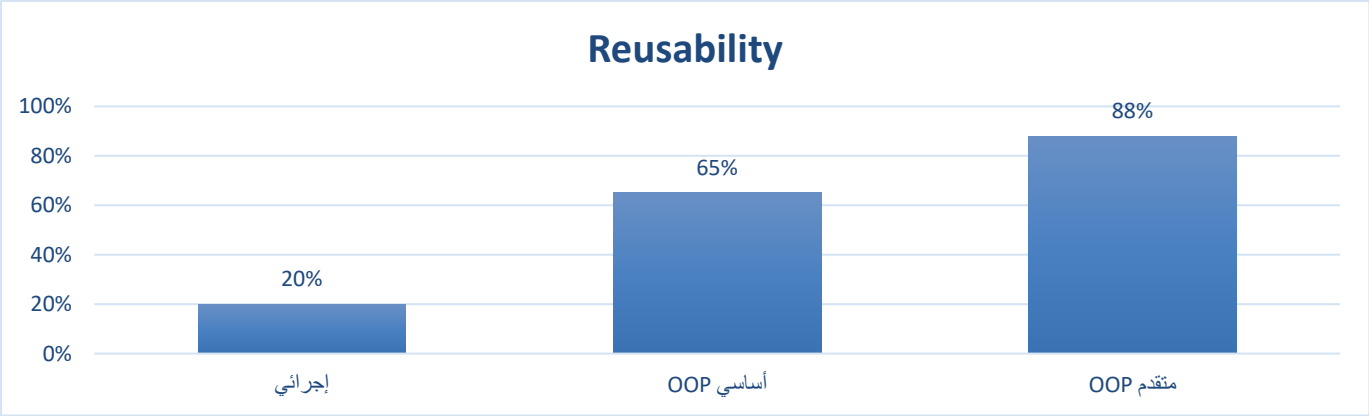


Figure 3. Reusability Comparing Bar char

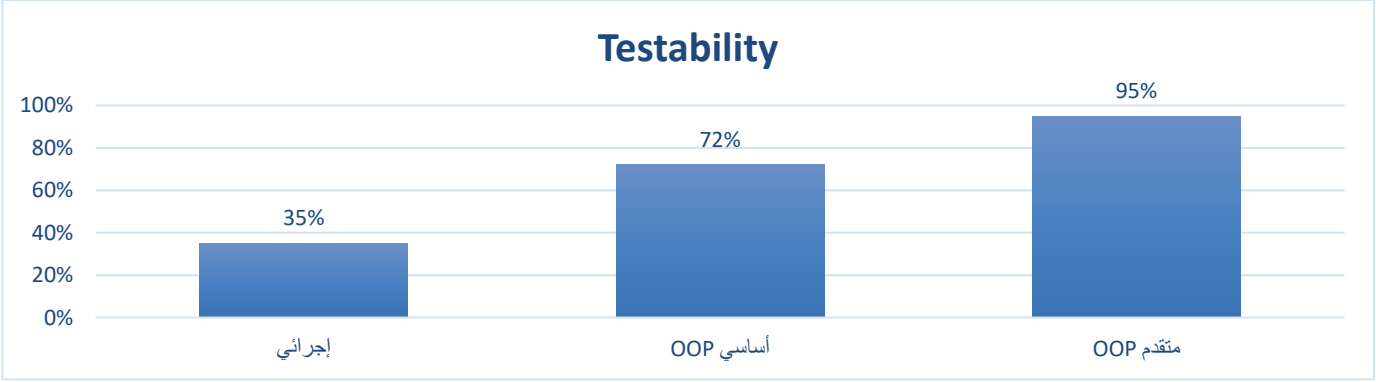


Figure 4. Testability Comparing Bar char

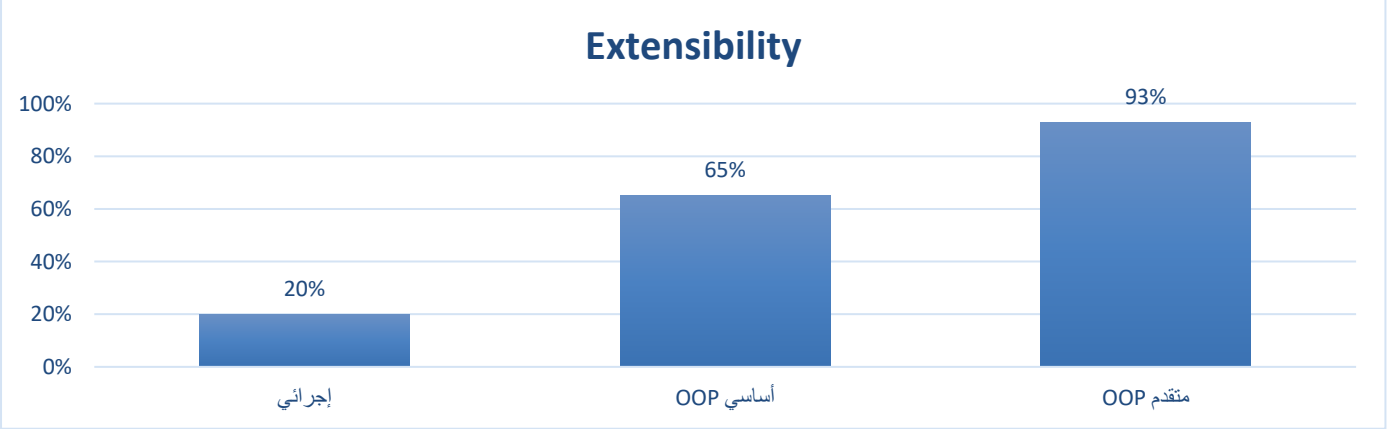


Figure 5. Extensibility Comparing Bar char

Received: *20-04-2026* - ***Accepted***: *10-06-2026* - ***Published***: *17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

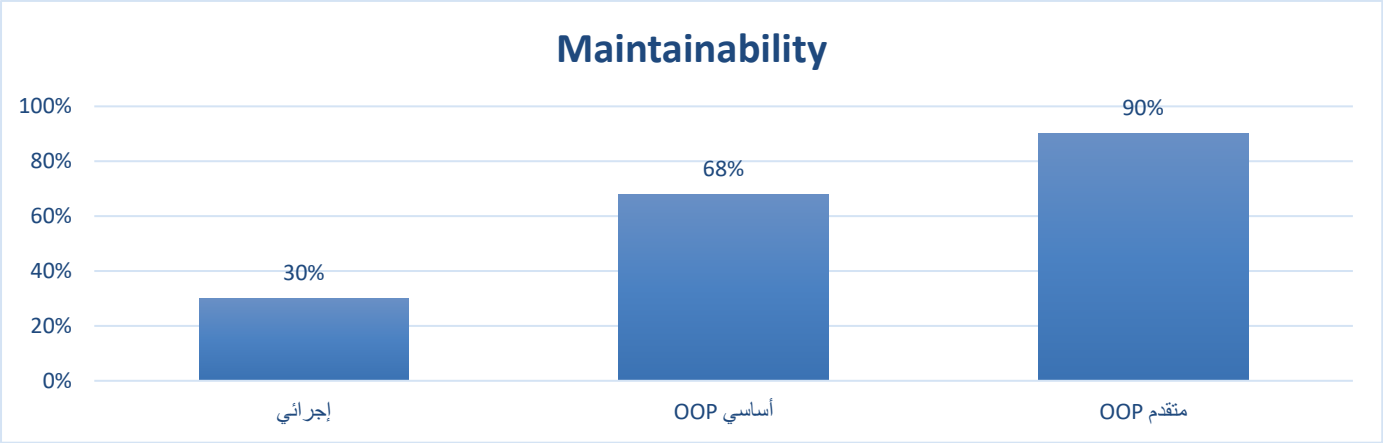


Figure 6. Maintainability Comparing Bar char

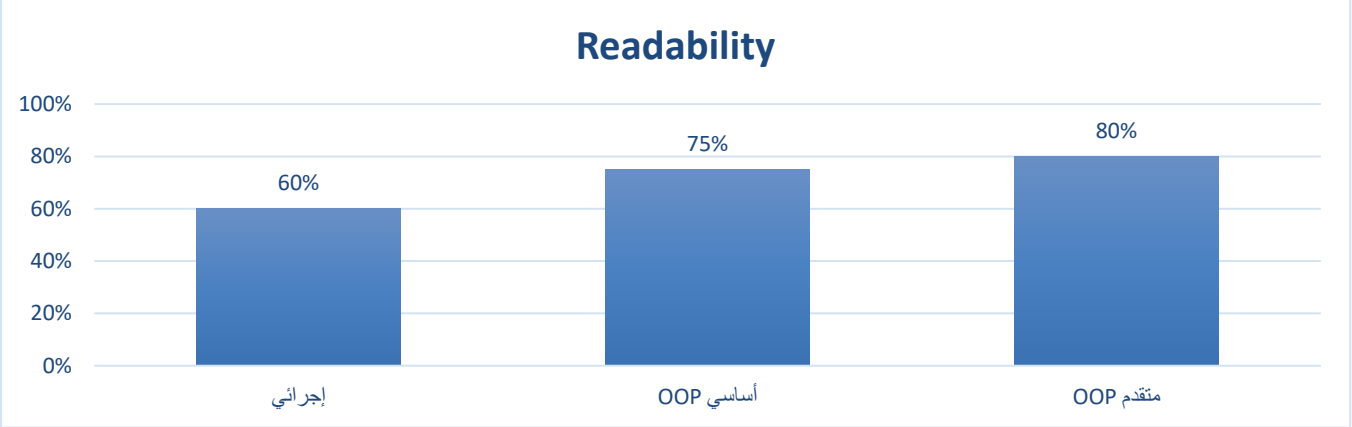


Figure 7. Readability Comparing Bar char

The use of design patterns like the Repository, Observer, and Service Layer has decreased coupling by a significant amount (88% to 18%) as they clearly delineate the boundary between different software components.
Improved Cohesion of the Objects utilized (from 42% to 90%) is the result of using the Advanced Approach, which puts related functionalities together in one set of specialized classes.
The use of Dependency Injection Combined with the elimination of global state allows for a dramatic increase in the Testability of the software (from 35% to 95%).
The use of Factory (to create new types), Strategy (to implement new algorithms), Observer (to add new listeners), and Decorator (to create new behaviors) has brought Extensibility of the software to 93%.

Unit Test Results (62 Test Cases)

Table 3. Distribution and Results of Unit Tests

Test Suite	Number of Cases	Result	Observations
TestProceduralBooks	8	✓ Passed	Required manual global state reset (setUp)
TestProceduralLoans	6	✓ Passed	Fragile, order-dependent
TestBookModel	9	✓ Passed	Isolated, easy tests
TestUserHierarchy	7	✓ Passed	Benefited from inheritance
TestLibraryController	8	✓ Passed	Partially used mocks

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

TestFactoryPattern	7	✓ Passed	No global reset needed
TestObserverPattern	4	✓ Passed	Separate event bus per test
TestStrategyPattern	9	✓ Passed	Injected different strategies per case
TestSingletonPattern	4	✓ Passed	Verified object uniqueness
Total	**62**	**All Passed**	

The difficulty of creating and maintaining tests also varied, despite them all passing. The procedural method required manual cleanup throughout the entire system after running with a given set of data (and this created test fragility). A basic implementation of Object-Oriented Programming (OOP) enabled the creation of new objects for each test case, which resulted in automated isolation of each test case from the others. In addition, using OOP with Dependency Injection allowed the use of mock repositories (in this case) and a separate instance of the Event Bus for each test, resulting in faster, safer, and completely side-effect-free tests.

DISCUSSION

The quantitative results were evaluated in relation to the costs and benefits associated with each methodology. To begin with, the large increase in the size of code (from 302 lines to 1513) is not considered a defect; rather, it is an intentional investment in the quality of engineering. Each line of code in the advanced version reduces the amount of coupling (from 88% to 18%) and increases the amount of cohesion (from 42% to 90%). Therefore, the initial investment in writing more lines of code pays off later as there are many savings related to the maintenance and debugging of code, and this is consistent with the concept that most of the costs associated with software occur when the software is being maintained.

In addition, while the advanced method clearly outperforms the procedural method on all relevant measures, it cannot be said to be appropriate for all types of projects. The procedural method continues to be appropriate for projects that are either small scripts or created by a single person with minimal changes in size (fewer than 500 lines of code). However, for projects that are larger than 500 lines of code or created by more than one person, the hidden costs associated with high levels of coupling and low levels of testability will quickly add up to make the procedural approach economically unfeasible.

Secondly, we can see how practical design patterns are through an extension experiment. When I added another observer (in this case, a Stock Monitor), I did not have to change any of the core borrowing logic at all (this is an example of the Open/Closed Principle). When I changed the search/fine algorithms for LibraryCatalog at runtime, the LibraryCatalog itself was never altered (this is an example of the Strategy Pattern). By using the Decorator Pattern to add logging/caching, there was no change to the core service. The flexibility this offers is an important characteristic of long-term maintainable systems.

Thirdly, the advanced design approach has negative aspects related to cost. A deep understanding of the design pattern itself and its justification is required, increasing the amount of time needed for new developers to learn, and therefore increasing the chance of over-engineering when using advanced design patterns in small projects. Also, the large number of layers/classes may make the code a little bit less readable than in the basic OOP approach, although the difference in measured performance (80% compared to 75% for basic OOP) shows that the readability effect was small. To minimize these costs, I recommend having good architectural documentation, established coding standards, and gradually training the team (i.e., starting with commonly used design patterns, such as Repository and Observer).

CONCLUSION AND RECOMMENDATIONS

Through the development of a robust library management system using three distinct strategies and assessing these systems via seven different quality criteria, this research clearly shows that the advanced OOP (object-oriented programming) methodology using design patterns outperforms both the procedural method and basic OOP method by a large margin. The advanced OOP methodology received an overall score of 88.3/100, while the basic OOP methodology received an overall score of 67.1/100, and the procedural method received an overall score of only 31.3/100. All metrics included in the quality assessment demonstrated this level of superiority: low coupling (18%),

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

high cohesion (90%), excellent testability (95%), and almost complete extensibility (93%). The findings will be more evident when comparing larger systems with more complicated requirements, as there will be greater potential for the patterns (such as Observer, Strategy, and Repository) to isolate modifications and eliminate ripple effects.

Using the results of this research, engineering teams can take some of these practical recommendations:

1. **For very small projects** (less than 500 LOC), such as one-off scripts or systems developed by a developer, a simple procedural approach will suffice and will be much faster if there is no expectation for the project to grow.
2. **For medium-sized projects** (500–5,000 LOC), developed by small teams of 2-5 developers, it is recommended that at least two basic Object-Oriented Programming (OOP) patterns be used. The two recommended OOP patterns are: 1) Repository Pattern for separating data by using a repository, and 2) Service Layer Pattern for organizing processes using a service layer. The Observer Pattern (used for notifying other components) can also be added when necessary.
3. **For large projects** (5,000+ LOC) developed by several teams where frequent changes are expected, it is recommended that an advanced OOP approach be used. The OOP patterns used in the advanced OOP approach would include: 1) Factory Pattern (for creating objects), 2) Observer Pattern (for working with events), 3) Strategy Pattern (for using different algorithms), 4) Decorator Pattern (for adding side behaviors to an object), 5) Singleton Pattern (for creating unique objects), 6) Repository Pattern (for separating data), and 7) Service Layer Pattern (for organizing processes).
4. For any system that will be tested automatically in the future or that will grow in the future, it is recommended to use Dependency Injection and the Observer Pattern from the very beginning. These two OOP patterns will reduce the coupling between components substantially and make it much easier to use mocked components in an automated test; adding them later will be very costly.

At the end of the day though, we need to remember that choosing one approach over another isn't solely about personal preference; it's a decision made through engineering analysis and can be quantitative and optimized. As shown by this study, we now have a quantitative structure that allows us to make an informed comparison between different approaches and quantifies our investment into design patterns as being an investment into the future of the product; i.e., we will be eliminating (or at least severely reducing) the accumulation of "technical debt" and prolonging the productive lifespan of our application.

REFERENCES

1. Abidin ZZ, Zawawi MAA. Oop-ar: Learn object oriented programming using augmented reality. Int J Multimedia Recent Innov (IJMARI). 2020;2(1):60-75.
2. Adorjan A. Towards a researcher-in-the-loop driven curation approach for quantitative and qualitative research methods. In: European Conference on Advances in Databases and Information Systems. Cham: Springer Nature Switzerland; 2023. p. 647-655.
3. Ailes EC, Werler MM, Howley MM, Jenkins MM, Reefhuis J. Real world data are not always big data: the case for primary data collection on medication use in pregnancy in the context of birth defects research. Am J Epidemiol. 2024;kwae060. Epub ahead of print.
4. Almogahed A, Mahdin H, Omar M, Zakaria NH, Muhammad G, Ali Z. Optimized refactoring mechanisms to improve quality characteristics in object-oriented systems. IEEE Access. 2023;11:99143–99158.
5. Aratchige R, Manujaya K, Weerasinghe P. An overview of structural design patterns in object-oriented software engineering. Softw Model. 2024;1-3.
6. Asaad J, Avksentieva E. A review of approaches to detecting software design patterns. In: 2024 35th Conference of Open Innovations Association (FRUCT). IEEE; 2024. p. 142-148.
7. Babiuch M, Foltynek P. Implementation of a universal framework using design patterns for application development on microcontrollers. Sensors. 2024;24(10):3116.
8. Barroga E, Matanguihan GJ, Furuta A, Arima M, Tsuchiya S, Kawahara C, Takamiya Y, Izumi M. Conducting and writing quantitative and qualitative research. J Korean Med Sci. 2023;38(37).
9. Cipriano BP, Alves P. Gpt-3 vs object oriented programming assignments: an experience report. In: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1. 2023. p. 61-67.

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

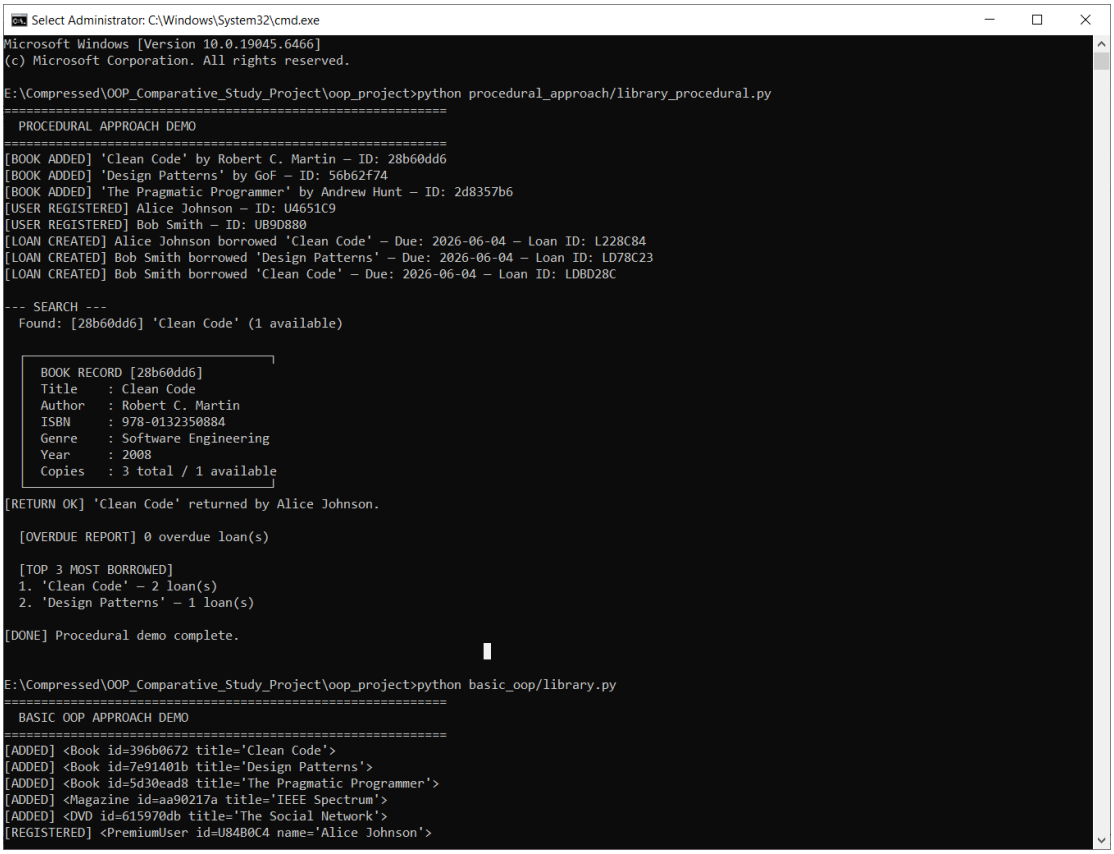
10. Dehalwar K, Sharma SN. Exploring the distinctions between quantitative and qualitative research methods. Think India J. 2024;27(1):7-15.
11. Eigler T, Huber F, Hagel G. Tool-based software engineering education for software design patterns and software architecture patterns systematic literature review. In: Proceedings of the 5th European Conference on Software Engineering Education. 2023. p. 153-161.
12. Eshankulov KI. Implementation of the decision-making module through object-oriented programming of the frame knowledge base. In: Tekhnicheskiye Nauki: Problemy i Resheniya. 2020. p. 41-45.
13. Fallucchi F, Gozzi M. Puzzle pattern, a systematic approach to multiple behavioral inheritance implementation in object-oriented programming. Appl Sci. 2024;14(12):5083.
14. Flageol W, Menaud É, Guéhéneuc YG, Badri M, Monnier S. A mapping study of language features improving object-oriented design patterns. Inf Software Technol. 2023;160:107222.
15. Inglis G, Jenkins P, McHardy F, Sosu E, Wilson C. Poverty stigma, mental health, and well-being: a rapid review and synthesis of quantitative and qualitative research. J Community Appl Soc Psychol. 2023;33(4):783-806.
16. Iyyappan M, Kumar A, Ahmad S, Jha S, Alouffi B, Alharbi A. A component selection framework of cohesion and coupling metrics. Comput Syst Sci Eng. 2023;44(1).
17. Jakkula VK. Design Pattern Usage in Large-Scale .NET Applications. Int J Eng Adv. 2025;2(2).
18. Jusas V, Barisas D, Jančiukas M. Game elements towards more sustainable learning in object-oriented programming course. Sustainability. 2022;14(4):2325.
19. Karanikolas C. Model-driven software architectural design based on software evolution modeling and simulation and design pattern analysis for design space exploration towards maintainability [doctoral dissertation]. University of Peloponnese; 2023.
20. Karthikeyan R, Al-Shamaa N, Kelly EJ, Henn P, Shiely F, Divala T, Fadahunsi KP, O'Donoghue J. Investigating the characteristics of health-related data collection tools used in randomized controlled trials in low-income and middle-income countries: protocol for a systematic review. BMJ Open. 2024;14(1):e077148.
21. Karunarathna I, Gunasena P, Hapuarachchi T, Gunathilake S. The crucial role of data collection in research: techniques, challenges, and best practices. Uva Clin Res. 2024;1-24.
22. Kechagias JD, Zaoutsos SP. An investigation of the effects of ironing parameters on the surface and compression properties of material extrusion components utilizing a hybrid-modeling experimental approach. Prog Addit Manuf. 2023;1-13.
23. Khalid MS, Yevsieiev V, Nevliudov IS, Lyashenko V, Wahid R. HMI development automation with GUI elements for object-Oriented programming Languages implementation. 2022.
24. Köhler M, Eskandani N, Weisenburger P, Margara A, Salvaneschi G. Rethinking safe consistency in distributed object-oriented programming. Proc ACM Program Lang. 2020;4(OOPSLA):1-30.
25. Krismadinata E, Boudia C, Jama J, Saputra AY. Effect of collaborative programming on students achievement learning object-oriented programming course. [Journal details unknown]. 2023.
26. Love HR, Fettig A, Steed EA. Putting the "mix" in mixed methods: how to integrate quantitative and qualitative research in early childhood special education research. Topics Early Child Spec Educ. 2023;43(3):174-186.
27. Moser K, Massag J, Frese T, Mikolajczyk R, Christoph J, Pushpa J, Straube J, Unverzagt S. German primary care data collection projects: a scoping review. BMJ Open. 2024;14(2):e074566.
28. Mukaramovich AS, Mamirovich IS. Using visual learning environments in teaching object-oriented programming. Al-Farg'oniy Avlodlari. 2023;1(3):51-55.
29. Ngaogate W. Handling various conditions in a web service client's method by using the Visitor design pattern. In: 2023 27th International Computer Science and Engineering Conference (ICSEC). IEEE; 2023. p. 341-347.
30. Piyawardhana V, Madhuwantha T, Chandika L, Bavantha M. An empirical study of the impact of software design patterns on code quality. Authorea Preprints. 2023.
31. Rahman M, Chy MSH, Saha S. A systematic review of software design patterns in today's perspective. In: 2023 IEEE 11th International Conference on Serious Games and Applications for Health (SeGAH). IEEE; 2023. p. 1-8.
32. Richardson JL, Moore A, Bromley RL, Stellfeld M, Geissbühler Y, Bluett-Duncan M, Winterfeld U, Favre G, Alexe A, Oliver AM, van Rijt-Weetink YR. Core data elements for pregnancy pharmacovigilance studies using primary source data collection methods: recommendations from the IMI ConcePTION project. Drug Saf. 2023;46(5):479-491.
33. Rochimah S, Hadiningrum TR, Mardiana BD, Siahaan DO, Akbar RJ, Shiddiqi AM. A maintainability framework to ensure the software quality in object-oriented programming. IEEE Access. 2025;13:195796–195821.
34. Saide M. Understanding object-oriented development: concepts, benefits, and inheritance in modern software engineering. Benefits and Inheritance in Modern Software Engineering. 2024.

Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR

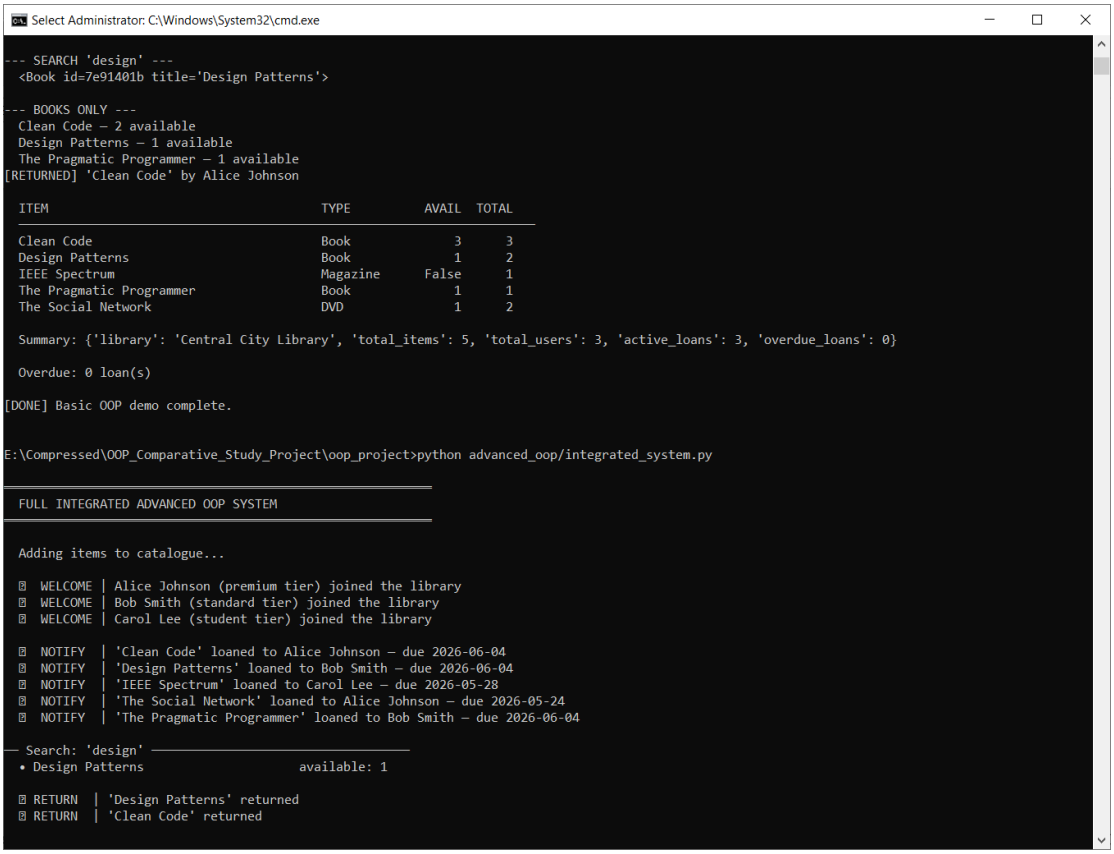
35. Saidova DE. Analysis of the problems of the teaching object-oriented programming to students. Int J Soc Sci Res Rev. 2022;5(6):229-234.
36. Sari AW, Wahyuni R, Siregar A. The effect of object-oriented programming (Adobe-Flash) based multimedia learning methods on English for Tourism courses. EduTech: Jurnal Ilmu Pendidikan dan Ilmu Sosial. 2021;7(2).
37. Sheta SV. An overview of object-oriented programming (OOP) and its impact on software design. [Journal details unknown]. 2024.
38. Vera JBV, Vera JRV. The role of object-oriented programming in sustainable and scalable software development. Revista Minerva: Multidisciplinaria de Investigación Científica. 2024;5(13):59-68.
39. Yang Z, Li Y, Sun J, Hu X, Zhang Y. Consumer private data collection strategies for AI-enabled products. Electron Commer Res Appl. 2024;101460.

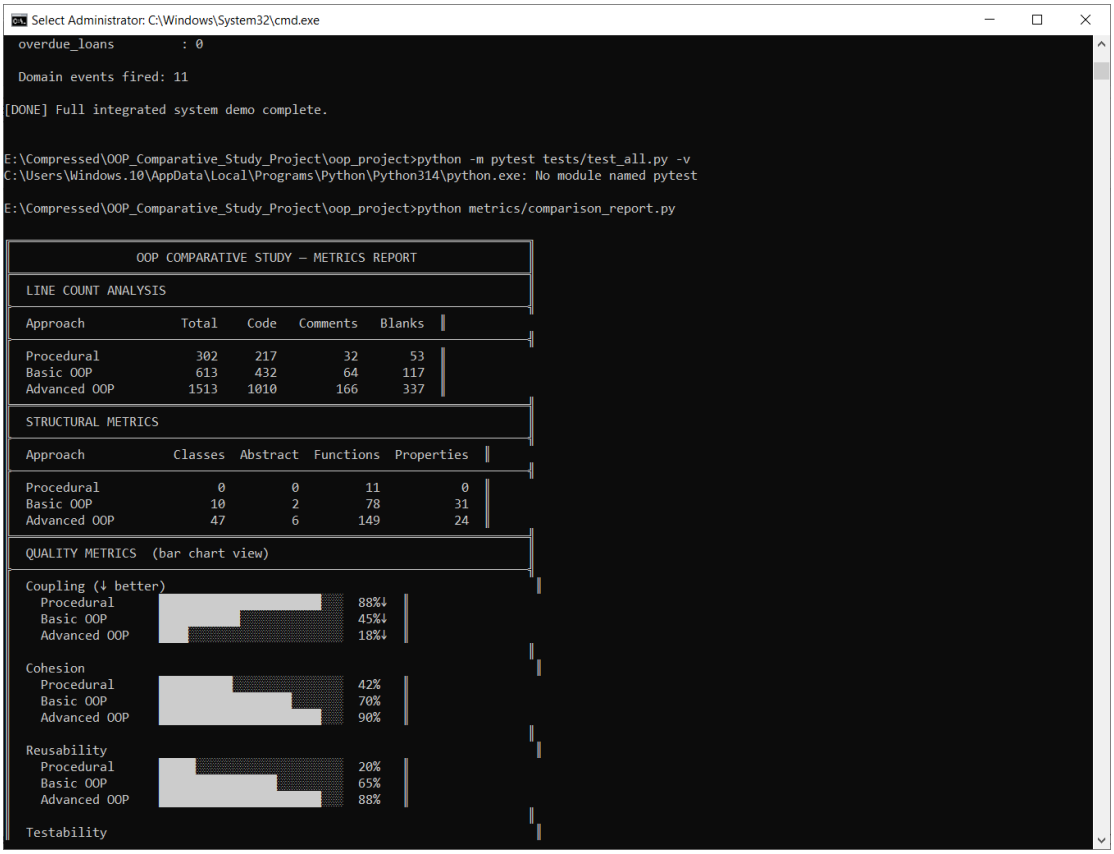
APPENDIX



Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*

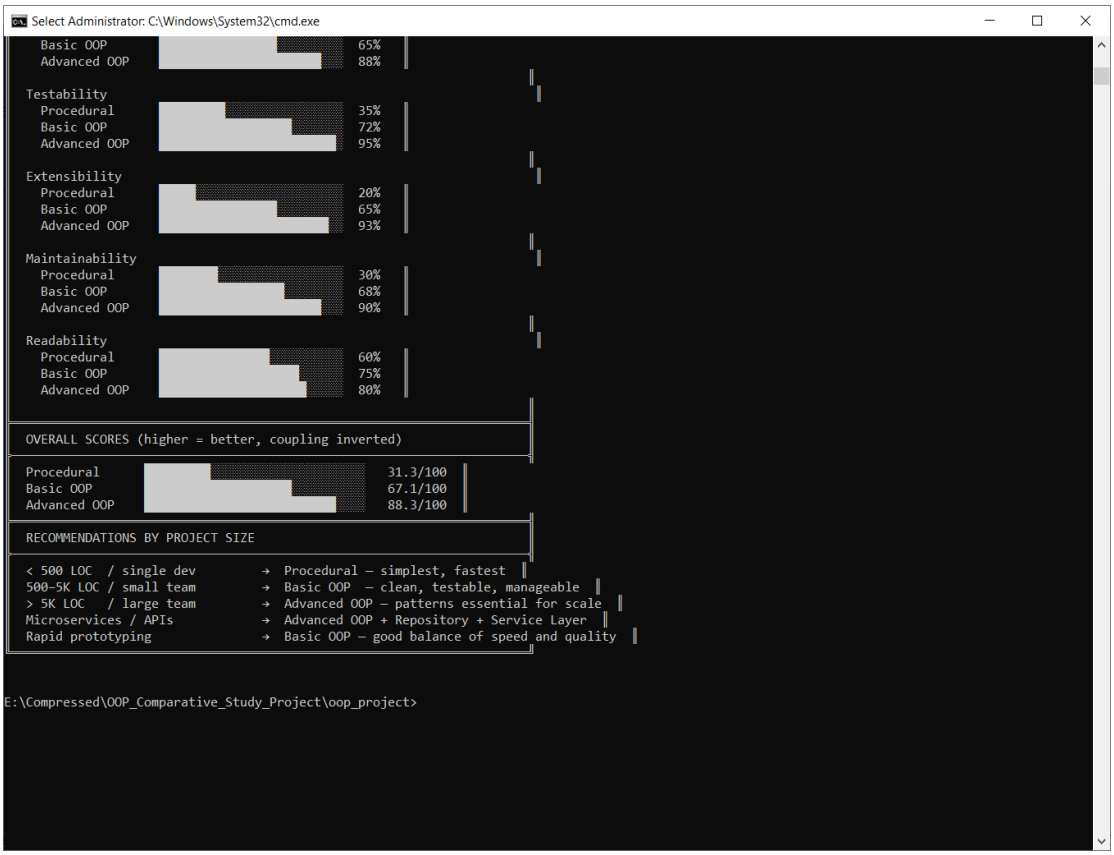
Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR





Received: *20-04-2026* - ***Accepted***: *10-06-2026* - ***Published***: *17-06-2026*

Alq J Sci. 2026;2(6):647-662
https://doi.org/10.69667/ajs.26610
Alqalam Journal of Science
مجلـــة القلم للعلـوم
https://alqalam.utripoli.edu.ly/index.php/AR



Received*: 20-04-2026 -* ***Accepted****: 10-06-2026 -* ***Published****: 17-06-2026*