WestLand Publishers

Digital Journal of Engineering Science and Technology

Case Study

Implementing Object-Oriented Programming Concepts in Low-Code Platforms A Case Study on Appian

Preeti Tupsakhare [1*]

[1]Engineer Lead - Medical Benefit Management Information Technology, Elevance Health.

***Corresponding Author:**

Preeti Tupsakhare, Engineer Lead - Medical Benefit Management Information Technology, Elevance Health, USA.

Received Date: **26 October, 2024**; **Published Date:** **19 November, 2024**

Keywords:

Low-Code Platform; Object Oriented Programming Concepts; OOPS; Appian.

Citation:

Preeti T (2024) Implementing Object-Oriented Programming Concepts in Low-Code Platforms A Case Study on Appian Digit J Eng Sci Technol 1(1): 101.

Abstract

In the evolving landscape of software development, low-code platforms like Appian provide rapid development environments that democratize application creation, allowing non-technical users to build solutions with minimal coding expertise. While low-code platforms traditionally lack native support for formal Object-Oriented Programming (OOP) paradigms, OOP principles such as encapsulation, inheritance, polymorphism, and modularity can still be applied to enhance the modularity, reusability, and scalability of applications. This paper explores the application of key OOP concepts in low-code environments and discusses their potential benefits, especially in the context of long-term application maintenance, team collaboration, and enterprise scalability.

Introduction

The rapid advancement in software development has led to the rise of low-code platforms like Appian, which empower users with minimal technical expertise to build robust applications. These platforms provide visual development environments and pre-built components, accelerating the software creation process. As businesses increasingly adopt low-code platforms to streamline development processes, the democratization of application development has emerged as a significant advantage, enabling non-technical users to contribute to system design and implementation. However, while low-code platforms offer notable time-saving benefits, they often lack support for traditional software engineering principles, particularly Object-Oriented Programming (OOP). OOP, which is based on encapsulation, inheritance, polymorphism, and modularity, has long been recognized for enhancing software modularity, maintainability, and scalability. In environments where application development may scale significantly or involve large teams, the absence of formal OOP structures can present challenges to long-term maintenance, extensibility, and collaborative development [4]. Despite these limitations, it is possible to apply OOP principles within low-code environments to address some of these challenges. This paper explores how key OOP concepts can be adapted and implemented in low-code platforms to enhance application design. The discussion will focus on the potential benefits of incorporating these principles to improve long-term application maintenance, facilitate team collaboration, and support enterprise-level scalability [4,5].

Object-Oriented Programming Concepts Overview

Encapsulation: Encapsulation is the principle of bundling data (attributes) and the methods (functions) that operate on that data into a single unit or class. This concept restricts direct access to some of an object's components, which enhances data security by preventing unauthorized or unintended modifications. By hiding the internal state of an object and exposing only the necessary functionalities through public methods, encapsulation promotes a clean separation of concerns. This separation is crucial

for building modular applications where individual components can be maintained or updated independently, ensuring that changes to one part of the system have minimal impact on other parts. In the context of low-code environments, encapsulation can be mimicked by creating self-contained reusable components or services that manage their own state and behaviors, which can be invoked or modified without exposing the underlying complexities [4].

Inheritance: Inheritance is an OOP concept where a class (child or subclass) inherits properties and behaviors (methods) from another class (parent or superclass). This allows for code reuse, making it easier to build scalable and extensible systems by defining common characteristics once in a parent class and then reusing or overriding those characteristics in child classes. Inheritance also supports the creation of hierarchical class structures, allowing for more organized and manageable code. In low-code platforms, while inheritance may not exist in a formal sense, similar concepts can be applied by creating reusable templates or components that can inherit and extend functionalities from a base template, promoting consistency across applications and reducing redundancy in code [4].

Polymorphism: Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. This capability enables flexibility in how different objects are used, allowing the same operation or method to behave differently depending on the object it is applied to. For example, a parent class Shape could have a method draw (), and each subclass (like Circle, Square, or Triangle) could implement this method differently. Polymorphism thus enables systems to be more dynamic and adaptable. In low-code environments, polymorphism can be reflected through configurable workflows or user interfaces that adapt based on user roles or application contexts, allowing different components to behave differently under varying circumstances while maintaining a unified interface [4].

Modularity: Modularity in software design refers to the practice of breaking down a system into smaller, self-contained components or modules that can be developed, tested, and maintained independently. OOP inherently supports modularity by allowing developers to define discrete classes and objects that perform specific functions and interact with one another through well-defined interfaces. This approach leads to more maintainable and scalable architectures, as individual modules can be easily modified, replaced, or extended without disrupting the entire system. In low-code environments, modularity is typically achieved by designing applications using building blocks like components, workflows, and services, each serving a specific purpose and reusable across different parts of the system [6].

Applying Oop Principals in Low-Code Platforms

Encapsulation

Encapsulation in Object-Oriented Programming (OOP) refers to the bundling of data and methods that manipulate that data into a single, cohesive unit, typically a class. In the context of Appian, a low-code platform, encapsulation is achieved through **Custom Data Types (CDTs)** and **Records**.

- **Custom Data Types (CDTs):** CDTs in Appian are akin to class attributes in OOP. They bundle related data fields together into a structured format, such as a "Person" CDT containing fields like name, address, and contact number. This grouping allows developers to organize and manage related data more effectively, ensuring that the attributes are treated as a single cohesive unit when passed through process models or interfaces. CDTs also facilitate data encapsulation by enabling the creation of reusable data structures that maintain integrity across different parts of the application [2].

- **Records:** Records in Appian combine data and behaviour, encapsulating both into a cohesive entity. They integrate data (often derived from CDTs or external systems) with interfaces, process models, and business logic, effectively mirroring how OOP classes encapsulate attributes and methods. For example, a "Customer Record" might bundle together a customer's data with processes related to order fulfilment, customer service interactions, and interface displays. This encapsulation ensures that users interact with a unified structure, while the underlying data and processes remain secure and maintain integrity [3]. Through this approach, Appian encapsulates data and behaviour together, allowing developers to package complex systems into manageable, reusable components while controlling data access and ensuring that data manipulation happens in controlled, structured ways.

Inheritance and Reusability

Traditional inheritance in OOP allows a subclass to inherit properties and behaviors from a parent class, promoting code reuse and extensibility. While Appian doesn't support direct inheritance in the same way, it provides powerful reusability features that mimic OOP principles of inheritance:

- **Interface Reuse:** Appian allows developers to create base interfaces, which act like parent classes in OOP. These interfaces can be extended and customized for specific use cases, reducing the need to recreate components from scratch. For example, a base interface for a user profile can be reused and extended to create specialized interfaces for different user roles, such as administrators or customers. This modular approach enhances maintainability and scalability by leveraging reusable components.

- **CDT Nesting:** In Appian, you can nest one CDT inside another, simulating "is-a" relationships commonly used in OOP inheritance. For instance, if you have a "Person" CDT and an "Employee" CDT, you can include the "Person" CDT as a field

within the "Employee" CDT, representing that an employee "is a" person while still adding employee-specific fields. This structure promotes the reuse of data definitions and simplifies system updates, as changes to the base CDT (like "Person") automatically propagate to the nested CDT (like "Employee").

- **Expression Rule Inheritance:** Appian's expression rules function as reusable logic blocks, akin to methods in OOP. Developers can write expression rules once and reuse them across multiple components, simulating inheritance by sharing common behaviors across different application areas. This improves consistency and reduces code duplication, ensuring that the same logic can be applied universally, and any changes to the expression rule will be reflected across the system.

Polymorphism

Polymorphism in OOP allows objects to be treated as instances of their parent class, enabling the same operation to behave differently depending on the specific object it is applied to. Appian supports polymorphism-like behaviour through **dynamic interfaces** and **decision-making mechanisms** that adapt based on conditions.

- **Dynamic Interfaces:** Appian interfaces can dynamically adapt based on input data or user roles. For example, an interface might display different fields or options depending on whether the user is an administrator or a customer. This allows the interface to behave differently while maintaining a unified structure, akin to method overriding in OOP, where different classes implement a shared method in different ways. Dynamic interfaces are crucial for building flexible, role-based user experiences in enterprise applications.

- **a!match() Function:** Appian's **a!match()** function operates similarly to OOP method overriding, allowing developers to implement different behaviours based on specific conditions. For example, based on the input data type or value, the system can decide which process or action to execute, mimicking how different objects in OOP might override a parent class method to achieve context-specific behaviour [2,3].

- **Decision Tables:** Decision tables in Appian provide another form of polymorphism by configuring different outcomes based on varying input conditions. Much like OOP polymorphism allows different objects to respond to the same method in different ways, decision tables can guide a process to take different actions based on the provided inputs. This ensures flexible, context-aware application behaviour.

Modular Design

Modular design is a cornerstone of OOP, and it is equally emphasized in Appian's architecture. Modular systems break down complex software solutions into smaller, self-contained units that can be developed, tested, and maintained independently. Appian promotes modular design through several mechanisms:

- **Reusable Interfaces:** Appian allows developers to create **interface fragments**, which are smaller reusable pieces of a larger interface. These fragments can be composed into bigger, more complex interfaces, ensuring that common design elements and logic can be reused across multiple pages or components. This modularity speeds up development and simplifies long-term maintenance, as changes to a fragment are reflected across all interfaces that use it [1].

- **Expression Rules: Expression rules** in Appian are modular logic units that can be reused across various parts of an application. These rules enable developers to define logic once and call it from multiple interfaces, workflows, or process models, promoting consistency and reducing duplication. For example, a validation rule for email formatting can be written once and reused anywhere an email address is required, ensuring uniform validation throughout the application [2].

- **Process Models:** In Appian, complex workflows can be broken down into smaller, reusable **subprocess models**. These subprocess models are like OOP functions, designed to handle specific tasks within a larger process. By reusing these subprocesses, developers can build scalable, maintainable workflows that can be easily modified or extended without disrupting the entire system. This modular approach ensures that workflows remain adaptable and efficient, even as application complexity grows. By applying these OOP concepts, you can create a more maintainable, scalable, and efficient Appian development ecosystem. This approach allows for faster development, consistent user experiences, and easier long-term maintenance of your applications [7].

Benefit of Implementing Oop Principals in Low-Code Platform

Enhanced Reusability: Utilizing Object-Oriented Programming (OOP)-like structures such as reusable interfaces and Complex Data Types (CDTs) can significantly enhance reusability in low-code platforms. By defining standard interfaces and reusable components, developers can avoid redundancy and ensure consistency across various parts of the application. This not only speeds up the development process but also minimizes the likelihood of errors, as the same tested and proven component is used multiple times [1].

Scalability: OOP-based designs are inherently modular, which allows low-code applications to scale more effectively. As businesses evolve and new features or modules are required, these modular components can be extended or reused without needing to rewrite existing code. This modularity supports the

application's ability to grow alongside the business, ensuring it can handle increased complexity and larger workloads efficiently [6].

Maintainability and Reduced Technical Debt: Encapsulation and modularity, core principles of OOP, help reduce the interdependency of components within an application. This makes it easier to maintain the system because changes in one module don't necessarily impact others. Developers can fix bugs or add new features to specific parts of the application without introducing issues elsewhere. This separation of concerns is crucial for reducing technical debt, as it allows for smoother updates and maintenance over time [6,7].

Improved Collaboration: Adopting OOP principles provides a common design pattern that multiple developers can follow, facilitating better collaboration on large projects. In a low-code platform, adhering to principles like encapsulation, component reuse, and modularity means that different team members can work on various parts of the application simultaneously without conflicts. This collaboration is further enhanced by clear, well-defined interfaces and components that make it easier to understand and extend the application [5].

Flexibility and Adaptability: OOP-like designs bring flexibility to low-code applications by allowing developers to build systems that can easily adapt to changing business requirements. Polymorphism, seen in dynamic interfaces and decision rules, enables the application to respond to different inputs and scenarios without needing extensive rewrites. This adaptability is critical in today's fast-paced business environment where requirements can change frequently.

Better Testing and Debugging: When application components are encapsulated and modularized, it becomes easier to isolate and test individual parts of the system. This leads to more efficient and effective debugging, as developers can focus on testing specific components independently. By isolating modules, developers can identify and resolve issues more quickly, enhancing overall application stability and reliability. This modular testing approach also supports continuous integration and delivery practices, ensuring that new changes are thoroughly vetted before deployment [5]. By incorporating these OOP principles within low-code environments, developers can achieve greater efficiency, scalability, and maintainability, while also fostering better collaboration and adaptability. These benefits collectively contribute to building robust, flexible, and high-performing applications that can grow and evolve with the business.

Best Practices for Reusable Components in Appian

Parameterization Design components with rule inputs to make them flexible and context independent. Parameterization is a critical strategy for creating reusable components in Appian. By designing components with rule inputs (parameters), you allow for flexibility and adaptability, as these components can be customized based on the context in which they are used. For instance, an interface component designed to display a form can accept different rule inputs for labels, fields, or validation logic, enabling it to be reused across various scenarios without needing modifications. This reduces duplication and enhances reusability while maintaining a single, centralized source of logic.

Best Practice: Ensure that components are built with a clear understanding of which parameters need to be adjustable and which should remain static. Define flexible rule inputs early on to accommodate varying contexts without tightly coupling components to specific use cases [2,3].

Data less Design: Create interfaces that are not tightly coupled to specific data objects, allowing for greater reusability. Data less design refers to the practice of separating the structure and logic of your interfaces from the specific data they manipulate. By designing interfaces that rely on rule inputs or external data sources (rather than hardcoding specific data objects), you can enhance their reusability. This approach allows the same interface to be used across multiple applications or scenarios by simply passing in the relevant data at runtime, rather than binding the interface to specific objects.

Best Practice: Use rule inputs or query rules to feed data into components rather than embedding static data objects. This keeps components flexible, allowing them to handle different types of data without requiring modifications to the underlying structure.

Design Library: Utilize Appian's Design Library to store and share reusable interface components across applications. Appian's Design Library is a centralized repository where reusable components—such as interface fragments, expression rules, and process models—can be stored and shared across applications. By leveraging this library, teams can access pre-built components, ensuring consistency and reducing development time. The Design Library also enables cross-team collaboration by making reusable assets available to all developers working on a project.

Best Practice: Regularly update the Design Library with new components and ensure that each component is clearly documented with usage instructions. Promote a culture of sharing and collaboration among development teams to maximize the value of reusable components.

Shared Components Application: Create a dedicated application to house reusable components, making them easily accessible across projects. A Shared Components Application is a dedicated Appian application that stores common, reusable components that can be accessed across various projects. This

approach enhances modularity by centralizing reusable assets in a single location, making them easier to maintain, version, and access. It also ensures that updates to these components are propagated to all dependent projects without requiring manual intervention.

Best Practice: Create a robust folder structure within the Shared Components Application to organize components logically by type (e.g., interfaces, expression rules, process models). Use consistent naming conventions and include clear documentation to make components easily discoverable and understandable.

Composition Over Inheritance: Use composition to build complex components from simpler ones, rather than relying on API inheritance. Appian doesn't support traditional inheritance, so composition is a more effective way to build complex components. Composition involves combining simpler, reusable components to create more complex functionality. For example, an interface may be composed of several smaller interface fragments, each responsible for rendering different aspects of the UI. This approach promotes modularity and reusability, as individual components can be modified or replaced without affecting the overall system.

Best Practice: When designing complex components, break them down into smaller, reusable building blocks that can be easily composed into larger structures. This enables greater flexibility and allows for easier maintenance and updates to individual components without impacting the entire system.

Clear Naming Conventions: Establish and follow naming conventions for reusable components to enhance discoverability and maintainability. Clear and consistent naming conventions are essential for managing reusable components in Appian. A well-defined naming convention ensures that components are easily identifiable and understandable by all team members. It also enhances maintainability by making it easier to locate specific components when updates or bug fixes are required [2].

Best Practice: Define a clear naming convention that includes descriptive names for components, indicating their purpose and usage. For example, use prefixes to denote the type of component (e.g., int_ for interfaces, exp_ for expression rules) and include information about the functionality (e.g., exp_CalculateTotalCost or int_UserProfileForm).

Documentation: Provide clear descriptions and usage instructions for reusable components in the Design Library. Documentation is crucial to ensure that reusable components can be easily understood and implemented by other developers. In Appian's Design Library, each component should be accompanied by clear descriptions that explain its purpose, how it works, and how to use it. This ensures that components are not only reusable but also easy to integrate into different projects without requiring significant knowledge transfer [3].

Best Practice: For each reusable component, provide documentation that includes details such as its parameters, expected input/output, any dependencies, and example usage scenarios. Keep documentation up to date to reflect any changes in the component's functionality.

Version Control: Implement proper change management processes for reusable components to prevent unintended impacts on dependent applications. As reusable components are often used across multiple projects, changes to a component can have wide-reaching effects. Implementing version control for reusable components helps manage updates and prevents unintended side effects. For example, changes to a shared interface should be versioned so that dependent applications can choose when (and if) to adopt the new version, ensuring stability and preventing breaking changes [6].

Best Practice: When updating a reusable component, create a new version rather than modifying the existing one. Document changes clearly and ensure that all affected applications are notified of the update. Consider using feature flags or toggles to gradually roll out new versions of components in production environments [7].

Conclusion

The implementation of Object-Oriented Programming (OOP) principles in low-code platforms like Appian provides a powerful approach to enhance the modularity, reusability, and scalability of applications. While low-code environments typically lack native support for traditional OOP features, this paper has shown how core concepts such as encapsulation, inheritance, polymorphism, and modularity can still be adapted to these platforms.

Encapsulation in Appian is effectively achieved through the use of Custom Data Types (CDTs) and Records, allowing developers to package data and behaviour into cohesive units that maintain data integrity. Although traditional inheritance is not available, Appian's emphasis on reusability through features like interface reuse, CDT nesting, and expression rule inheritance offers a flexible alternative to build scalable systems. Furthermore, polymorphism-like behaviour can be realized through dynamic interfaces and decision-making functions, allowing applications to adapt based on varying input conditions. Finally, Appian's architecture promotes modularity, enabling developers to create complex workflows from smaller, reusable components, ensuring that applications remain maintainable and adaptable over time [4,6,7]. By adopting OOP-inspired design practices, developers can take full advantage of low-code platforms' rapid development capabilities while still ensuring the long-term maintainability and scalability of their applications. As businesses increasingly turn to low-code platforms to streamline development processes,

these practices will be essential for building robust and efficient systems that can grow with evolving business needs. The future of low-code platforms lies in further refining these approaches, making it easier to incorporate the benefits of traditional software engineering into environments designed for rapid application development. As low-code technologies advance, the convergence of OOP principles and low-code methodologies will play a pivotal role in shaping the next generation of enterprise applications.

Conflicts of interest

None

Funding

None

References

1. Reuse Promotion, Appian Community, Accessed: Oct. 24, 2024.
2. Frequently Reused Appian Components, Appian Community, Accessed: Oct. 24, 2024.
3. Unlocking Reusability with Appian's Design Library, All About Appian, Jul. 31, 2023.
4. G. Schlageter e (1988) OOPS - an object-oriented programming system with integrated data management facility, in Proc. Fourth Int. Conf. Data Eng, Los Angeles, CA, USA, pp 118-125.
5. Lethbridge TC (2021) Low-Code Is Often High-Code, So We Must Design Low-Code Platforms to Enable Proper Software Engineering. In: Margaria T, Steffen B. (eds) Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2021. Lecture Notes in Computer Science 13036: 202-212.
6. Tisi M, Mottu JM, Kolovos DS, Lara JD, Guerra EM, et al. (2019) Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. HAL open science.
7. Rokis K, Kirikova M (2022) Challenges of Low-Code/No-Code Software Development: A Literature Review. BIR 2022 462: 3-17.