ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Research Article

Open Access

# Strategy Design Pattern Applied on a Mobile App Building

**Nilesh D Kulkarni[1]* and Saurav Bansal[2]**

[1]Sr. Director – Enterprise Architecture, Fortune Brands Home & Security, USA

[2]Sr. Manager - Digital Applications, Fortune Brands Home & Security, USA

**ABSTRACT**

This paper provides the importance and application of design patterns in software engineering, particularly focusing on the Strategy Design Pattern. It outlines how design patterns offer efficient, flexible, and reusable solutions to common problems in object-oriented software development. The paper presents a case study of Strategy Design Pattern's application in a mobile app builder, emphasizing its role in creating adaptable and maintainable software architecture. Additionally incorporates commentary on the SOLID Open and Close principle, explaining how it allows software entities to be extendable without modifying existing code, thus enhancing the scalability and robustness of the application. The OC principle integration with the Strategy Design Pattern demonstrates its practicality in promoting flexible and stable software development.

***Corresponding author**

Nilesh D Kulkarni, Sr. Director – Enterprise Architecture, Fortune Brands Home & Security, USA.

**Received:** March 01, 2022; **Accepted:** March 10, 2022, **Published:** March 18, 2022

**Keywords:** Design Patterns, Strategy, Object, .NET, Software Maintainability

## Introduction

The importance of design experience is widely recognized. How often have you encountered a familiar problem during design, sensing that you've tackled something similar in the past, yet struggling to recall the specifics of where and how it was resolved? If you were able to recall the nuances of that past challenge and the strategy you employed to overcome it, you could leverage that previous experience instead of having to re-explore the solution from scratch.

A design pattern represents a universally recognized solution, widely observed in various cases, that effectively addresses a specific problem in a context that may not be predefined. It offers a highly efficient approach to developing object-oriented software that is not only flexible and elegant but also reusable. The utilization of design patterns facilitates the reuse of successful designs and architectural models. By translating proven technologies and methodologies into design patterns, they become more easily accessible to developers building new systems. Design patterns guide developers in selecting design options that enhance the reusability of a system, while steering clear of choices that could hinder it. Moreover, design patterns can significantly enhance the documentation and maintenance of existing systems by providing a clear and explicit description of class and object interactions, along with their fundamental purposes. In essence, design patterns empower designers to achieve a more effective design more swiftly.

Typically, a design method comprises a set of synthetic notations usually graphical and a set of rules that govern how and when we use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate the design. Each pattern describes a problem which occurs over and over again in the environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over without ever doing it the same way twice [1].

Design patterns describe problems that occur repeatedly, and describe the core of the solution to that problem, in such a way that the solution can be used many times in different contexts and applications. A good design should always be independent of the technology and the design should help both experience and the novice designer to recognize situation in which these designs can be used and reused.

Eric gamma at el in their book Design Patterns, discussed total 23 design patterns clarified by two criteria. The first criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose. Creational patterns concern the purpose of object creation. Structural pattern deals with the composition of classes or objects. Behavioral pattern characterizes the ways in which classes or objects interact and distribute responsibility [2]. The second criteria called scope, specifies whether the pattern applies primarily to the class or to the object.

**Table 1**

| Scope | Purpose | | |
|---|---|---|---|
| | **Creational** | **Structural** | **Behavioral** |
| Class | Factory Method | Adapter | Interpreter Template Method |
| Object | Abstract Factory Builder Prototype Singleton | Adapter Bridge Composite Decorator Façade Flyweight Proxy | Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor |

## UML Basics

The first versions of UML were created by "Three Amigos" — Grady Booch at el defines "The Unified Modeling Language (UML), is a standardized visual language for specifying, constructing, and documenting the artifacts of software systems. It provides a set of diagrams and notations to represent various aspects of software design and architecture, allowing software engineers to communicate, visualize, and model complex systems effectively."

## Three Types of Relations between the Classes

Association relationship: When classes are connected together conceptually, that connection is called an association. As shown in the figure 1., let's examine the association between passenger and airplane. A passenger can sit in an airplane or multiple passengers can sit in an airplane.

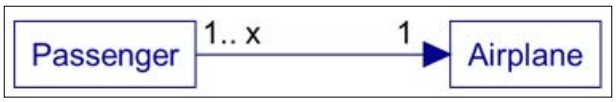


**Figure 1:** Association Relationship

Aggregation relationship: This is a special type of relationship, used to model situations where one class (the whole) contains or is composed of other classes or objects (the parts), and the parts have a lifecycle that is independent of the whole. As shown in the figure 2., next examine the aggregation relationship, an engine (whole) can have many Pistons (parts) similarly an airplane (whole) can have multiple engines (parts) as well as an airplane can have multiple wheels (parts).

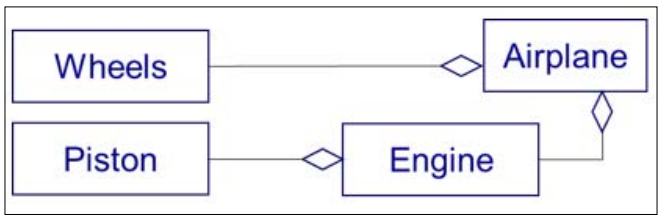


**Figure 2:** Aggregation Relationship

Composition relationship: a composition is a strong type of aggregation where each component in the composite can belong to just one whole. As shown in figure 3., a dog can have a tail, four legs, two ears, and two eyes, but eyes, legs, tail, and ears cannot exist on its own.

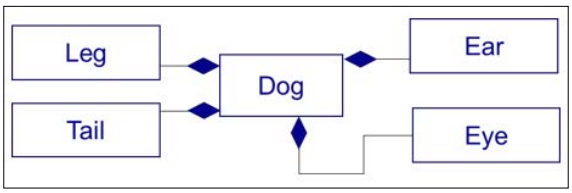


**Figure 3:** Composition Relationship

## Inheritance / Generalization

In this relationship one class (the child class or subclass) can inherit attributes and operations from another (the parent class or superclass). The generalization allows for polymorphism. In generalization, a child is substitutable for parent. That is anywhere the parent appears, the child may appear. The reverse isn't true [3]. As shown in the Figure 4, signifies that "Bus," "Car," and "Truck" inherit from "Vehicle." They are expected to share common characteristics or behaviors that are defined in "Vehicle." For instance, if "Vehicle" has attributes like 'number of wheels' and 'fuel type' and operations like 'start engine ()', then "Bus," "Car," and "Truck" would inherit these operations and attributes.

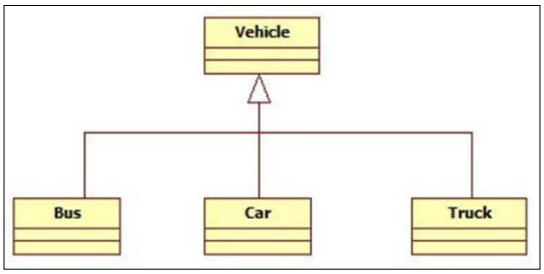


**Figure 4:** Generization

## Interface

An interface is a set of operation that specifies some aspect of classes behavior, and it's set of operation class presents to other classes [3]. As shown in figure 5., the "Electric System" is considered an interface between the light bulb and the light switch. The "Electric System" serves as a contract between the light bulb and the light switch, stipulating that when the switch is turned on, the bulb should light up. Interfaces are used to decouple the implementation and the abstract design, allowing for changes in implementation without affecting the system that uses the interface. Similarly, the light switch and bulb are decoupled from each other, you could replace either the bulb or the switch without needing to change the other, as long as they both adhere to the same electrical system standards. Interface also allows different classes to be treated through a single interface type, the electric system could work with any device that conforms to its standards, not just a light bulb. This could include a fan, a heater, or any other electric device that can be turned on or off.

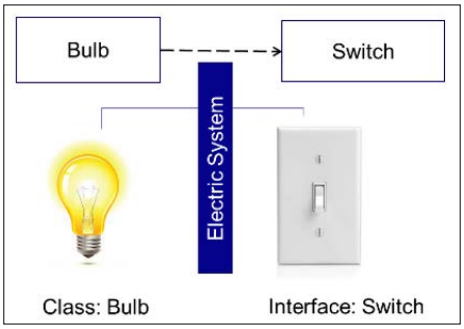


**Figure 5:** Interface representation

**Programming Technologies**

We will using the basic programming tools to show the implementation of the Strategy design pattern.

**NET Framework**

The .NET Framework, is a software development framework designed and supported by Microsoft. It provides a controlled environment for developing and running applications on Windows. Few features listed below

- **Windows-Specific:** The .NET Framework is designed to work on Windows operating systems.
- **Base Class Library (BCL):** It includes a large class library known as the Framework Class Library (FCL), providing user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications.
- **Common Language Runtime (CLR):** Programs written for the .NET Framework execute in a software environment named the Common Language Runtime, which provides services such as security, memory management, and exception handling.
- **Languages:** The .NET Framework supports multiple programming languages, such as C#, VB.NET, and F#.
- **CLI:** Console programming refers to the process of writing software applications that interact with the user through a text-based interface. These applications run in a console or a command-line interface (CLI), where the user inputs text commands and the program provide output in text form.

**Visual Studio Code (VS Code) for .NET Development**

Visual Studio Code is a lightweight, open-source, and cross-platform code editor developed by Microsoft. It's not specific to any one programming language or framework. With the help of extensions, it can support a wide variety of languages and frameworks, including those of the .NET ecosystem. Few features listed below

- **Cross-Platform:** VS Code runs on Windows, Linux, and macOS.
- **Extensions:** The C# extension by Omni Sharp adds support for .NET development, including features like IntelliSense, debugging, project file navigation, and run tasks.
- **Lightweight Editor:** VS Code is designed to be a fast and lightweight editor, with a smaller footprint than a full IDE like Visual Studio.
- **Integrated Terminal:** Developers can use the integrated terminal to execute .NET CLI commands, enabling them to create, build, run, and test .NET applications.
- **Git Integration:** VS Code has built-in Git support, which is essential for modern software development workflows.
- **Language Features:** VS Code with the C# extension supports advanced language features like code refactoring, unit testing, and code snippets for .NET.

**Behavioral Pattern**

Behavioral design patterns are a set of design patterns in software engineering that focus on the interaction and communication between different objects and classes in a system. They help in defining how objects collaborate and communicate with each other to achieve a specific behavior or functionality. Behavioral design patterns primarily deal with the delegation of responsibilities among objects and how they interact to accomplish tasks.

Common behavioral design patterns -

- **Strategy Pattern:** The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows to select an algorithm or behavior at runtime without altering the client code that uses it. This pattern is useful for providing multiple ways to accomplish a task.
- **Observer Pattern:** This pattern defines a one-to-many relationship between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. It's commonly used in implementing distributed event handling systems.
- **Command Pattern:** The command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of requests. It also provides support for undoable operations.
- **Chain of Responsibility Pattern:** In this pattern, a request is passed along a chain of handlers. Each handler decides either to process the request or pass it to the next handler in the chain. It's commonly used in implementing event-driven systems like event handling in GUI-Graphical User Interface applications.
- **State Pattern:** The state pattern allows an object to alter its behavior when its internal state changes. It represents various states of an object as separate classes and delegates the state specific behavior to these classes. This pattern is useful when there is an object that needs to change its behavior dynamically based on its internal state.

**Lift Map App - Use Case**

A team of entrepreneurs came together to create a company called "Lift." The journey of "Lift" began with the development of an app called "Lift Map." This app provided driving directions to end users of the app, helping them navigate to their destinations efficiently.

Following its significant success, numerous current users expressed their desire to utilize the app for pedestrian directions as well. These requests were consistently made by users through their comments, and due to the ongoing demand and to enhance the adoption rate, the Lift team incorporated a new feature called "Walk" in the subsequent version 2. Shortly after the release of version 2, prompted by substantial user demand, Lift introduced another option that allowed users to incorporate public transport into their routes.

The latest demand emerged from cyclists, urging the addition of cycling routes, particularly for dedicated bike tracks.

While the investors in "Lift" expressed satisfaction with these new features, the app developers were less pleased. Each time a new routing algorithm was added, the code of the navigator doubled in size, resulting in a tangled and unwieldy codebase "Code Spaghetti". This not only made it challenging to manage the code but also led to a decline in performance, raising concerns among users.

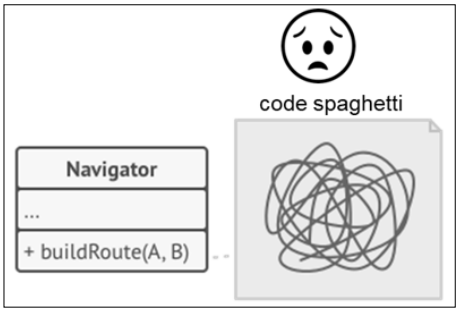


**Figure 6:** Code Spaghetti

**Behavioral Pattern-Strategy**

The Strategy Design Pattern defines a family of algorithms which encapsulates each of them, and makes them interchangeable. It allows the client to choose the appropriate algorithm or strategy to use at runtime without altering the client code. This pattern promotes flexibility, extensibility, and maintainability by separating the algorithms from the client code.

Here are the key components and participants in the Strategy Pattern:

1. **Context:** This is the class that maintains a reference to the selected strategy object and is responsible for executing the algorithm. The context object doesn't implement the algorithm itself but delegates the task to the strategy.
2. **Strategy:** This is an interface or an abstract class that defines the common interface for all concrete strategies. It usually consists of one or more methods that represent the algorithm's contract.
3. **Concrete Strategies:** These are the actual algorithm implementations that implement the Strategy interface. Each concrete strategy provides a specific implementation of the algorithm.
4. The Client creates a specific strategy object and passes it to the context.

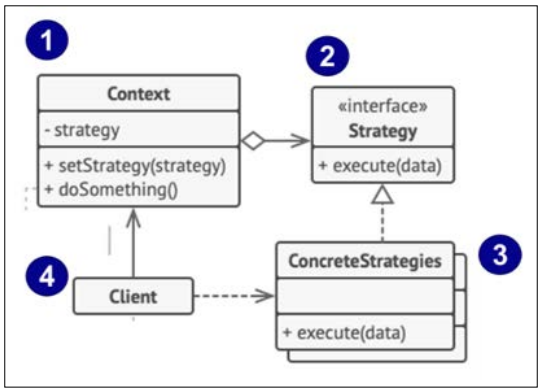


**Figure 7:** Strategy Design Pattern

**Design Pattern Application**

The figure 8, illustrates an application of the Strategy Design Pattern. The central component of the design is the `Navigator` class. This class contains a reference to the `RouteStrategy` interface, which declares the `buildRoute(A, B)` method. This method is intended to build a route between points A and B. The `Navigator` class does not implement the route-building logic itself but delegates that responsibility to the `RouteStrategy` interface.

The `RouteStrategy` interface is then implemented by several concrete strategy classes:

1. `Road Strategy` - Provides an algorithm for route building by roads, suitable for vehicular traffic.
2. `Walking Strategy` - Has a logic that produces a pedestrian-friendly route, prioritizing sidewalks and paths.
3. `Public Transport Strategy` - Contains an algorithm for constructing a route based on available public transportation, like buses and trains.

The Strategy Design Pattern as depicted in this diagram provides a flexible and maintainable way to vary the algorithm used for route building in a navigation application. Which promotes the use of composition over inheritance and adheres to the "open/closed principle," allowing for new strategies to be added without modifying the client. Open Close Principle have two primary attributes (1) They are open for extension – this means that the behavior of the module can be extended as the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. (2) they are closed for modification – extending the behavior off module does not result in changes to the source, or binary, code of the module [4].

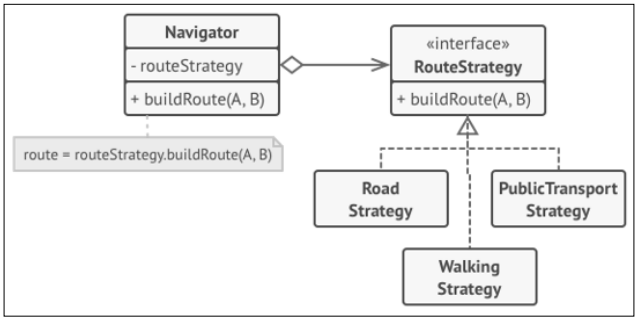


**Figure 8:** Applying Strategy Design Pattern

**Strategy Construction**

The figure 9 shows the ability to extend the RouteStrategy by adding a new concrete strategy class "Cycling Strategy", the algorithm to build a route for a cyclist is within the buildRoute() implemented by Cycling Strategy class. This pattern not only enabled extension to the existing capabilities, but also did not affect the client implimentation.

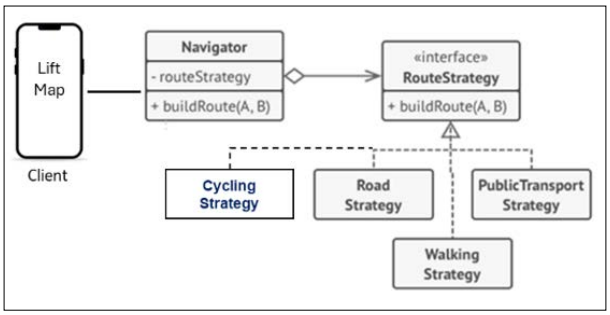


**Figure 9:** Strategy Implimentation

Code Construction

The representation of the code using C# , Visual Studio Code and .Net Framework shown below-

```
using System;
namespace StrategyPatternDemo
{
   // The 'Strategy' abstract class
   public interface IRouteStrategy
   {
      void MapMyRoute();
   }

   // 'ConcreteStrategy' class
   public class RoadMap : IRouteStrategy
   {
      public void MapMyRoute()
      {
          Console.WriteLine("Driving Direction - Drive 5 miles straight to the North");
      }
   }
```

```
   // Another 'ConcreteStrategy' class
   public class WalkMap : IRouteStrategy
   {
      public void MapMyRoute()
      {
         Console.WriteLine("Walking Direction - Walk for next 2
min towards Mill Blvd.");
      }
   }

   // Yet another 'ConcreteStrategy' class
   // Assume similar implementation for PublicTransportMap
   public class PublicTransportMap : IRouteStrategy
   {
      public void MapMyRoute()
      {
         // Implementation for public transport route mapping
      }
   }

   // The 'Context' class
   public class Navigator
   {
      private IRouteStrategy _routeStrategy;

      public Navigator(IRouteStrategy routeStrategy)
      {
         _routeStrategy = routeStrategy;
      }

      public void ContextInterface()
      {
         _routeStrategy.MapMyRoute();
      }
   }
}


public class Program
{
   public static void Main(string[] args)
   {
using StrategyPatternDemo;
StrategyPatternDemo.Navigator context;

         Console.WriteLine("Please select your travel strategy:");
          Console.WriteLine("A - RoadMap");
          Console.WriteLine("B - WalkMap");
          Console.WriteLine("C - PublicTransportMap");
          ConsoleKeyInfo cki;

          do {
             cki = Console.ReadKey(true);
             if (cki.Key == ConsoleKey.A)
             {
                context = new Navigator(new RoadMap());
             }
             else if (cki.Key == ConsoleKey.B)
             {
                context = new Navigator(new WalkMap());
             }
             else if (cki.Key == ConsoleKey.C)
             {
             context = new Navigator(new PublicTransportMap());
              }
             else
             {
             Console.WriteLine("Invalid Input, press esc to exit !");
             }
          } while (cki.Key != ConsoleKey.Escape);    }
}
```

**Design Pattern and Software Maintainability**

The original study to evaluate the impact of design patterns on software maintenance was applied by [5]. They conducted an experiment call PatMain by comparing the maintainability of two implementations of an application, one using a design pattern and the other using a simple alternative. They used four different subject systems in the same programming language. They addressed five patterns - decorator, composite, abstract factory, observer and visitor. The researchers measure the time and correctness of the given maintenance task for professional participants. They found that it was useful to use a design pattern but in case where simple solution is preferred, it is good to follow the software engineer common sense about whether to use a pattern or not, and in case of uncertainty it is better to use a pattern as a default approach.

**Conclusion**

A design pattern is a generalized reusable solution two commonly occurring problem in a software design. It can be defined as a description or template for how to solve a problem that can be used in many different situations [6]. In this paper, we aim to demonstrate the practical application of the strategy design pattern in a specific use case. Design patterns serve as invaluable communication tools and expedite the design process. They empower solution providers to focus on solving the business problem while promoting reusability in the design. Reusability extends not only to individual components but also to the entire design process, from problem-solving to the final solution. The ability to apply patterns that offer repeatable solutions is well worth the time invested in learning them. There are promising results indicating that the utilization of design patterns enhances quality and contributes to maintainability. The proportion of source code lines involved in design patterns within a system shows a strong correlation with maintainability. However, it's important to note that these findings represent just a small step in the empirical analysis of software quality concerning design patterns. Design patterns should facilitate the reuse of software architecture across different application domains and promote the reuse of flexible components.

**References**

1. C Alexander, S Ishikawa, M Silverstein, M Jacobson, I Fiksdahl-King, et al. (1977) A Pattern Language. Oxford University Press https://global.oup.com/academic/product/a-pattern-language-9780195019193?cc=us&lang=en&.
2. E Gamma, R Helm, R Johnson, J Vlissides (1995) Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley https://www.javier8a.com/itc/bd1/articulo.pdf.
3. J Schmuller (1999) Sams Teach Yourself Uml in 24 Hours. SAMS https://nibmehub.com/opac-service/pdf/read/Sams%20teach%20yourself%20UML%20in%2024%20hours%20by%20Joseph%20Schmuller%20-A.pdf.
4. R C Martin (2006) Agile Principles, Patterns, and Practices in C#. Prentice Hall https://ivanderevianko.com/wp-content/uploads/2013/10/Agile-Principles-Patterns-and-Practices-in-C.pdf.

5. L Prechelt, B Unger, WF Tichy, P Brossler, LG Votta (2001) A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27: 1134-1144.

6. C Zhang, D Budgen (2012) What Do We Know about the Effectiveness of Software Design Patterns?. IEEE Transactions on Software Engineering 38: 1213-1231.