15

Low Level Design

MASTERING DESIGN PATTERNS

Manish Jaiswal

Manish Jaiswal

LLD

MASTERING

Factory Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Factory?

The Factory Design Pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code.

swipe

Manish Jaiswal

LLD

02

Real-World Example: Logging System

Imagine a logging system where we can log messages to different targets such as a file, database, or console. We will use the Factory Design Pattern to create a logger based on the specified target.

swipe

Manish Jaiswal

LLD

03

Step 1: Define the Logger Interface

```
1 public interface ILogger
2 {
3     void Log(string message);
4 }
```

swipe

Manish Jaiswal

LLD

04

Step 2: Implement Concrete Loggers

```
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // Log to a file
    }
}

public class DatabaseLogger : ILogger
{
    public void Log(string message)
    {
        // Log to a database
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

swipe

Manish Jaiswal

LLD

05

Step 3: Create the Logger Factory

```
1  public class LoggerFactory
2  {
3      public ILogger CreateLogger(string type)
4      {
5          switch (type)
6          {
7              case "File":
8                  return new FileLogger();
9              case "Database":
10                 return new DatabaseLogger();
11             case "Console":
12                 return new ConsoleLogger();
13             default:
14                 throw new ArgumentException("Invalid logger type");
15         }
16     }
17 }
```

swipe

Manish Jaiswal

LLD

06

Step 4: Using the Logger Factory

```
1  public class Program
2  {
3      public static void Main()
4      {
5          LoggerFactory factory = new LoggerFactory();
6
7          ILogger logger = factory.CreateLogger("Console");
8          logger.Log("This is a console log message.");
9
10         logger = factory.CreateLogger("File");
11         logger.Log("This is a file log message.");
12     }
13 }
```

swipe

Manish Jaiswal

LLD

07

Why Use Factory?

Encapsulation: Hides the instantiation logic.

Flexibility: Makes it easier to introduce new types of objects.

Decoupling: Reduces dependency on concrete classes.

Maintainability: Simplifies the maintenance and extension of the codebase.

swipe

Manish Jaiswal

LLD

08

Where to Use Factory?

- When the exact type of object to create is not known until runtime.
- When you want to centralize the creation logic of objects.
- When you need to manage and maintain a group of related subclasses.

swipe

Manish Jaiswal

LLD

MASTERING

Abstract Factory Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Abstract Factory?

The Abstract Factory is a creational design pattern that provides an interface for creating families of related objects without specifying their concrete classes. Think of it as a super factory that delegates object creation to specialized sub-factories based on your needs.

swipe

Manish Jaiswal

LLD

02

Real-World Example: Building Furniture

Let's build a simple furniture example! We have two furniture factories – ModernFactory and ClassicFactory – each creating chairs and sofas that belong to their respective style.

swipe

Manish Jaiswal

LLD

```
// Abstract Factory Interface
public interface IFurnitureFactory
{
    IChair CreateChair();
    ISofa CreateSofa();
}

// Concrete Factories
public class ModernFactory : IFurnitureFactory
{
    public IChair CreateChair() => new ModernChair();
    public ISofa CreateSofa() => new ModernSofa();
}

public class ClassicFactory : IFurnitureFactory
{
    public IChair CreateChair() => new ClassicChair();
    public ISofa CreateSofa() => new ClassicSofa();
}

// Abstract Products (Interfaces)
public interface IChair
{
    void Sit();
}

public interface ISofa
{
    void LieDown();
}
```

swipe

Manish Jaiswal

LLD

```
// Concrete Products
public class ModernChair : IChair
{
    public void Sit()
    {
        Console.WriteLine("Sitting on a Modern Chair");
    }
}

public class ClassicChair : IChair
{
    public void Sit()
    {
        Console.WriteLine("Sitting on a Classic Chair");
    }
}

public class ModernSofa : ISofa
{
    public void LieDown()
    {
        Console.WriteLine("Lying down on a Modern Sofa");
    }
}

public class ClassicSofa : ISofa
{
    public void LieDown()
    {
        Console.WriteLine("Lying down on a Classic Sofa");
    }
}
```

swipe

Manish Jaiswal

LLD

```
// Client Code
public class Client
{
    private IFurnitureFactory factory;

    public Client(IFurnitureFactory factory)
    {
        this.factory = factory;
    }

    public void CreateFurniture()
    {
        IChair chair = factory.CreateChair();
        ISofa sofa = factory.CreateSofa();

        chair.Sit();
        sofa.LieDown();
    }
}

public static void Main(string[] args)
{
    Client client = new Client(new ModernFactory());
    client.CreateFurniture();
    // Output: Sitting on a Modern Chair, Lying down on a Modern Sofa

    client = new Client(new ClassicFactory());
    client.CreateFurniture();
    // Output: Sitting on a Classic Chair, Lying down on a Classic Sofa
}
```

swipe

Manish Jaiswal

LLD

06

Why Use Abs. Factory?

- **Decoupling:** Helps in decoupling client code from concrete classes.
- **Consistency:** Ensures consistency among products.
- **Scalability:** Makes it easier to introduce new product families without changing existing code.

swipe

08

Where to Use Abs. Factory?

- UIs with different themes (e.g., light/dark mode)
- Database access with various providers (e.g., SQL Server, Oracle)
- Document creation with different formats (e.g., PDF, Word)

swipe

Manish Jaiswal

.NET / C#

MASTERING

Singleton Design Pattern

swipe

Manish Jaiswal

.NET / C#

01

What is Singleton?

At its core, the Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. Think of it as a guardian of a single resource, ensuring that there's only one gatekeeper controlling access.

swipe

Manish Jaiswal

.NET / C#

02

Why Use Singleton?

Singletons are like superheroes of efficiency. They're perfect for scenarios where exactly one object is needed to coordinate actions across the system. Whether it's managing a shared resource, controlling access to a database, or regulating configuration settings, Singletons shine by streamlining access and minimizing resource usage.

swipe

Manish Jaiswal

.NET / C#

03

Where to Use Singleton?

From managing logging utilities to handling thread pools, Singletons find their home in situations where we need precisely one instance to serve a common purpose. They're particularly handy in scenarios where creating multiple instances could lead to resource contention or unnecessary overhead.

swipe

Manish Jaiswal

.NET / C#

04

Singleton in C#.

```
1  public class Singleton
2  {
3      private static Singleton instance;
4
5      // Object used for locking
6      private static readonly object lockObject = new object();
7
8      private Singleton() { }
9
10     public static Singleton GetInstance()
11     {
12         if (instance == null)
13         {
14             lock (lockObject)
15             {
16                 instance ??= new Singleton();
17             }
18         }
19         return instance;
20     }
21 }
```

swipe

Manish Jaiswal

.NET / C#

05

Locking for Thread Safety

In our code implementation, we've applied locking (using object) to prevent concurrent access and ensure thread safety. This ensures that only one instance of the Singleton class is created, even in multi-threaded environments.

swipe

Manish Jaiswal

LLD

MASTERING

Prototype Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Prototype?

Ever feel like creating a new object is a whole ordeal, especially when it involves complex configurations or nested objects? The Prototype design pattern can be your hero!

The Prototype pattern is like a cloning machine for objects in our code! It allows us to create new objects based on existing ones, but without tying our code to their specific classes. Think of it as making copies of a blueprint rather than building from scratch each time.

swipe

Manish Jaiswal

LLD

02

Why Use Prototype?

Efficiency: Cloning an existing object is often faster than creating a new one from scratch, especially for complex objects.

Customization: You can modify the cloned object without affecting the original, making it ideal for creating variations.

Decoupling: The client code doesn't need to know the specific class of the object being cloned, promoting loose coupling.

swipe

Manish Jaiswal

LLD

03

Where to Use Prototype?

- When creating objects with expensive initialization processes (e.g., database connections, network resources).
- When you need multiple objects with slight variations based on a common template.
- When the cost of creating a new object is high due to complex internal state.

swipe

Manish Jaiswal

LLD

04

Real-World Example: Prototyping a Document

Imagine a document editing software where users can create different types of documents (reports, invoices, etc.). Each document type might have a specific layout, formatting, and default content.

swipe

Manish Jaiswal

LLD

05

Code

```
1
2  public abstract class Document
3  {
4    public string Title { get; set; }
5    public List<string> Content { get; set; }
6
7    public abstract Document Clone();
8  }
9
10 public class Report : Document
11 {
12   public override Document Clone()
13   {
14     return new Report()
15     {
16       Title = this.Title,
17       Content = new List<string>(this.Content) // Deep copy for list
18     };
19   }
20 }
```

swipe

Manish Jaiswal

LLLD

06

Code

```
1  public class Invoice : Document
2  {
3    public string Client { get; set; }
4
5    public override Document Clone()
6    {
7      return new Invoice()
8      {
9        Title = this.Title,
10       Content = new List<string>(this.Content),
11       Client = this.Client
12     };
13   }
14 }
15
16 // Usage
17 Report templateReport = new Report() { Title = "Monthly Sales Report" };
18 Report newReport = (Report)templateReport.Clone();
19 newReport.Title = "Quarterly Sales Report";
20
21 // Similarly, create Invoice clones with specific client details.
```

swipe

07

Conclusion

In this example, the Document class defines the core structure and a Clone method for creating copies. Specific document types like Report and Invoice inherit from Document and implement their own cloning logic.
This allows users to quickly create new documents based on existing templates, saving time and ensuring consistency.

Key takeaway: The Prototype pattern offers a powerful and flexible way to clone objects, promoting efficiency, customization, and loose coupling.

Manish Jaiswal

LLD

MASTERING

Builder Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Builder?

It's a creational pattern that separates the construction of a complex object from its representation. In simpler terms, it allows you to construct objects step by step, controlling every aspect of their creation.

swipe

Manish Jaiswal

LLD

02

Why Use Builder?

Imagine dealing with an object that requires numerous parameters to be set during its construction. The Builder Pattern comes to the rescue by providing a clean and flexible solution, eliminating the need for multiple constructors or telescoping constructors.

swipe

Manish Jaiswal

LLD

03

Where to Use Builder?

Anywhere you need to create complex objects with varying configurations! Whether it's building user profiles, crafting customized orders, or constructing intricate game characters, the Builder Pattern shines in scenarios where object creation involves multiple steps and options.

swipe

Manish Jaiswal

LLD

04

Example: Pizza ordering system.

Imagine you're building a pizza ordering system. 🍕 Let's use the Builder Pattern to construct delicious pizzas with different toppings and crust types!

swipe

Manish Jaiswal

LLD

05

Code

```
1
2  public class Pizza
3  {
4      public string Toppings { get; set; }
5      public string CrustType { get; set; }
6  }
7
8  public class PizzaBuilder
9  {
10     private Pizza _pizza = new Pizza();
11
12     public PizzaBuilder AddToppings(string toppings)
13     {
14         _pizza.Toppings = toppings;
15         return this;
16     }
17
18     public PizzaBuilder ChooseCrust(string crust)
19     {
20         _pizza.CrustType = crust;
21         return this;
22     }
23
24     public Pizza Build()
25     {
26         return _pizza;
27     }
28 }
```

swipe

Manish Jaiswal

LLD

06

Code

```
1
2  class Program
3  {
4      static void Main(string[] args)
5      {
6          // Constructing a custom pizza using the Builder Pattern
7          Pizza customPizza = new PizzaBuilder()
8              .AddToppings("Pepperoni, Mushrooms, Olives")
9              .ChooseCrust("Thin Crust")
10             .Build();
11
12
13         Console.WriteLine("Custom Pizza:");
14         // Output: Custom Pizza:
15         Console.WriteLine($"Toppings: {customPizza.Toppings}");
16         // Output: Toppings: Pepperoni, Mushrooms, Olives
17         Console.WriteLine($"Crust: {customPizza.CrustType}");
18         // Output: Crust: Thin Crust
19     }
20 }
```

swipe

Manish Jaiswal

.NET / C#

MASTERING

Adapter Design Pattern

swipe

Manish Jaiswal

.NET / C#

01

What is Adapter?

Adapter Design Pattern acts as a bridge between incompatible interfaces, allowing them to work together seamlessly. It converts the interface of a class into another interface that a client expects.

swipe

Manish Jaiswal

.NET / C#

02

Why Use Adapter?

In real-world scenarios, systems evolve independently, leading to incompatible interfaces between components. Adapter Pattern helps in integrating such components without modifying their existing code, thus promoting code reusability and flexibility.

swipe

Manish Jaiswal

.NET / C#

03

Where to Use Adapter?

Adapter Pattern is ideal when integrating legacy systems with modern ones, incorporating third-party libraries, or working with diverse APIs. It's invaluable in scenarios where interface mismatches hinder interoperability.

swipe

Manish Jaiswal

.NET / C#

04

How to implemant?

To implement Adapter Pattern, create an adapter class that implements the target interface and wraps an instance of the adaptee class. The adapter class then maps calls from the target interface to the adaptee's interface.

swipe

Manish Jaiswal

.NET / C#

05

Payment Gateway Intergration

let's consider a scenario where a client's e-commerce platform needs to integrate various payment gateways to process transactions. One such gateway is PayPal, which has its own unique interface.

swipe

Manish Jaiswal

.NET / C#

06

Code

```
1
2  // Target interface expected by the client
3  interface IPaymentGateway
4  {
5      void ProcessPayment(double amount);
6  }
7
8  // Adaptee - PayPal payment gateway with its unique interface
9  class PayPalService
10 {
11     public void MakePayment(double amount)
12     {
13         Console.WriteLine($"PayPal payment of {amount} processed successfully.");
14     }
15 }
```

swipe

Manish Jaiswal

.NET / C#

07

Code

```
// Adapter for PayPalService
class PayPalAdapter : IPaymentGateway
{
    private PayPalService _payPalService;

    public PayPalAdapter(PayPalService payPalService)
    {
        _payPalService = payPalService;
    }

    public void ProcessPayment(double amount)
    {
        _payPalService.MakePayment(amount);
    }
}
```

swipe

Manish Jaiswal

.NET / C#

08

Code

```
// Client code
class PaymentProcessor
{
    public void ProcessPayment(IPaymentGateway paymentGateway, double amount)
    {
        paymentGateway.ProcessPayment(amount);
    }
}

class Program
{
    static void Main(string[] args)
    {
        PaymentProcessor processor = new PaymentProcessor();

        // Integrating PayPal payment gateway using adapter
        PayPalService payPalService = new PayPalService();
        IPaymentGateway paypalAdapter = new PayPalAdapter(payPalService);
        processor.ProcessPayment(paypalAdapter, 100.0);

        // Similar integration can be done for other payment gateways
        // using adapters
    }
}
```

swipe

Manish Jaiswal

LLD

MASTERING

Bridge Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Bridge?

The Bridge Design Pattern is a structural pattern that decouples an abstraction from its implementation, allowing both to vary independently. It helps in reducing the complexity by separating abstraction and implementation into different class hierarchies.

swipe

Manish Jaiswal

LLD

02

Why Use Bridge?

Scalability: Easily extend both abstraction and implementation independently.

Maintainability: Simplifies code maintenance by minimizing interdependencies.

Flexibility: Enhances the ability to switch between implementations dynamically.

swipe

Manish Jaiswal

LLD

03

Where to Use Bridge?

Multiple Implementations: When you have multiple ways to perform the same operation (e.g., different database access techniques).

Abstraction from Implementation: When you need to isolate core functionalities from specific details of the platform or technology.

Runtime Flexibility: Want to switch between implementations dynamically at runtime? Bridge allows for that!

swipe

Manish Jaiswal

LLD

04

Example: Dynamic Database Access

Imagine an application that can connect to different databases (e.g., MySQL, SQL Server) based on user configuration. The Bridge Design Pattern allows us to achieve this elegantly.

swipe

Manish Jaiswal

LLD

```
// Abstraction (IDataAccess)
public interface IDataAccess
{
    List<string> GetData(string query);
}

// Implementations (Data Access Providers)
public class MySqlDataAccess : IDataAccess
{
    private readonly string _connectionString;

    public MySqlDataAccess(string connectionString)
    {
        _connectionString = connectionString;
    }

    public List<string> GetData(string query)
    {
        // Connect to MySQL database using _connectionString
        // Execute query and return data
    }
}
```

swipe

```
1
2  public class SqlServerDataAccess : IDataAccess
3  {
4      private readonly string _connectionString;
5
6      public SqlServerDataAccess(string connectionString)
7      {
8          _connectionString = connectionString;
9      }
10
11     public List<string> GetData(string query)
12     {
13         // Connect to SQL Server database using _connectionString
14         // Execute query and return data
15     }
16 }
17
18 // Business Logic (using Bridge)
19 public class DataManager
20 {
21     private readonly IDataAccess _dataAccess;
22
23     public DataManager(IDataAccess dataAccess)
24     {
25         _dataAccess = dataAccess;
26     }
27
28     public List<string> GetCustomerData()
29     {
30         string query = "SELECT * FROM Customers";
31         return _dataAccess.GetData(query);
32     }
33 }
```

swipe

Manish Jaiswal

LLD

07

Code: Usage

```
// Usage (Dynamic Database Selection)
string userSelectedDb = GetUserSelectedDatabase(); // Get user preference

IDataAccess dataAccess;
if (userSelectedDb == "MySql")
{
    dataAccess = new MySqlDataAccess("connection_string_for_mysql");
}
else
{
    dataAccess = new SqlServerDataAccess(
        "connection_string_for_sqlserver");
}

DataManager manager = new DataManager(dataAccess);
List<string> customerData = manager.GetCustomerData();
```

swipe

08

Conclusion

Here's the breakdown:

- We have an abstraction for IDataAccess that defines core functionalities like GetData(string query).
- Concrete implementations like MySqlDataAccess and SqlServerDataAccess handle the specific connection logic and query execution for each database type.

This way, we can switch between databases at runtime by simply changing the concrete implementation injected into our business logic.

Manish Jaiswal

LLD

MASTERING

Composite Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Composite?

Imagine treating a group of objects like a single object! That's the magic of the Composite Design Pattern. It lets you compose objects into tree structures, representing part-whole hierarchies. Clients can then interact with individual objects or entire groups uniformly.

swipe

Manish Jaiswal

LLD

02

Why Use Composite?

Unified Treatment: Apply operations to single objects or entire structures seamlessly.

Hierarchical Organization: Model complex systems with nested components, making code more readable.

Flexibility: Easily add new object types to the hierarchy without modifying existing code.

swipe

Manish Jaiswal

LLD

03

Where to Use Composite?

- When your application deals with tree-like data like file systems, organizational structures, or graphical elements.

- When you need to perform operations on both individual elements and composite groups consistently.

swipe

Manish Jaiswal

LLD

04

Real-World Example: File System

Think of your computer's file system. Folders act as composites, holding files (leaves) and other folders (sub-composites). Both folders and files inherit from a common base class (e.g., IFileSystemEntry) with operations like "GetSize()" or "Move()".

swipe

Manish Jaiswal

LLD

05

Code

```
// Base interface for all file system entries
public interface IFileSystemEntry
{
    // Get the size of the entry
    int GetSize();

    // Move the entry to a new path
    void Move(string newPath);
}

// Represents a single file
public class File : IFileSystemEntry
{
    // Implementation specific to files
    public int GetSize() { ... }

    // Implementation specific to files
    public void Move(string newPath) { ... }
}
```

swipe

```
// Represents a folder containing entries
public class Folder : IFileSystemEntry
{
  // List of child entries in the folder
  private List<IFileSystemEntry> entries;

  public Folder()
  {
    entries = new List<IFileSystemEntry>();
  }

  // Add a child entry to the folder
  public void AddEntry(IFileSystemEntry entry)
  {
    entries.Add(entry);
  }

  // Recursively calculate the total size of
  // the folder and its contents
  public int GetSize()
  {
    int totalSize = 0;
    foreach (var entry in entries)
    {
      totalSize += entry.GetSize();
    }
    return totalSize;
  }

  // Implementation specific to folders
  public void Move(string newPath) { ... }
}
```

swipe

Manish Jaiswal

LLD

07

Code: Usage

```
1  public class Client
2  {
3    public static void Main(string[] args)
4    {
5      // Create a root folder
6      var rootFolder = new Folder("Documents");
7
8      // Add files and subfolders
9      rootFolder.AddEntry(new File("report.txt"));
10     var subFolder = new Folder("Images");
11     subFolder.AddEntry(new File("photo.jpg"));
12     subFolder.AddEntry(new File("scan.pdf"));
13     rootFolder.AddEntry(subFolder);
14
15     // Get the total size of the document folder
16     // (including subfolders and files)
17     int totalSize = rootFolder.GetSize();
18     Console.WriteLine($"Total size of Documents folder: {totalSize} bytes");
19   }
20 }
```

swipe

Manish Jaiswal

LLD

08

Conclusion

The Composite Design Pattern offers a powerful approach to structuring hierarchical data and operations. By leveraging this pattern, you can create cleaner, more maintainable, and flexible applications.

swipe

Manish Jaiswal

LLD

MASTERING

Facade Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Facade?

Imagine a fancy restaurant with a complex menu. The Facade acts like the head chef, offering a curated set of dishes (methods) that combine the best of the kitchen (subsystems) without exposing all the underlying recipes (implementation details).

The Facade is a class that provides a simplified interface to a complex subsystem. It hides the inner workings and delegates tasks to the appropriate classes behind the scenes.

swipe

Manish Jaiswal

LLD

02

Why Use Facade?

Reduced Complexity: Clients (your code) interact with a single, easy-to-understand interface, making your code cleaner and more maintainable.

Improved Decoupling: Changes within the subsystem won't break your client code as it only interacts with the Facade.

Loose Coupling: The Facade doesn't depend on concrete implementation details of the subsystem classes, promoting flexibility.

swipe

Manish Jaiswal

LLD

03

Where to Use Facade?

Complex Subsystems: If your code interacts with a large number of classes with intricate dependencies, a Facade can simplify usage.

Legacy Systems: For integrating with older, less maintainable codebases, a Facade can provide a cleaner interface.

swipe

Manish Jaiswal

LLD

04

Example: E-Commerce Checkout Process

Consider an e-commerce platform with a labyrinthine checkout process involving **inventory** management, **payment** processing, and **order** fulfillment. Let's simplify this using the Facade pattern

swipe

Manish Jaiswal

LLD

05

Code

```
public class CheckoutFacade
{
    private readonly InventoryService _inventoryService;
    private readonly PaymentService _paymentService;
    private readonly OrderFulfillmentService _orderFulfillmentService;

    public CheckoutFacade()
    {
        _inventoryService = new InventoryService();
        _paymentService = new PaymentService();
        _orderFulfillmentService = new OrderFulfillmentService();
    }

    public bool PlaceOrder(int productId, int quantity, PaymentDetails paymentDetails)
    {
        bool orderPlaced = false;

        if (_inventoryService.IsProductAvailable(productId, quantity))
        {
            if (_paymentService.ProcessPayment(paymentDetails))
            {
                orderPlaced = _orderFulfillmentService.PlaceOrder(productId, quantity);
            }
        }

        return orderPlaced;
    }
}
```

swipe

06

Conclusion

In this example, the CheckoutFacade simplifies the checkout process for clients. It abstracts away the complexities of inventory management, payment processing, and order fulfillment, providing a single method PlaceOrder for placing an order.

The Facade Design Pattern empowers developers to tame complexity, making systems more manageable and easier to maintain.

Manish Jaiswal

LLD

MASTERING

Flyweight Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Flyweight?

The Flyweight Design Pattern is a structural pattern that minimizes memory usage by sharing as much data as possible with similar objects. It is especially useful when working with a large number of objects that share common data.

swipe

Manish Jaiswal

LLD

02

Why Use Flyweight?

When your application creates a large number of similar objects, memory usage can become significant. The Flyweight pattern helps reduce memory footprint by sharing common data among these objects, improving performance and resource efficiency.

swipe

Manish Jaiswal

LLD

03

Where to Use Flyweight?

Use the Flyweight pattern in scenarios with:

- High volume of similar objects
- Objects that share common data
- Applications where memory usage optimization is critical (e.g., graphical applications, game development, text processing)

swipe

Manish Jaiswal

LLD

04

Real-World Example: Text Editors

In a text editor, each character can be a separate object. However, characters share common properties (font, size, color). The Flyweight pattern allows us to share these properties across many character objects.

swipe

Manish Jaiswal

LLD

05

Code: Flyweight class

```
// Flyweight class
public class Character
{
    public char Symbol { get; }
    public string Font { get; }
    public int Size { get; }

    public Character(char symbol, string font, int size)
    {
        Symbol = symbol;
        Font = font;
        Size = size;
    }

    public void Display(int position)
    {
        Console.WriteLine($"{Symbol} (Font: {Font}," +
        " Size: {Size}) at position {position}");
    }
}
```

swipe

Manish Jaiswal

LLD

06

Code: Flyweight Factory

```
// Flyweight Factory
public class CharacterFactory
{
    private readonly Dictionary<char, Character> _characters = new();

    public Character GetCharacter(char symbol, string font, int size)
    {
        if (!_characters.ContainsKey(symbol))
        {
            _characters[symbol] = new Character(symbol, font, size);
        }
        return _characters[symbol];
    }
}
```

swipe

Manish Jaiswal

LLD

07

Code: Usage

```
// Usage
class Program
{
    static void Main()
    {
        CharacterFactory factory = new CharacterFactory();

        string document = "Flyweight";
        int position = 0;

        foreach (char c in document)
        {
            Character character = factory.GetCharacter(c, "Arial", 12);
            character.Display(position++);
        }
    }
}
```

swipe

Manish Jaiswal

LLD

08

Conclusion

The Flyweight Design Pattern is a powerful tool for optimizing memory usage by sharing common data among similar objects. Implementing this pattern can lead to significant performance improvements, especially in memory-intensive applications.

swipe

Manish Jaiswal

LLD

MASTERING

Proxy Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Proxy?

Have you ever encountered a situation where a simple action triggers a complex process behind the scenes? That's the magic of the Proxy Design Pattern!

This pattern introduces a "middleman" object, the Proxy, that acts as a gateway to the actual object (RealSubject). It controls access and can add additional functionalities before reaching the real deal.

swipe

Manish Jaiswal

LLD

02

Why Use Proxy?

Enhanced Security: The Proxy can act as a security guard, filtering requests or adding access control mechanisms before reaching the RealSubject.

Improved Performance: For expensive operations (like database calls), the Proxy can implement caching or lazy loading, optimizing resource usage.

Flexibility: The Proxy allows you to intercept requests and add functionalities like logging or error handling without modifying the RealSubject itself.

swipe

Manish Jaiswal

LLD

03

Where to Use Proxy?

Remote Objects: When working with remote services, a Proxy can handle network communication and potential connection issues.

Lazy Initialization: For resource-intensive objects, a Proxy can delay their creation until absolutely needed.

Access Control: The Proxy can enforce access restrictions based on user permissions or security rules.

swipe

Manish Jaiswal

LLD

04

Real-World Example: Secure File Download

Imagine downloading a file from a server. A Proxy can act as an intermediary:

1. You request the file.
2. The Proxy checks your access permissions.
3. If authorized, the Proxy retrieves the file from the server.
4. The Proxy delivers the file to you.

swipe

Manish Jaiswal

LLD

05

Code

```
interface IFile
{
    void Download();
}

class RealFile : IFile
{
    public void Download()
    {
        // Download logic
    }
}
```

swipe

Manish Jaiswal

LLD

06

Code

```
class ProxyFile : IFile {
  private RealFile realFile;
  private string username;
  private string password;

  public ProxyFile(string username, string password)
  {
    this.username = username;
    this.password = password;
  }

  public void Download()
  {
    if (Authenticate(username, password))
    {
      realFile ??= new RealFile();
      realFile.Download();
    }
    else Console.WriteLine("Access Denied!");
  }

  private bool Authenticate(string username, string password)
  {
    // Implement authentication logic
    return true; // Placeholder for actual logic
  }
}
```

swipe

Manish Jaiswal

LLD

07

Conclusion

The Proxy Design Pattern offers a powerful way to control access, enhance performance, and add flexibility to your system design. By understanding this pattern, you can create more robust and secure applications.

swipe

Manish Jaiswal

LLD

MASTERING

Decorator Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Decorator?

Imagine decorating a cake You start with a base cake (the original object) and then add layers of frosting, sprinkles, and other goodies (decorators) for a customized masterpiece. Similarly, the Decorator pattern lets you dynamically add functionalities to objects without modifying their core structure. It's like wrapping an object in a layer of extra features!

swipe

Manish Jaiswal

LLD

02

Why Use Decorator?

Flexibility: Add features on-the-fly without subclassing madness.

Maintainability: Keeps your base code clean and easy to understand.

Open/Closed Principle: New functionalities can be added without breaking existing code.

swipe

Manish Jaiswal

LLD

03

Where to Use Decorator?

Decorator is a great choice for scenarios like:

Adding Optional Functionalities: Think logging, caching, or security checks.

Building Complex UIs: Compose UIs by dynamically adding borders, shadows, or effects.

Stream Processing: Enhance data streams with compression, encryption, or error handling.

swipe

Manish Jaiswal

LLD

04

Real-World Example: Adding Borders to Pizzas

Let's create a example using pizzas! We'll have a base Pizza class and decorators for adding StuffedCrust and GarlicButterCrust.

swipe

```
1
2  public interface IPizza
3  {
4      string GetDescription();
5      double GetCost();
6  }
7
8  public class Pizza : IPizza
9  {
10     public string GetDescription()
11     {
12         return "Basic Pizza";
13     }
14
15     public double GetCost()
16     {
17         return 8.00;
18     }
19 }
20
21 public abstract class PizzaDecorator : IPizza
22 {
23     private readonly IPizza _pizza;
24
25     public PizzaDecorator(IPizza pizza)
26     {
27         _pizza = pizza;
28     }
29
30     public virtual string GetDescription()
31     {
32         return _pizza.GetDescription();
33     }
34
35     public virtual double GetCost()
36     {
37         return _pizza.GetCost();
38     }
39 }
```

swipe

```
public class StuffedCrustDecorator : PizzaDecorator
{
    public StuffedCrustDecorator(IPizza pizza) : base(pizza) { }

    public override string GetDescription()
    {
        return base.GetDescription() + " + Stuffed Crust";
    }

    public override double GetCost()
    {
        return base.GetCost() + 2.00;
    }
}

public class GarlicButterCrustDecorator : PizzaDecorator
{
    public GarlicButterCrustDecorator(IPizza pizza) : base(pizza) { }

    public override string GetDescription()
    {
        return base.GetDescription() + " + Garlic Butter Crust";
    }

    public override double GetCost()
    {
        return base.GetCost() + 1.50;
    }
}
```

swipe

```
// Usage example
IPizza myPizza = new Pizza();  // Basic pizza
Console.WriteLine(myPizza.GetDescription() + ": $" + myPizza.GetCost());

// Add a stuffed crust
myPizza = new StuffedCrustDecorator(myPizza);
Console.WriteLine(myPizza.GetDescription() + ": $" + myPizza.GetCost());

// Add garlic butter crust on top of the stuffed crust
myPizza = new GarlicButterCrustDecorator(myPizza);
Console.WriteLine(myPizza.GetDescription() + ": $" + myPizza.GetCost());

```

Explanation:

1. We create an instance of the Pizza class (myPizza).
2. We can then decorate it with StuffedCrustDecorator and GarlicButterCrustDecorator in a chain-like manner. Each decorator "wraps" around the previous one, inheriting its behavior and adding its own modification.
3. Finally, we call GetDescription() and GetCost() on the decorated myPizza object to see the dynamically modified results.

swipe

Manish Jaiswal

LLD

MASTERING

Mediator Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Mediator?

The Mediator is a behavioral design pattern that introduces a central communication hub – the mediator object. This mediator object orchestrates interactions between a set of objects, eliminating the need for direct communication between them.

swipe

Manish Jaiswal

LLD

02

Why Use Mediator?

Reduces Complexity: Simplifies object communication by centralizing it in a mediator.

Promotes Loose Coupling: Objects no longer need to reference each other directly.

Improves Maintainability: Makes the system easier to maintain and modify by encapsulating the interaction logic.

swipe

Manish Jaiswal

LLD

03

Where to Use Mediator?

- When a large number of objects need to interact with each other.
- When the communication logic between objects becomes complex.
- When you want to simplify testing individual objects by isolating their interactions.

swipe

Manish Jaiswal

LLD

04

Real-World Example: Chat Room

Imagine a chat application where users can send messages to each other. Without a mediator, each user would need to know every other user. With a mediator, users only need to know the mediator, which handles message distribution.

swipe

Manish Jaiswal

LLD

05

Code: Chat Room (Part 1)

```
1
2  // Mediator interface
3  public interface IChatRoomMediator
4  {
5      void ShowMessage(User user, string message);
6  }
7
8  // Concrete Mediator
9  public class ChatRoom : IChatRoomMediator
10 {
11     public void ShowMessage(User user, string message)
12     {
13         Console.WriteLine(
14           $"{DateTime.Now.ToShortTimeString()} [{user.Name}]: {message}"
15           );
16     }
17 }
```

Explanation: The IChatRoomMediator interface defines the method for showing messages. The ChatRoom class implements this interface and handles the message distribution.

swipe

Manish Jaiswal

LLD

```
// Colleague class
public class User
{
    private IChatRoomMediator _chatRoom;
    public string Name { get; private set; }

    public User(IChatRoomMediator chatRoom, string name)
    {
        _chatRoom = chatRoom;
        Name = name;
    }

    public void Send(string message)
    {
        _chatRoom.ShowMessage(this, message);
    }
}

// Usage
public class Program
{
    public static void Main(string[] args)
    {
        IChatRoomMediator chatRoom = new ChatRoom();

        User user1 = new User(chatRoom, "Alice");
        User user2 = new User(chatRoom, "Bob");

        user1.Send("Hello, Bob!");
        user2.Send("Hi, Alice!");
    }
}
```

swipe

Manish Jaiswal

LLD

07

Explanation

The User class represents participants in the chat room. Each user sends messages through the mediator. The Program class demonstrates how users communicate via the mediator.

swipe

Manish Jaiswal

.NET / C#

MASTERING

Observer Design Pattern

swipe

Manish Jaiswal

.NET / C#

01

What is Observer?

The Observer Design Pattern is a behavioral pattern that defines a one-to-many dependency between objects. In this pattern, when the state of one object changes, all its dependents are notified and updated automatically.This promotes loose coupling between objects, allowing for easier maintenance and scalability.

swipe

Manish Jaiswal

.NET / C#

02

Why Use Observer?

By using the Observer Pattern, you can separate the concerns of the subject (the object being observed) and the observers (the objects interested in its state changes). It provides a flexible way to design systems where changes in one object need to trigger updates in multiple other objects.Observer Pattern promotes reusable and modular code, making it easier to extend and maintain systems.

swipe

Manish Jaiswal

.NET / C#

03

Where to Use Observer?

it proves invaluable in event-driven systems, such as message brokers or reactive programming frameworks, where events trigger actions across multiple components. By leveraging the Observer Pattern in these contexts, developers can build more responsive, modular, and maintainable software solutions.

swipe

Manish Jaiswal

.NET / C#

04

Real-world Application: Stock Market Updates

Stock market serves as the **subject**, while investors act as **observers**. When the price of a stock changes, all subscribed investors receive immediate updates.

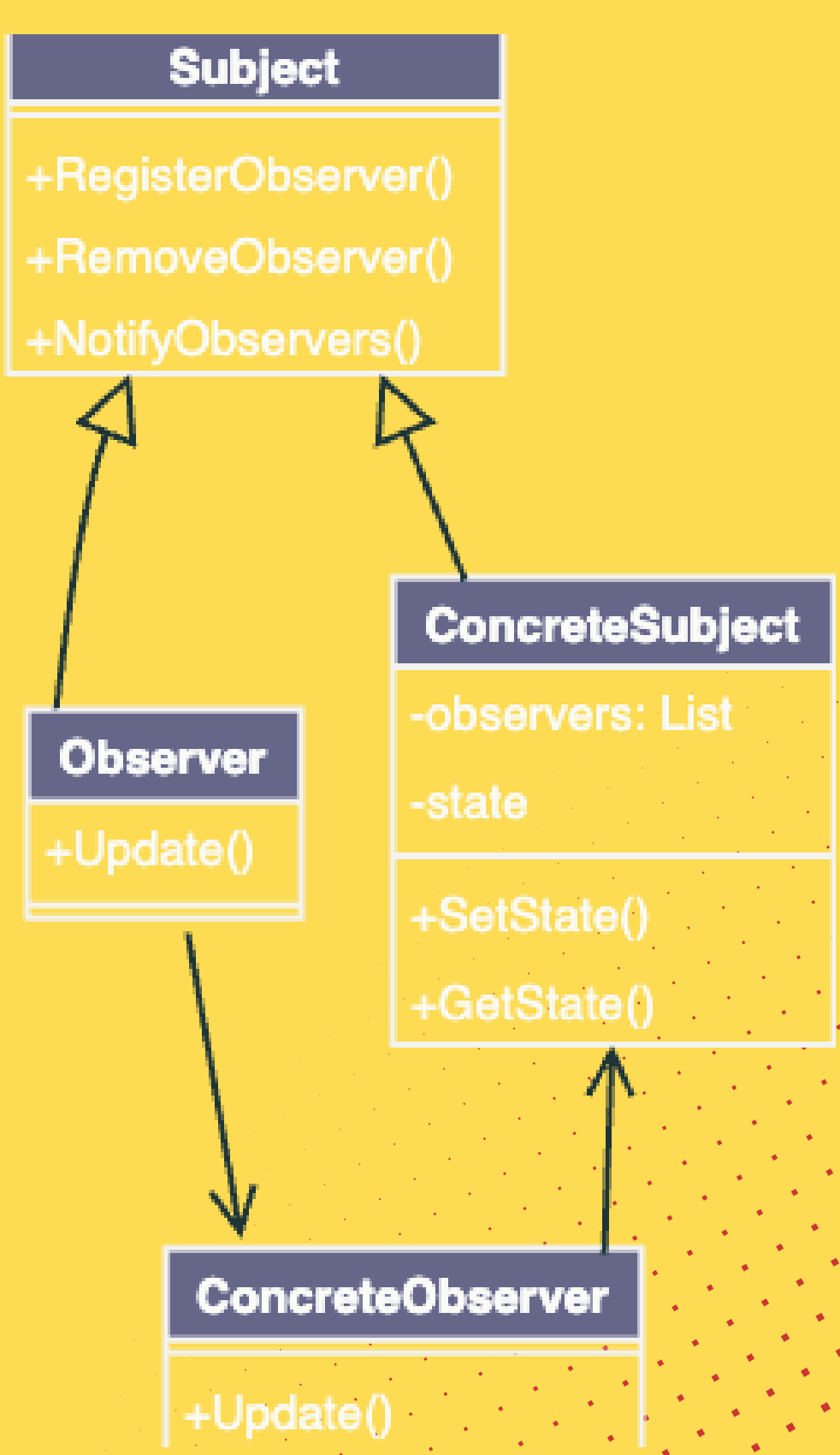
swipe

Manish Jaiswal

.NET / C#

05

Class Diagram



swipe

Manish Jaiswal

.NET / C#

06

Code

```
1
2  // Define Subject Interface
3  public interface ISubject
4  {
5      void RegisterObserver(IObserver observer);
6      void RemoveObserver(IObserver observer);
7      void NotifyObservers();
8  }
9
10 // Define Observer Interface
11 public interface IObserver
12 {
13     void Update(float price);
14 }
```

swipe

Manish Jaiswal

.NET / C#

07

Code Continues

```
// Concrete Subject
public class StockMarket : ISubject
{
    private List<IObserver> observers = new List<IObserver>();
    private float price;

    public void RegisterObserver(IObserver observer)
    {
        observers.Add(observer);
    }

    public void RemoveObserver(IObserver observer)
    {
        observers.Remove(observer);
    }

    public void NotifyObservers()
    {
        foreach (var observer in observers)
        {
            observer.Update(price);
        }
    }

    public void SetPrice(float newPrice)
    {
        price = newPrice;
        NotifyObservers();
    }
}
```

swipe

Manish Jaiswal

.NET / C#

08

Code Continues

```
1  // Concrete Observer
2  public class Investor : IObserver
3  {
4      private string name;
5
6      public Investor(string name)
7      {
8          this.name = name;
9      }
10
11     public void Update(float price)
12     {
13         Console.WriteLine($"Notified {name} : Price changed to {price}");
14     }
15 }
```

swipe

Manish Jaiswal

.NET / C#

09

Code Continues

```
1 // Usage
2 var stockMarket = new StockMarket();
3 var investor1 = new Investor("Alice");
4 var investor2 = new Investor("Bob");
5
6 stockMarket.RegisterObserver(investor1);
7 stockMarket.RegisterObserver(investor2);
8
9 stockMarket.SetPrice(100); // Notifies both investors
```

swipe

Manish Jaiswal

LLD

MASTERING

Strategy Design Pattern

swipe

Manish Jaiswal

LLD

01

What is Strategy?

The Strategy pattern is a behavioral design pattern that lets you define a family of algorithms, encapsulate them in separate classes, and make them interchangeable. This means you can choose which algorithm to use at runtime, offering greater flexibility to your code.

swipe

Manish Jaiswal

LLD

02

Why Use Strategy?

Dynamic Behavior: Change an object's behavior based on context without modifying its core functionality.

Open/Closed Principle: Add new algorithms without modifying existing code.

Maintainable Code: Encapsulate complex algorithms, improving code readability.

Testable Code: Isolate and test individual algorithms for better unit testing.

swipe

Manish Jaiswal

LLD

03

Where to Use Strategy?

Different Algorithms: When you need multiple algorithms for a specific task and want to switch them easily.

Avoiding Conditional Statements: To replace complex conditional statements for algorithm selection.

Runtime Decisions: When the decision of which algorithm to use is made at runtime.

swipe

Manish Jaiswal

LLD

04

Real-World Example: Payment Processing

Imagine an e-commerce platform that needs to process payments using different methods: Credit Card, PayPal, and Bitcoin. Using the Strategy pattern, each payment method is encapsulated in a separate class, allowing the system to switch between them seamlessly.

swipe

Manish Jaiswal

LLD

05

Code: Payment Processor Context

```
1  public class PaymentProcessor
2  {
3      private IPaymentStrategy _paymentStrategy;
4
5      public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)
6      {
7          _paymentStrategy = paymentStrategy;
8      }
9
10     public void ProcessPayment(double amount)
11     {
12         _paymentStrategy.Pay(amount);
13     }
14 }
```

swipe

Manish Jaiswal

LLD

06

Code: Payment Strategy Interface

```
1 public interface IPaymentStrategy
2 {
3     void Pay(double amount);
4 }
```

swipe

Manish Jaiswal

LLD

07

Code: Concrete Payment Strategies

```
1  ublic class CreditCardPayment : IPaymentStrategy
2  {
3      public void Pay(double amount)
4      {
5          // Implementation for credit card payment
6          Console.WriteLine($"Paid {amount} using Credit Card.");
7      }
8  }
9
10 public class PayPalPayment : IPaymentStrategy
11 {
12     public void Pay(double amount)
13     {
14         // Implementation for PayPal payment
15         Console.WriteLine($"Paid {amount} using PayPal.");
16     }
17 }
18
19 public class BitcoinPayment : IPaymentStrategy
20 {
21     public void Pay(double amount)
22     {
23         // Implementation for Bitcoin payment
24         Console.WriteLine($"Paid {amount} using Bitcoin.");
25     }
26 }
```

swipe

Manish Jaiswal

LLD

08

Code: Client Code Example

```
class Program
{
    static void Main(string[] args)
    {
        var paymentProcessor = new PaymentProcessor();

        paymentProcessor.SetPaymentStrategy(new CreditCardPayment());
        paymentProcessor.ProcessPayment(100);

        paymentProcessor.SetPaymentStrategy(new PayPalPayment());
        paymentProcessor.ProcessPayment(200);

        paymentProcessor.SetPaymentStrategy(new BitcoinPayment());
        paymentProcessor.ProcessPayment(300);
    }
}
```

swipe