International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 4, Issue 2, 40-45, 2023
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V4I2P105

Original Article

Apex Design Patterns: Practical Insights for Developing Resilient and Scalable Applications

Laxman Vattam
Independent Researcher, Washington, USA.

Abstract *- Design patterns in Apex represent established best practices and reliable solutions for common design challenges in software development. These patterns provide a structured approach to organizing code, improving efficiency, maintainability, and scalability in applications. By implementing these principles, developers can streamline the development process while ensuring a solid framework for future enhancements and adaptations. This article examines the role of design patterns within the Salesforce ecosystem, offering insights into their purpose, significance, and practical implementation.*

Keywords *- Salesforce, Apex Design Pattern, Singleton, Strategy, Decorator, Facade, Bulk State Transition.*

1. Introduction

Efficiency and scalability are fundamental in Salesforce development, and design patterns serve as essential tools in achieving these objectives. These patterns offer well-established solutions to recurring challenges, enabling developers to build robust and well-structured Salesforce applications. In software development, design patterns are reusable solutions that address common architectural and coding challenges. They act as blueprints that guide developers in implementing best practices, leading to the creation of maintainable, efficient, and scalable applications. Within the Salesforce ecosystem, these patterns play a crucial role in optimizing the development process while improving the overall quality of applications. By leveraging established design patterns, Salesforce developers can address specific challenges without the need to devise new solutions from scratch. This approach not only accelerates development but also minimizes the likelihood of errors. Given that Salesforce applications continually evolve, maintaining a well-structured and easily comprehensible codebase is vital. Design patterns facilitate code maintainability by enhancing clarity and organization, making it easier for developers to understand, modify, and extend applications as requirements change. Additionally, Salesforce applications must often scale to accommodate expanding datasets, increasing user demands, and evolving business needs. Design patterns support scalability by ensuring that applications are architected to handle growth efficiently. Moreover, these patterns promote consistency in problem-solving approaches, fostering a structured and streamlined development process that enhances productivity and code reliability.

2. Categories of Design Patterns

Design patterns are generally classified into three primary categories, each addressing different aspects of software design:

2.1 Creational Patterns

Creational patterns focus on the process of object creation, ensuring that objects are instantiated in a controlled and efficient manner. These patterns promote flexibility and reusability by abstracting the specific instantiation details, allowing for more adaptable and maintainable code.

Some widely used creational patterns include:

- Singleton Pattern – Ensures that a class has only one instance and provides a global access point to it.
- Factory Pattern – Encapsulates object creation logic to create instances based on input parameters.
- Abstract Factory Pattern – Provides an interface for creating related objects without specifying their concrete classes.
- Builder Pattern – Simplifies complex object construction by breaking it into step-by-step procedures.
- Prototype Pattern – Creates new objects by copying existing ones rather than instantiating new instances directly.

2.2 Structural Patterns

Structural patterns focus on the composition of classes and objects to form larger, more efficient structures. They define relationships between components, enabling developers to extend functionality without modifying existing code, thereby promoting scalability and maintainability.

Common structural patterns include:

- **Adapter Pattern** – Bridges incompatibilities between different interfaces, allowing them to work together.
- **Decorator Pattern** – Dynamically enhances object functionality without altering its structure.
- **Composite Pattern** – Treats individual objects and groups of objects uniformly, enabling hierarchical structures.
- **Proxy Pattern** – Controls access to an object, improving security and performance.
- **Bridge Pattern** – Separates an abstraction from its implementation, increasing flexibility and reducing dependencies.

2.3 Behavioral Patterns

Behavioral patterns define how objects interact and manage communication between them to achieve desired functionality. These patterns help distribute responsibilities effectively, ensuring better organization and reducing tight coupling in a system.

Some key behavioral patterns include:

- **Observer Pattern** – Establishes a subscription-based mechanism where objects react to state changes in other objects.
- **Strategy Pattern** – Enables the selection of an algorithm at runtime by defining a family of interchangeable behaviors.
- **Command Pattern** – Encapsulates requests as objects, allowing users to parameterize and queue actions.
- **State Pattern** – Allows an object to alter its behavior when its internal state changes dynamically.
- **Chain of Responsibility Pattern** – Passes requests along a chain of handlers, where each handler processes or forwards the request.

These design patterns provide well-structured, reusable solutions to common development challenges, contributing to more efficient, scalable, and maintainable Salesforce applications.

2.4 Advantages of Utilizing Apex Design Patterns

Implementing design patterns in Apex development provides numerous benefits, enhancing the efficiency, maintainability, and scalability of Salesforce applications. These patterns establish a structured approach that optimizes development processes while ensuring long-term sustainability.

- **Encourages Best Practices:** Incorporating design patterns promotes adherence to coding best practices. This not only enhances code quality but also serves as an educational tool for new developers, helping them adopt effective programming techniques.
- **Promotes Code Reusability:** One of the key advantages of design patterns is their ability to facilitate the reuse of well-established solutions. Developers can apply the same pattern across multiple scenarios, resulting in quicker development cycles and a standardized approach to problem-solving.
- **Enhances Maintainability:** By introducing a systematic structure, design patterns make code easier to understand and manage. When modifications or updates are required, developers can efficiently locate and update specific components without disrupting the entire system.
- **Provides Flexibility:** Although design patterns introduce structure, they also allow for adaptability. Developers can modify and tailor these patterns to suit specific business needs, ensuring solutions are both robust and customized for particular project requirements.
- **Future-Proofs the Codebase:** As technology evolves, software systems must adapt. Design patterns provide a flexible framework that accommodates changes, ensuring that applications remain functional, relevant, and easy to upgrade over time.
- **Supports Scalability:** Given Salesforce's multi-tenant architecture, scalability is a critical concern. Design patterns inherently promote solutions that accommodate increasing data volumes, user growth, and evolving business requirements without compromising performance.
- **Optimizes Performance:** Many Apex design patterns are designed to address common inefficiencies, ensuring that applications run smoothly. By following these optimized solutions, developers can maximize resource utilization while minimizing performance bottlenecks.
- **Reduces Errors and Bugs:** By leveraging established, time-tested solutions, the likelihood of introducing errors is significantly reduced. Since these patterns have been refined over time, they help prevent common pitfalls and improve code reliability.
- **Improves Cost Efficiency:** The structured nature of design patterns leads to reduced development time, fewer errors, and lower maintenance efforts. As a result, organizations benefit from cost savings by avoiding prolonged development cycles and frequent debugging.

3. Type of Apex Design Patterns

Here are some common patterns.

3.1 Singleton Pattern:

The Singleton pattern falls under the creational design patterns category. It ensures that only one instance of a class is created and provides a global access point to it. This approach helps reduce memory consumption, avoid redundant object instantiations, and enhance overall application efficiency.

3.3.1 Key Objective:

The primary goal of the Singleton pattern is to restrict a class to a single instance while making that instance accessible throughout the application lifecycle. This ensures consistent data management and prevents unnecessary reloading of information.

3.2 Use Cases in Salesforce:

- **Managing Configuration Settings**:
  The Singleton pattern is widely used to handle application-level settings, such as API keys, authentication credentials, and custom configurations.
  It ensures that configuration data is loaded once and shared across the system, eliminating redundant queries and improving performance.
- **Tracking Governor Limits**:
  Since Salesforce imposes governor limits on resource usage, Singleton helps centralize and monitor these limits, ensuring that operations remain within acceptable constraints.
- **Encapsulating Global Utility Functions:**
  Common functions like data formatting, encryption, or logging can be stored in a Singleton class, providing a centralized and reusable solution across multiple components.

3.3 Problem Scenario:

Imagine a Salesforce application used by a large organization with multiple departments, each requiring distinct email settings. If these configurations are hardcoded within multiple classes and triggers, the system may experience:

- **Code duplication**, leading to inconsistencies and inefficiencies.
- **Difficulties in updating settings**, as changes need to be manually implemented across various components.
- **Reduced maintainability**, making future modifications complex and error prone.

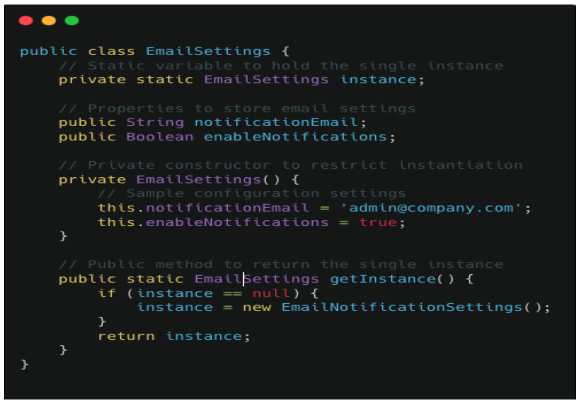
.

3.4 Solution using Singleton Pattern in Apex:

To resolve these challenges, a Singleton-based Email Settings class can be created. This class ensures that email notification settings are managed centrally, providing a single access point for consistent configuration.

4. How This Solution Works

- **Ensures a Single Instance:** The get Instance() method ensures that only **one instance** of Email Settings is created during execution.
- **Centralized Configuration Management:** Instead of defining settings in multiple locations, all components retrieve settings through Email Settings. Get Instance().
- **Enhances Maintainability:** Updates to the settings are made in **one place**, reducing complexity and eliminating inconsistencies.

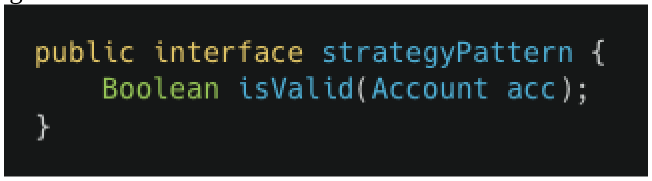


```
public class EmailSettings {
    // Static variable to hold the single instance
    private static EmailSettings instance;

    // Properties to store email settings
    public String notificationEmail;
    public Boolean enableNotifications;

    // Private constructor to restrict instantiation
    private EmailSettings() {
        // Sample configuration settings
        this.notificationEmail = 'admin@company.com';
        this.enableNotifications = true;
    }

    // Public method to return the single instance
    public static EmailSettings getInstance() {
        if (instance == null) {
            instance = new EmailNotificationSettings();
        }
        return instance;
    }
}
```

4.1 Strategy pattern

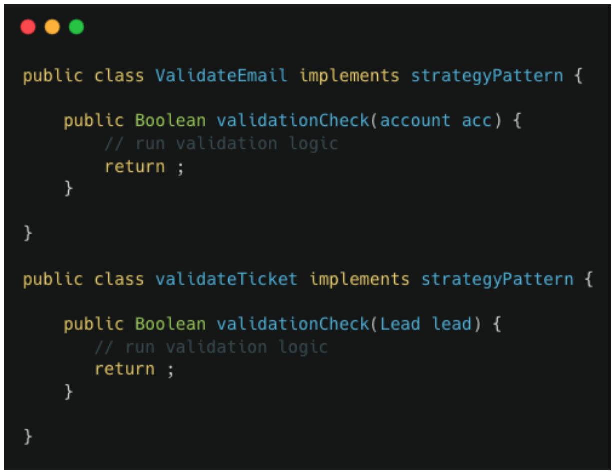
The Strategy pattern involves encapsulating interchangeable algorithms and behaviors within distinct concrete classes that share a common interface. This design enables the application to dynamically select and switch between these options at runtime, promoting flexibility and adaptability.

Step 1: Create an interface for the algorithm:



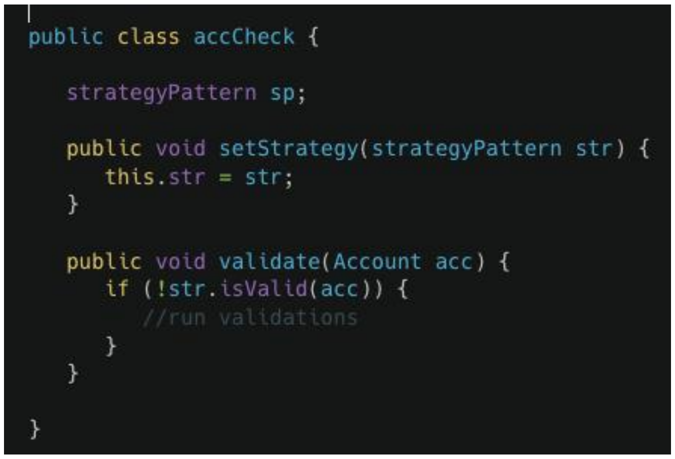
```
public interface strategyPattern {
    Boolean isValid(Account acc);
}
```

Step 2: Create different variations:



```
public class ValidateEmail implements strategyPattern {

    public Boolean validationCheck(account acc) {
        // run validation logic
        return ;
    }

}

public class validateTicket implements strategyPattern {

    public Boolean validationCheck(Lead lead) {
       // run validation logic
       return ;
    }

}
```

Step 3: Run dynamic validation:



```
public class accCheck {

    strategyPattern sp;

    public void setStrategy(strategyPattern str) {
        this.str = str;
    }

    public void validate(Account acc) {
        if (!str.isValid(acc)) {
            //run validations
        }
    }

}
```

5. Factory pattern

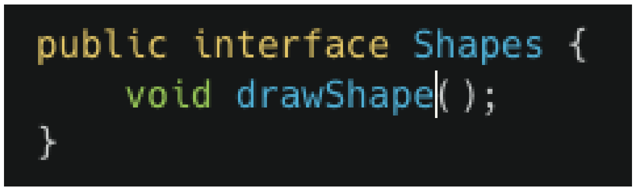
The Factory Design Pattern is a widely adopted creational pattern that enables object creation without exposing the instantiation logic to the client. Instead of directly instantiating objects, the pattern centralizes the object creation process within a dedicated factory class. This approach enhances maintainability, encapsulates complexity, and promotes scalability by allowing modifications to object creation without altering client code. This pattern establishes an interface for creating objects while allowing subclasses to determine which specific class should be instantiated. By delegating object creation to a factory, the system maintains loose coupling, making it easier to introduce new object types with minimal modifications.

5.1 Key Components of the Factory Pattern

The Factory Design Pattern is composed of the following core elements:

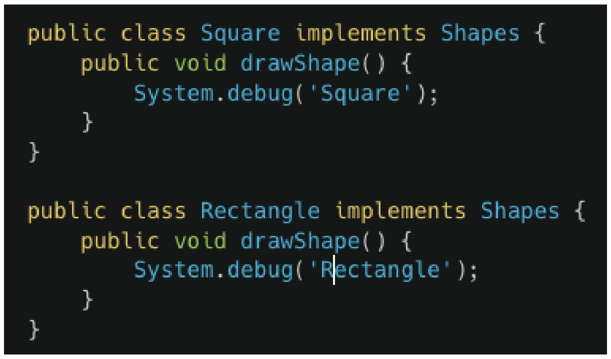
- **Creator (Factory Class):** An abstract class or interface that defines a factory method. This method returns an instance of the **Product** type.
- **ConcreteCreator:** A specific implementation of the **Creator**, which overrides the factory method to return an appropriate **Product** instance.
- **Product (Abstract Type):** Defines the interface for objects that will be created by the factory method.
- **ConcreteProduct:** A class that implements the **Product** interface, representing the actual objects instantiated by the factory.

Step 1: Create Product Interface



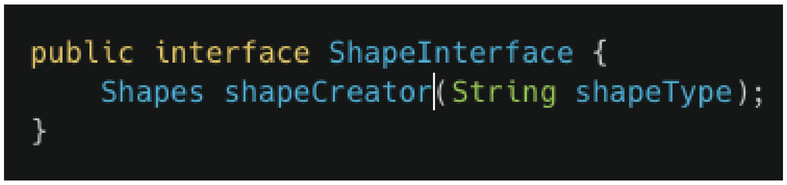
```
public interface Shapes {
    void drawShape();
}
```

Step 2: Implement the Shape Interface



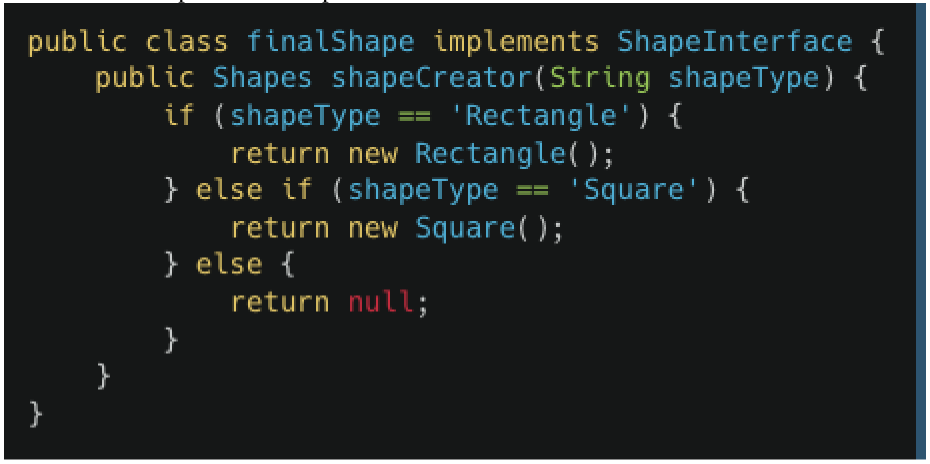
```
public class Square implements Shapes {
    public void drawShape() {
        System.debug('Square');
    }
}

public class Rectangle implements Shapes {
    public void drawShape() {
        System.debug('Rectangle');
    }
}
```

Step 3: Create Creator interface



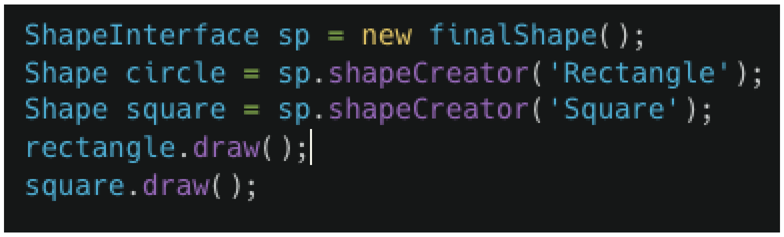
```
public interface ShapeInterface {
    Shapes shapeCreator(String shapeType);
}
```

Step 4: Create ConcreteCreator that implements Shape



```
public class finalShape implements ShapeInterface {
    public Shapes shapeCreator(String shapeType) {
        if (shapeType == 'Rectangle') {
            return new Rectangle();
        } else if (shapeType == 'Square') {
            return new Square();
        } else {
            return null;
        }
    }
}
```

Step 5: Test the logic



```
ShapeInterface sp = new finalShape();
Shape circle = sp.shapeCreator('Rectangle');
Shape square = sp.shapeCreator('Square');
rectangle.draw();
square.draw();
```

Step 6: Output:
Rectangle
Square

6. Conclusion

Leveraging Apex Design Patterns in Salesforce is essential for developing efficient, maintainable, and scalable applications. These patterns provide a structured approach to organizing code, reducing bugs, and accelerating the delivery of new features. While incorporating design patterns requires an initial investment in learning and tooling, their long-term benefits far outweigh the effort. At first glance, design patterns may appear complex, but they are fundamentally rooted in object-oriented programming principles that developers naturally strive to follow. The real challenge is not just understanding these patterns but consistently applying them in real-world scenarios. The next time you develop a new feature, take a moment to consider whether a design pattern could simplify implementation and strengthen your application's architecture.

References

[1] Salesforce, "Apex Design Best Practices"
[2] Available: https://developer.salesforce.com/ja/wiki/apex_code_best_practices
[3] ApexHours, "Apex Design Patterns,"
[4] Available: https://www.apexhours.com/apex-design-patterns/
[5] Medium, "Apex Design Patterns,"
[6] Available: https://medium.com/@sfdcbrewery/apex-design-patterns-sfdc-brewery-salesforce-developer-interview-preparation- series-2c5296a9ed0f
[7] Salesforce Ben, "3 Apex Design Patterns for yor Salesforce Development Team"
[8] Available: https://www.salesforceben.com/3-apex-design-patterns-for-your-salesforce-development-team/
[9] FreeCodeCamp, "The three Types of Design Patterns All Developers Should Know,"
[10] Available: https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/
[11] WIKI, "Separation of concerns,"
[12] Available: https://en.wikipedia.org/wiki/Separation_of_concerns
[13] Trailead, "Understand Separation of Concerns,"
[14] Available: https://trailhead.salesforce.com/content/learn/modules/apex_patterns_sl/apex_patterns_sl_soc