CHAPTER 3

Structural Design Patterns

Introduction

Structural Design Patterns are a crucial part of software development process that emphasizes on organizing and composing classes and objects to build larger, flexible structures. Picture these patterns akin to LEGO blocks – each block representing an individual class or object. When arranged adeptly, these "blocks" enable the creation of not just visually stunning but also inherently sturdy structures. Much like assembling LEGO pieces, structural design patterns offer solutions to common design dilemmas, fostering code reusability and scalability within intricate systems. Skilful arrangement and organization of these classes and objects empower us to craft a resilient system that champions the separation of concerns, enhancing code readability in the process.

Structure

This chapter will cover the following topics:

* Introduction to Structural Design Patterns
* Adapter pattern
* Bridge pattern
* Composite pattern
* Decorator pattern
* Façade pattern
* Flyweight pattern
* Proxy pattern
* Pattern Selection Tips

Objectives

Upon completing this chapter, you will acquire a mastery of structural design patterns, unlocking the secrets to architecting applications with finesse and flexibility. From demystifying common coding challenges to understanding the usage of these patterns, you will be able to craft code that is modular, scalable, and lucid. Structural design patterns will help you to sculpt code that can withstand the test of time.

Introduction to Structural Design patterns

In our last chapter, we delved into the significance of classes and objects in JavaScript, unravelling their crucial roles in constructing scalable and resilient applications. Recognizing that design patterns in JavaScript hinge on the manipulation of objects and classes to address longstanding challenges, we gained insights into strategies for creating them. This chapter marks the continuation of our journey, shifting focus to another pivotal aspect in alleviating developer concerns – the art of organizing and structuring objects and classes.

Before delving into the intricacies of structural design patterns, it's crucial to grasp the real-world problems that developers routinely encounter, and which can be effectively addressed through the application of these patterns. So, before immersing ourselves in the definition of structural design patterns, let's take a moment to identify some challenges that developers commonly face in their day-to-day work, challenges that can find solutions through the introduction of structural design patterns. Our exploration of solutions for these identified problem statements will unfold later in this chapter, after comprehensively understanding various types of structural design patterns. Stay tuned as we navigate through these patterns and ultimately address the challenges presented.

Problem Statement #1

Let’s look at this code example showcasing a nested object one might commonly encounter while developing an application:

1. const person = {
2. name: 'John',
3. address: {
4. street: '123 Main St',
5. city: 'Seattle',
6. country: 'USA'
7. }
8. };
9. *// Accessing the country property*
10. const country = person.address.country;

Code Issue

The code tightly couples the person and the address object, making it challenging to modify or manage their relationships. For instance, if you wish to change the structure of the address object, it will involve modifying every object and the code in multiple places. This makes code less flexible and harder to maintain.

Problem Statement #2

For our second scenario, let's examine a situation involving two classes that exhibit a notable degree of similarity, yet harbour distinct differences:

1. class BasicCar {
2. constructor() {
3. this.start = function() {
4. console.log('Basic car started.');
5. };
6. }
7. }
8. class LuxuryCar {
9. constructor() {
10. this.start = function() {
11. console.log('Luxury car started.');
12. };
13. this.performLuxuryAction = function() {
14. console.log('Performing luxury action.');
15. };
16. }
17. }
18. *// Usage*
19. const car = new LuxuryCar();
20. car.start();
21. car.performLuxuryAction(); *// Throws an error if used with BasicCar*

Code Issue

Since we looked at the creational design patterns in the preceding chapter, we might have noticed the glaring mistake of not having a link between the two classes BasicCar and LuxuryCar. Lack of a common interface makes it challenging to use different car types interchangeably.

Problem Statement #3

In our final example, we will be looking at an example related to inheritance. Let’s have a closer look at the problem that might arise due to a lack of organization:

1. class Animal {
2. constructor(name) {
3. this.name = name;
4. }
5. eat() {
6. console.log(`${this.name} is eating.`);
7. }
8. }
9. class Bird extends Animal {
10. fly() {
11. console.log(`${this.name} is flying.`);
12. }
13. }
14. *// Usage*
15. const sparrow = new Bird('Sparrow');
16. sparrow.eat();

Code Issue

The provided code snippet seems harmless at first glance. It introduces a class Bird that extends the Animal class. Through this extension, Bird inherits both the name property and the eat method from Animal, subsequently invoking the eat function. However, a potential pitfall lies in the fact that the eat function exclusively belongs to the Animal class.

In the dynamic landscape of a growing application, structural modifications to the parent class Animal are conceivable. This could involve drastic changes, including the removal of the eat function in a large-scale application. If such alterations occur, all classes inheriting the eat property would encounter failures, potentially leading to application crashes that prove challenging to debug. Furthermore, as the hierarchy expands, it exacerbates code readability issues. The inherent tight coupling of classes also hinders their reuse, adding another layer to the challenge.

Structural Design Patterns

In a nutshell, structural design patterns help solve problems highlighted in the earlier sections by enhancing code organization. These patterns revolve around the concept of organizing and composing classes and objects to create robust and flexible software architectures. They provide solutions to problems commonly faced by developers and at the same time, these patterns promote reusability of code, maintainability and scalability. Let's briefly explore some key attributes of structural design patterns:

Organization of Classes

Structural Design patterns help in organizing classes and objects in a way that enhances code clarity and separation of concerns. This aids in managing the complexity of large codebases by providing a structured and coherent architecture.

Object Composition

These patterns emphasize the composition of objects to create larger, more complex structures. By composing objects instead of relying on class inheritance, we can create a flexible and dynamic relationship between components.

Code Reusability

One of the biggest advantage and goal of the structural design pattern is code reusability. Structural design patterns like Adapter, Composite, and Decorator, lean on creating reusable components that can be seamlessly integrated into different parts of the application.

Flexibility and Adaptability

One of the major advantages of using the structural design pattern is that it provides mechanisms to alter or extend the structure of existing objects and classes without modifying the source code. This makes the code adaptable to changing requirements.

Common Design Problems

Structural design patterns address common design problems such as managing object relationships, handling interface mismatches, and organizing code in a way that is conducive to both current and future requirements.

In summary, true to their name, structural design patterns play a pivotal role in imparting structure and organization to code, ultimately fortifying the system and enabling it to adapt seamlessly to the evolving requirements of the application. In the forthcoming sections, we will delve into various types of structural design patterns, each offering unique solutions to diverse problems.

Adapter design pattern

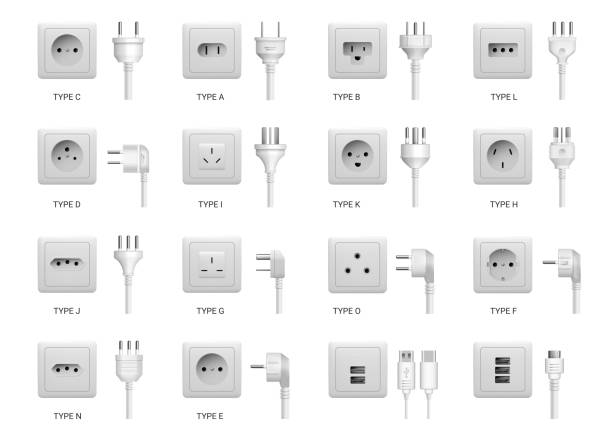
Let’s kickstart our journey to understand structural design patterns by exploring the very first pattern from this category – Adapter pattern. In simple words, Adapter pattern allows objects with incompatible interfaces to collaborate.

Problem Scenario

Let’s understand the problems solved by the Adapter pattern with the following two examples. Our first example has no reference to any coding application.

Scenario #1

In the context of *Figure 3.1*, which illustrates various electronic devices used in India, it is apparent that the chargers for devices such as the MacBook, Android phones, and smartwatches differ significantly. For instance, the MacBook charger typically utilizes a USB-C connector, while Android chargers may have micro USB connectors. The smartwatch charger, on the other hand, is specifically designed for USB Type-B ports.



**Figure 3.1:** Different types of ports and sockets used by different countries

When traveling to countries like the UK or various European nations, a challenge arises due to the distinct power outlets. The power outlets in the UK and Europe differ from those in India, and this incompatibility may render the chargers brought from home unusable. In such instances, a practical solution is required – a device commonly known as a travel adapter or plug converter. This device should be capable of accepting Indian-style power outlets and converting them to the formats required in the UK or Europe.

Scenario #2

In our second scenario, let's explore the realm of a **Content Management System** (**CMS**) application tailored for managing articles and blog posts. For this illustration, let's assume our application exclusively supports the storage and retrieval of data in the relational database format, exemplified by SQL tables.

The challenge arises when a decision is made to enhance our application by incorporating a full-text search engine. After exhaustive research, a third-party search engine library emerges as the ideal candidate due to its efficiency and seamless compatibility with our requirements. However, a significant hurdle surfaces – this library exclusively operates on document-based storage, such as MongoDB, and necessitates content in JSON format.

Now confronted with a critical dilemma, the question looms: should we undergo substantial modifications to our CMS code to align with this third-party library, or should we opt for a costly and less efficient alternative that fails to fully meet our needs?

In both of the scenarios outlined above, a common need emerges for a device or tool capable of seamlessly converting or facilitating the adaptation between disparate devices or codebases. Put simply, what's required is an adapter. The Adapter structural design pattern is precisely crafted to fulfil this purpose.

Implementation

The implementation of the Adapter design pattern can be broken down into five distinct steps, culminating in clear instructions on its utilization within an actual client code:

Step 1: Define the Existing Class

In this step, you define the existing class that has a specific interface. This class represents some functionality that you want to use in your application but does not conform to the desired interface.

Step 2: Create the Target Interface

Here, you define the target interface that your client code expects. This interface represents the set of methods that your client code will use. It's what you want your existing class to conform to.

Step 3: Implement the Adapter

Next, we craft the adapter that functions as a bridge connecting the target interface and the existing class. In this step, the adapter class is formed by extending the target interface and accepting an instance of the existing class as a parameter. Subsequently, a new method is introduced within the adapter class to execute the pertinent method of the existing class.

Step 4: Client Code

In this step, we define a function that utilizes the method that you have defined in the adapter implementation in step 3. This function will receive an instance of the adapter as a parameter and implement the new method that we have defined inside it.

Step 5: Usage

In the concluding step, we bring the implementation to fruition by connecting the existing class with the target interface using the adapter. This involves instantiating the existing class, creating an adapter, passing the instance of the existing class to it, and ultimately utilizing the method we defined in the previous step to establish the connection between the two.

Code Example

We’ve broken down the implementation of the Adapter pattern into five distinct steps in the previous section. Now, let’s utilize the example of an authentication system to realise the steps that we have seen in the previous section. In this example, we have a legacy authentication system that accepts a username and custom token. However, modern architecture calls for a username and a password to authenticate a user. Let’s see how we can implement the Adapter pattern to solve this issue:

Step 1: Define the Existing Class

The very first thing that we need to do is to define the existing class. From the preceding section, we know that the term *existing class* stands for the code that is not compatible with the change and that it would require the assistance of an adapter to achieve that. Let’s go ahead and define our existing class:

1. class LegacyAuthenticator {
2. authenticate(username, customToken) {
3. *// Legacy authentication logic*
4. return "Legacy Authentication Successful";
5. }
6. }

In the above example, we have defined the class LegacyAuthenticator. This class possesses a single method authenticate that accepts two parameters – a username and a customToken. On execution, it returns the statement “Legacy Authentication Successful”.

Step 2: Create the Target Interface

Now, let’s consider that we have decided to adopt a modern form of authentication, like OAuth. A modern authentication would probably require a username and a password, so we create an interface like this:

1. class ModernAuthenticator {
2. login(username, password) {
3. *// The client expects this method*
4. }
5. }

In the above code example, we have defined a class – ModernAuthenticator. In this class, we have a method login that accepts username and password. This is the desired interface that follows the modern authentication method.

Step 3: Implement the Adapter

Up to this point, we've outlined the original class/interface adhering to the proprietary method of authentication and a modern interface specifying a newer authentication method. Now, the pivotal step is to forge the bridge that seamlessly connects the two. Let's proceed with the creation of the adapter:

1. class LegacyToModernAuthAdapter extends ModernAuthenticator {
2. constructor(legacyAuthenticator) {
3. super();
4. this.legacyAuthenticator = legacyAuthenticator;
5. }
6. login(username, password) {
7. *// Implement the login method using the legacy authenticator*
8. return this.legacyAuthenticator.authenticate(username, password);
9. }
10. }

In the above code example, we have created an adapter, aptly titled LegacyToModernAuthAdapter. It's crucial to ensure that this class serves as a seamless bridge between the two interfaces. The new code will align with contemporary standards while internally invoking the legacy system for authentication. For a clearer understanding, let's refer to *Figure 3.2*, based on the example from Scenario 1 in the Problem Scenario segment of the Adapter design pattern.

A hand holding a plug

Description automatically generated

**Figure 3.2:** An adapter acts as a bridge connecting two dissimilar items of the same type

In *Figure 3.2*, on the left, we have a charger with a connector tailored for American users. On the extreme right, there's an outlet port in a Parisian hotel in Europe. To address this discrepancy, an adapter, depicted in the middle, is necessary. This adapter takes the Indian connector as input and provides its own connector, adaptable to European ports, as output. Upon connecting the device through this adapter, it seamlessly accommodates both the original and newer formats, ensuring the correct functioning of the device.

Likewise, we can draw a parallel between the LegacyAuthenticator and the European outlet port, while the Indian charger corresponds to the ModernAuthenticator. To enable the Indian charger to seamlessly operate on European ports, we require the adapter LegacyToModernAuthAdapter. This adapter must be capable of interfacing with the Indian charger, accepting its connector, and delivering the charge to the European outlet.

In a similar vein, the LegacyToModernAuthAdapter must extend the ModernAuthenticator to gain access to its login method. Simultaneously, it needs to connect to the European outlet port in its original format, necessitating an internal connection or execution of the LegacyAuthenticator's authenticate method. This intricate dance ensures the smooth compatibility and operation of the ModernAuthenticator with the legacy LegacyAuthenticator.

Step 4: Client Code

Now, let’s create a function that will in essence call the method of the adapter. Let’s first look at the code example before understanding what transpires in it:

1. function handleAuthentication(authenticator) {
2. console.log(authenticator.login("user123", "password123"));
3. }

In the above code example, we have created a function handleAuthentication that accepts an authenticator as a parameter. In essence, on the invocation of this function, it expects an instance of the adapter. Hence, as you can see on the next line, we will be calling the login method of the authenticator, i.e., the adapter’s login method. This method will be useful in completing the bond required for transitioning from the target interface to the existing class.

Step 5: Usage

In the ultimate step, we consolidate all the information gathered thus far to bring the adapter method to fruition:

1. *// Instantiate the existing class (legacy authenticator)*
2. const oldAuthenticator = new LegacyAuthenticator();
3. *// Create an adapter and pass the existing class instance to it*
4. const adaptedAuthenticator = new LegacyToModernAuthAdapter(oldAuthenticator);
5. *// Use the client code to handle authentication with the adapted authenticator*
6. handleAuthentication(adaptedAuthenticator);

As we can see in the above code example, we have divided the final implementation into three parts:

1. We first create an instance of the old authentication system and store it in the oldAuthenticator variable.
2. Next, we create an instance of the adapter. As explained in step 3, it requires an instance of the existing class as a parameter. Therefore, in our case we pass the oldAuthenticator variable to the constructor of the adapter by passing it in as an argument. We, then, store the instance of the adapter in the variable adapterAuthenticator.
3. In the conclusive step, we invoke the method delineated in step 4 to log into the system as a modern authenticator. As observed in the preceding code, this operation executes the original authenticator within its context, and we consequently receive the resulting outcome.

With this completion, the Adapter method has been successfully implemented. This method allows a new code format to seamlessly interact with an existing code structure without necessitating any alterations to the original codebase. The adapter serves as a vital intermediary, facilitating cohesion between the modern and legacy components, ultimately enabling them to work together harmoniously.

Applications

From the preceding sections, we have learnt that the adapter pattern is extremely useful for scenarios where you want to bring about new changes to your code-base without changing the original code written. This benefit naturally finds a lot of applications in modern JavaScript programming. Let’s look at some of those applications:

Integration Legacy Code

As we have already seen in the Code Example section, the Adapter pattern works like a charm for handling legacy code that has a different interface to modern JavaScript applications.

Third-party API Integration

More often than not, we need to rely on third-party APIs to solve a minor chunk of our applications. We may often encounter a situation where we need to integrate several third-party APIs or libraries to create a unified interface for our application. This can be handled by the Adapter pattern without affecting the original code base.

Microservices Integration

A microservices architecture is often the sound solution for an application that is implemented on a scale. In essence, the various major chunks of the application’s backend and frontend services are divided into individual services which are then seamlessly connected to each other behind the scenes to display a unified UI to the user. In such a scenario, Adapter pattern can help adapt communication between services with different protocols or message formats.

Version Upgrades

When upgrading a library or framework to a new version that has breaking changes in its interface, and we wish to minimize the impact on our existing code, the Adapter pattern can help minimize the damage.

Internationalization

For applications that are built on a global scale, internationalization or the support for multiple international languages and/or currency is a must-have feature. Adapter pattern can help adapt our application to support multiple languages and support the need to change the behaviour of certain components based on the selected language without modifying their source code.

Pros and Cons

Here are some of the advantages of using Adapter patterns in your code:

* **Single Responsibility Principle (SRP):** The Adapter pattern resolves the only problem of separating the interface or the logic for mapping legacy code to new code from the original source code.
* **Open/Close Principle:** We can add new adapters to the code without breaking or modifying the original source code.

However, the introduction of Adapter pattern also signifies adding several new interfaces and classes that might increase the overall complexity of the code. It is useful to check if the changes required are minor or major with an eye on posterity before implementing this pattern.

Bridge design pattern

The Bridge design pattern is a structural design pattern that is separates the abstraction part of the object from its implementation, allowing them to vary independently. It allows splitting large classes or closely related classes into two parts – abstraction and its implementation, allowing them to be developed independently of each other.

Problem Scenario

The Bridge design pattern proves instrumental in resolving issues where the abstraction of a class needs separation from its implementation, allowing them to function independently. Let's delve into two scenarios that mirror problematic situations, demonstrating how the bridge design pattern can effectively address and solve these challenges:

Scenario #1

Let's explore another scenario using the example of geometric shapes and their colours. Initially, we have a class Shape with two subclasses – Square and Circle. Now, envision the need to expand these classes to encompass colours – Blue and Red. This expansion results in a total of four classes – BlueSquare, BlueCircle, RedSquare, and RedCircle. However, the complexity intensifies when the requirement arises to introduce two additional shapes and two more colours. This leads to a proliferation of subclasses, resulting in a more intricate and less robust code structure.

Scenario #2

For our second scenario, let’s consider the example of a UI toolkit. A UI toolkit is a library that provides common UI elements like buttons, windows, text fields, and so on, to allow developers to create UI for applications. Now, the challenge lies in supporting multiple operating systems like MacOS, Windows, and Linux. Each operating system has its own native windowing system and UI conventions. Application developers want to create cross-platform applications without worrying about the intricacies of each operating system's UI implementation. Once again, the developers should not require modifying the application code.

The common thread between the above two scenarios is that in both the cases, there is a need for the abstraction (colour and UI toolkit) to be independent of the specific details of various platforms (shape and operating system). As new elements or structures are introduced, there should be a way to seamlessly integrate and support them without modifying the existing codebase.

Implementation

Let's deconstruct the implementation of the Bridge design pattern into discrete steps. While these steps may initially appear complex or overwhelming, in the following section, we will explore the same steps with a relevant example to enhance our understanding of the implementation of this design pattern.

Step 1: Define the Implementor Interface

In this step, the focus is on defining the interface or an abstract class for the implementor. When we refer to the implementor, we are addressing the underlying system with its various implementations. During this phase, the objective is to outline the functions that will be implemented by the concrete implementors, which we will be creating in the subsequent step.

Step 2: Create Concrete Implementors

Now, the next step involves creating several concrete classes, each dedicated to implementing the operations declared in the implementor interface. Each of these concrete implementors serves to represent a distinct and specific implementation of the functions outlined in the implementor interface.

Step 3: Define the Abstraction

Create an abstraction class that contains a reference to the implementor. The abstraction class declares higher-level operations that use the operations declared in the implementor interface. This separation allows the abstraction to be independent of the specific details of the implementations.

Step 4: Create Refined Abstractions

Upon preparing the concrete implementors and the abstraction, the final piece of the puzzle comes together with the creation of refined abstractions. These refined abstractions are classes designed to implement the abstraction for specific use cases, utilizing the operations defined in the implementor interface to execute their tasks. Additionally, these refined abstractions have the flexibility to incorporate additional properties and methods beyond those inherited from the implementor interface.

Step 5: Client Code

In the client code, we will create instances of the concrete implementors and pass them to the abstraction instances. The client interacts with the abstraction, and the abstraction delegates the implementation-specific details to the concrete implementor at runtime. This allows the client to work with the abstraction without being concerned about the specific implementation details, promoting flexibility and maintainability.

Now, let’s realise this by utilising a relevant example to implement the above steps.

Code Example

Without further ado, let’s deep dive into understanding the implementation of the Bridge design pattern with the help of a code example. Let’s consider the example of a messaging system that needs to send messages via different channels (e.g. SMS, email, Slack) while keeping the message content and sending logic separate.

Step 1: Define the Implementor Interface

In accordance with the steps outlined in the preceding segment, the initial step involves establishing the foundation, namely, the implementor interface. In our example, the implementor, or the core essence, is a message sender. Subsequently, the variations of this class's implementation will be employed for sending messages across different channels.

1. *// Implementor interface*
2. class MessageSender {
3. sendMessage(message, to) {
4. *// Implemented by concrete implementors*
5. }
6. }

Here, we have created an abstract class MessageSender with a singular method, sendMessage that accepts two parameters message and to. In the subsequent steps, this method will play a central role in guiding the implementation of the Bridge design pattern.

Step 2: Create Concrete Implementors

With the establishment of the designated abstract class defining our implementor, or the foundational structure upon which an array of implementors (message senders) will be built, let's proceed to create the three concrete implementors. These concrete classes will extend the implementor formulated in step 1.

1. *// Concrete Implementor A*
2. class EmailSender extends MessageSender {
3. sendMessage(message, to) {
4. console.log(`Sending email: "${message}" to ${to}`);
5. }
6. }
7. *// Concrete Implementor B*
8. class SMSSender extends MessageSender {
9. sendMessage(message, to) {
10. console.log(`Sending SMS: "${message}" to ${to}`);
11. }
12. }
13. *// Concrete Implementor C*
14. class SlackSender extends MessageSender {
15. sendMessage(message, to) {
16. console.log(`Sending Slack message: "${message}" to ${to}`);
17. }
18. }

As we see in the above code example, we have now created three versions of our message senders by extending our original implementor, MessageSender. These three versions or the concrete implementors have extended and used the original method sendMessage defined in MessageSender to create their unique versions of sending messages.

Step 3: Define the Abstraction

Drawing reference from the example of Shape used in Scenario #1 from the Problem Scenarios section of the Bridge design pattern, we've successfully tackled the implementation aspect—the shapes, represented by the circle and square. Now, our focus shifts to the abstraction aspect—the colours—of the problem scenario. This implies that having created the message senders, our next task is to construct the actual messages. To achieve this, we initiate the process by creating an abstract class for the variations of messages.

1. *// Abstraction*
2. class Message {
3. constructor(sender) {
4. this.sender = sender;
5. }
6. send(message, to) {
7. *// Delegated to the concrete implementor*
8. this.sender.sendMessage(message, to);
9. }
10. }

In the provided code example, we have established the abstract class for our abstraction—Message. As evident in the code, we've defined a constructor to accept a sender, which is subsequently assigned to the sender property of the class. This parameter corresponds to the specific variation of the sender needed for its message equivalent (EmailSender to its equivalent email message). Additionally, we've outlined the send method, which takes two arguments—message and to. These parameters align with the requirements of the sender concrete implementors we defined in the previous example. The logical next step involves executing the sendMessage method of the sender received while creating a new instance of this class. We have done that inside the send method defined for this abstract class.

Step 4: Create Refined Abstractions

Our next logical step is to create the individual variants of our abstraction. In other words, we need to generate distinct versions of messages that align with the concrete implementors of senders crafted in the preceding steps. Let's proceed to create the three refined abstractions of our abstraction, Message.

1. *// Refined Abstraction A*
2. class EmailMessage extends Message {
3. constructor(sender) {
4. super(sender);
5. }
6. }
7. *// Refined Abstraction B*
8. class SMSMessage extends Message {
9. constructor(sender) {
10. super(sender);
11. }
12. }
13. *// Refined Abstraction C (You can add more refined abstractions as needed)*
14. class SlackMessage extends Message {
15. constructor(sender) {
16. super(sender);
17. }
18. }

With the illustrated code example above, we now possess three message senders and their corresponding messages. It's noteworthy that all three refined abstractions—EmailMessage, SMSMessage, and SlackMessage—receive a sender in their constructor. In the final step of this implementation, we'll explore how to appropriately utilize this sender to implement the Bridge design pattern.

Step 5: Client Code

In the ultimate step of our implementation, we seamlessly integrate both the implementors and the abstractions to work in harmony, ultimately achieving the end goal of delivering successful messages on the appropriate platforms.

1. *// Example usage*
2. const emailSender = new EmailSender();
3. const smsSender = new SMSSender();
4. const slackSender = new SlackSender();
5. const emailMessage = new EmailMessage(emailSender);
6. const smsMessage = new SMSMessage(smsSender);
7. const slackMessage = new SlackMessage(slackSender);
8. emailMessage.send("Hello via email!", "john@example.com");
9. smsMessage.send("Hello via SMS!", "+123456789");
10. slackMessage.send("Hello via Slack!", "John Doe");

In the final implementation, we initiated the process by creating instances of the concrete implementors, storing them in their respective variables. Following that, instances of the refined abstractions were crafted, with the corresponding senders passed as arguments. Finally, the send method from each refined abstraction was utilized to dispatch the message to the respective platforms.

This successful implementation demonstrates the effective separation of the implementation part from the abstraction part, allowing both entities to function independently of each other. Leveraging the Bridge design pattern, the addition of various platforms and the modification of their message styles can be executed individually without necessitating any alterations to the original code base.

Applications

The Bridge pattern proves highly advantageous in scenarios where the abstraction and the implementation of your code must operate independently without mutually impacting each other. Here are some instances where this pattern excels:

UI Framework for cross-platform applications

When creating a user interface framework for cross-platform applications (e.g., web, mobile, desktop), the Bridge pattern can be employed to separate the UI components from the underlying platform-specific implementations.

Database abstraction layer

When building a database abstraction layer, especially in cases where your application supports multiple databases (e.g., MySQL, PostgreSQL, MongoDB), the Bridge pattern can be used to separate the database operations from the specific database implementations.

Messaging System for communication channels

As we have seen from the example used in the Code Example, if you have a messaging system that sends messages through different channels (e.g., email, SMS, push notifications), the Bridge pattern can help separate the message abstraction from the specific implementations for each messaging channel.

Network communication framework

When designing a network communication framework that needs to support different protocols (e.g., HTTP, WebSocket, TCP), the Bridge pattern can be applied to separate the abstraction of network communication from the specific protocol implementations.

Pros and Cons

Let’s take a look at some of the advantages of using the Bridge pattern:

* One of the biggest advantage of using the Bridge pattern is that it allows us to build cross-platform and platform-independent classes and applications.
* The client code works with the high-level abstractions. At no point, it is exposed to the implementation details of the code.
* It follows the Single Responsibility Principle by solving only one crucial problem of separating the abstractions and implementations from each other.
* It also follows the Open/Closed principle by allowing addition of several new elements without impacting the original code.

As it is with the design patterns that we have seen so far, a potential disadvantage of using the Bridge pattern is that it can increase the complexity of the code if not used correctly.

Composite design pattern

The Composite design pattern is a structural pattern that allows you to deal with hierarchical structures such as tree-like data structures or graphical user interfaces. It lets you compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.

Problem Scenario

The Composite design pattern proves invaluable in scenarios such as managing a file system in an operating system, where there are two distinct objects: Files and Folders. A Folder can contain multiple Files as well as other Folders, creating a hierarchical structure.

Now, imagine we are tasked with calculating the total size of a Folder, which includes the sizes of all nested Files and Folders within it.

JavaScript developers grappling with this task encounter several hurdles. Firstly, dealing with different types of items becomes a concern, as the Folder class must manage both files and other folders. Designing the composite structure to accommodate diverse item types while upholding a consistent interface demands thoughtful consideration. Additionally, the complexity is heightened by the fact that the Folder class also handles other folders, introducing the challenge of circular references. Implementing checks to prevent infinite loops during operations becomes a crucial aspect of the solution.

This overview only scratches the surface, and further exploration into the implementation of this coding problem reveals additional intricacies.

Implementation

The implementation of the Composite design pattern can be broken down into four distinct steps:

Step 1: Define the Component interface or the base class

In a tree structure, a leaf node is the node that does not have any child nodes. To think of it in the context that we have seen above, a leaf node could be a File which cannot have any child element inside it, unlike a Folder which can possess another Folder. However, you have to keep in mind that an empty Folder, i.e., a Folder that has no more child elements inside it can also be termed as a leaf node. In the very step to implement the Composite design pattern, we create the interface or a base class that declares common operations for both leaf and composite components.

Step 2: Implement Leaf Class (Individual components)

In this step, we will create a class or interface for the leaf classes, i.e., individual components that do not have any child nodes. This class will be extending the component interface that we have created in the first step.

Step 3: Implement Composite Class

Now, we create a class for the composite containers (containers of other components). This class will also extend the component interface that we had created in the first step. Additionally, this class should contain a collection to store child components.

Step 4: Client Code

In the final step of the implementation, we use the created class to compose and work with the hierarchical structure.

Now, let's solidify our understanding of the implementation steps for the Composite design pattern with a concrete code example to reinforce the topic.

Code Example

Let's put the steps for implementing the Composite pattern, as discussed in the previous section, into action. We will use the same example of a file system that we have discussed in the Problem Scenario segment for Composite pattern.

Step 1: Define the Component interface or the base class

In the very first step, we need to define the root component interface which will be extended by both the leaf nodes and the composite nodes. In our case, for the sake of simplicity, let’s consider only files to be the leaf nodes while the folders can be considered as a composite nodes. Therefore, in our scenario, this component interface will be extended by both the leaf node (File) and the composite node (Folder). Let’s look at the code to implement that:

1. class Component {
2. constructor(name) {
3. this.name = name;
4. }
5. *// Declare common operations*
6. operation() {
7. throw new Error("Operation must be implemented by subclasses");
8. }
9. }

In the provided code example, we've established the class Component. Its constructor accepts a name argument, which is then assigned to the class property name. This class acts as the foundational element, and to reinforce its role, we've included the operation method, signalling that it must be implemented by the subclasses.

Step 2: Implement Leaf Class (Individual components)

As previously mentioned, we will designate only the File class as our leaf node, implying that the File class will not have any child elements. In this step, let’s create the class for a leaf node.

1. class File extends Component {
2. constructor(name, size) {
3. super(name);
4. this.size = size;
5. }
6. operation() {
7. console.log(`File: ${this.name}, Size: ${this.size} KB`);
8. }
9. }

In the given example, we've introduced the File class, extending the Component class. Leveraging the super method, we inherit the name property from the Component class. Furthermore, an additional property named size is introduced, accepted as an argument in the constructor. To tailor the behaviour for the File class, we override the operation method defined in the Component class.

Step 3: Implement Composite Class

Following the leaf node implementation, our next step involves creating an interface or a class for our composite elements. In this case, we'll designate folders, which can contain other folders or files, as our composite elements. Let's proceed by defining the class for it.

1. class Folder extends Component {
2. constructor(name) {
3. super(name);
4. this.children = [];
5. }
6. *// Add a child component*
7. add(child) {
8. this.children.push(child);
9. }
10. *// Remove a child component*
11. remove(child) {
12. const index = this.children.indexOf(child);
13. if (index !== -1) {
14. this.children.splice(index, 1);
15. }
16. }
17. *// Implement operation to traverse and perform actions on children*
18. operation() {
19. console.log(`Folder: ${this.name}`);
20. for (const child of this.children) {
21. child.operation();
22. }
23. }
24. }

In this segment, we've defined the Folder class for our composite elements, extending the Component class. Similar to the leaf node interface, we inherit the name property using the super method. Since folders can contain other folders or files, a new property called children is introduced, initialized with an empty array.

Given that the Folder class can now contain children, we formulate methods to manage these children. First, we create two dedicated methods, add and remove, to facilitate the addition and removal of children in the **children** array. Additionally, we override the operation method to print the name of the folder and then traverse through each of its children, invoking their operation method to print their names and sizes, as defined in the File class.

Step 4: Client Code

In the final step of the implementation, let's first examine the code example before delving into how to execute the last piece of the puzzle:

1. const rootDirectory = new Folder("Root");
2. const file1 = new File("Document.txt", 200);
3. const file2 = new File("Image.jpg", 500);
4. const subDirectory = new Folder("SubDirectory");
5. const file3 = new File("Code.js", 300);
6. subDirectory.add(file3);
7. rootDirectory.add(file1);
8. rootDirectory.add(file2);
9. rootDirectory.add(subDirectory);
10. *// Perform operations on the composite structure*
11. rootDirectory.operation();

For the final execution, we begin by creating a variable rootDirectory and store within it an instance of the Folder class with the argument "Root." This instance serves as our root directory. Subsequently, we create two more variables, storing instances of two files—file1 and file2. Following that, another folder is created, stored in subDirectory, and a final file, file3, is created.

Next, we add file3 to the subDirectory and add both file1 and file2 to the rootDirectory. As the concluding step, we add the subDirectory to our rootDirectory and invoke its operation method. When you run this code, the following values will be displayed on your console:

1. Directory: Root
2. File: Document.txt, Size: 200 KB
3. File: Image.jpg, Size: 500 KB
4. Directory: SubDirectory
5. File: Code.js, Size: 300 KB

Indeed, by invoking the operation method of the rootDirectory, we successfully obtain the entire folder structure, including all the nested files and other folders, printed in our console. This exemplifies the Composite design pattern in action, allowing us to seamlessly work with both individual leaf nodes (files) and composite elements (folders) in a unified manner.

Applications

The Composite Design Pattern is particular useful in JavaScript for managing hierarchical structures where individual objects and compositions of objects need to be treated uniformly. Here are some scenarios where the Composite pattern is highly effective:

GUI Components

Graphical user interfaces often involve a hierarchy of components like panels, buttons, and text fields. The Composite pattern enables uniform treatment of individual GUI elements and complex layouts, simplifying their creation and manipulation..

Tree Structures

This pattern is ideal for scenarios involving tree-like structures such as organizational hierarchies, family trees, or nested categories. It provides a consistent way to traverse and manipulate these structures while reducing complexity.

Document Object Model Manipulation

The DOM in web development can be represented as a composite structure. Using the Composite pattern allows developers to perform operations on individual HTML elements or complex nested document structures in a consistent manner.

File System Operations

In file systems, directories can contain both files and subdirectories. The Composite pattern enables uniform operations on the entire structure, such as calculating directory sizes or applying bulk actions.

Menu Systems

Nested menus, such as those in navigation systems, benefit from this pattern. Each menu item can act as a leaf (individual menu option) or a composite (submenu with options), allowing for seamless management of the hierarchy.

E-commerce shopping cart

In e-commerce applications, shopping carts often contain individual products and product bundles (composed of multiple products). The Composite pattern simplifies operations like calculating total prices or applying discounts uniformly across all items.

Pros and Cons

Let’s look at some of the benefits of using Composite pattern in a JavaScript code:

* It is easier to work with complex tree structures, as we have seen in the implementation examples above.
* Additionally, it follows the Open/Closed Principle, where new elements can be added to the system without breaking the original code.

Unfortunately, the biggest con of using this pattern is its difficulty in providing a common interface for classes whose functionality differs too much. Overgeneralizing the central component could lead to more complications than required.

Decorator design pattern

The Decorator design pattern is a structural design pattern that allows behaviours to be added to an object, either statically or dynamically, without affecting the behaviour of the other objects from the same class.

Problem Scenario

Upon reviewing the description above, one might initially consider solving this problem using one of the design patterns encountered so far, or even resorting to simple inheritance. However, the intricacies of the problem reveal a layer of complexity that cannot be effectively addressed by the patterns we've explored thus far. To comprehend the problem scenario that can be effectively tackled by a Decorator design pattern, let's explore a real-life analogy.

Consider a car manufacturing company that produces standard cars equipped with a set of basic features, including standard seats, a dashboard, and an air-conditioning system. However, after a customer purchases a car, they often desire to customize it according to their preferences. Some may opt for leather seats to enhance comfort, while others might want to install a sunroof or upgrade to a superior sound system.

The challenge lies in enabling customers to customize their cars with optional features such as leather seats, a sunroof, and an advanced sound system without resorting to creating an exhaustive set of subclasses for every conceivable combination of features. For example, some customers may desire only leather seats, while others might prefer a combination of leather seats and an advanced sound system. This leads to numerous permutations and combinations, and as more services and additional features are introduced, the system becomes more error-prone and increasingly complex. We want a flexible solution that allows customers to pick and choose additional features dynamically.

Implementation

Let us break down the implementation of the Decorator design pattern into distinct steps:

Step 1: Define the component interface

Let’s start with the atomic task of building the interface of the base class, i.e., the class that will be extended by all the concrete components to add their respective versions or set of features.

Step 2: Create concrete component

Now, we implement a concrete component that extends our base component interface. This will be the basic object that we want to decorate.

Step 3: Create the decorator class

To incorporate decorators or specific sets of features into our base component, we initially need to establish an interface that facilitates this extension. Consequently, we create the decorator class that extends the base component interface. However, this class holds a reference to the concrete component object, enabling it to wrap and augment the behaviour of that object.

Step 4: Create concrete decorators

With the interface in place for creating decorators, our next step involves crafting concrete decorators by extending the decorator class. These classes have the capability to add or modify behaviour while ensuring compatibility with the base component interface.

Step 5: Usage

The ultimate step is to create instances of the concrete component and decorators. We need to compose them in a way that builds the desired combination of behaviour.

As done previously, let's concretize these implementation steps through a code example to enhance our understanding.

Code Example

We have outlined the steps for implementing the Decorator design pattern. Now, let's put them into practice with a simple example of making coffee. In this scenario, we have a base variety of coffee with a certain cost. Users can opt to add their own decorators to this base coffee, such as milk and/or sugar, and the price will be adjusted accordingly. Let's take a look:

Step 1: Define the component interface

Our very first step is to define the interface or the base class for our primary component, i.e., the coffee. So, let’s go ahead and do that:

1. class Coffee {
2. cost() {
3. return 5;
4. }
5. }

We have created a straightforward class **Coffee** with no properties but a single method, **cost**, that returns the value as 5 dollars. As we intend to use this **cost** method to enhance or update the cost of a coffee, we have opted to keep it as a number.

Step 2: Create concrete component

Next, we proceed to create a concrete component that will construct the base version of our component. Let's proceed with this step:

1. class SimpleCoffee extends Coffee {
2. *// You can override the methods if needed*
3. cost() {
4. return super.cost();
5. }
6. }

In this step, we have introduced a class SimpleCoffee that extends our base class, Coffee. Since decorators will be adding their respective variations to this simple coffee, we utilize the inherited method **cost** from the Coffee class. Inside this method, we return the base price of the coffee using the super.cost() method.

Step 3: Create the decorator class

Our next logical step is to define the interface or the base class for our multitude of decorators that we can create:

1. class CoffeeDecorator extends Coffee {
2. constructor(coffee) {
3. super();
4. this.\_coffee = coffee;
5. }
6. cost() {
7. return this.\_coffee.cost();
8. }
9. }

In the provided code example, we've introduced a CoffeeDecorator class that extends our original base component. This extension is necessary because the decorators essentially add their own flavour to the actual base component. Therefore, it's crucial to ensure that they have access to the original base component. In this code snippet, we accept an instance of coffee in the constructor, storing it as the local property \_coffee. Simultaneously, we inherit the properties of the Coffee class using super(). The cost method from the Coffee class is overridden, and it is used to return the cost of the coffee received in the constructor's argument while creating a new instance of the decorator.

Step 4: Create concrete decorators

In this step, we leverage the decorator base class created in the previous step to craft our decorators of the base component, Coffee. Let's first examine the code example before delving into a detailed understanding:

1. class MilkDecorator extends CoffeeDecorator {
2. cost() {
3. return super.cost() + 2;
4. }
5. }
6. class SugarDecorator extends CoffeeDecorator {
7. cost() {
8. return super.cost() + 1;
9. }
10. }

In the provided code example, two decorators, MilkDecorator and SugarDecorator, are created to impart their specific flavours to our base component, Coffee. These decorators extend the CoffeeDecorator interface, effectively isolating the entire decorator instances from the original code. This ensures that the original code remains unaffected and unmodified.

Within each respective decorator, the cost method is overridden to include the specific cost of the decorator in addition to the original coffee cost.

Step 5: Usage

For our final step in using the Decorator pattern, we will create instances of the concrete components and decorators, structuring them in the desired way. Let's proceed with this step:

1. *// Usage*
2. const simpleCoffee = new SimpleCoffee();
3. console.log("Cost of simple coffee:", simpleCoffee.cost());
4. const milkCoffee = new MilkDecorator(simpleCoffee);
5. console.log("Cost of coffee with milk:", milkCoffee.cost());
6. const sugarMilkCoffee = new SugarDecorator(milkCoffee);
7. console.log("Cost of coffee with sugar and milk:", sugarMilkCoffee.cost());

In the above code example, we have first created the instance of the simple coffee. Next, we create the instance of the milk coffee and add the simple coffee to its decorator instance. Finally, we make a coffee with both sugar and milk, and add the milk coffee instance to the SugarDecorator class to build this version.

On running the above code, you will receive the following output on your console:

1. Cost of simple coffee: 5
2. Cost of coffee with milk: 7
3. Cost of coffee with sugar and milk: 8

This demonstrates the substantial advantage of using the Decorator pattern to address this crucial problem. Throughout the final usage, there was no direct access or reference to the original code. It remained well encapsulated behind the scenes, safeguarding it while still influencing the overall behaviour. Furthermore, users only need to define several decorators in the initial steps. They can then employ these decorators as desired to create their personalized versions of coffee. This makes the process highly user-friendly, concealing the implementation details to a significant extent.

Applications

The Decorator design pattern in JavaScript can be useful in various scenarios where you want to add or modify functionality of objects dynamically and flexibly. Here are a few examples:

User Interface Components

Decorator patterns can prove highly beneficial in constructing UI components. Consider a base UI component class, such as a button, where users can dynamically add features like tooltips, borders, or additional styling without altering the existing code. This flexibility allows for the creation of customizable and feature-rich UI elements, enhancing the user interface without complicating the underlying structure.

Data Validation

Decorator patterns can be useful when we have data validation requirements for our code. For instances where we want to add validation rules to objects without modifying their original validation logic, these patterns can come to our assistance. For example, decorating a form validation object with additional validation rules like checking for password complexity or email format.

Authentication

We can use the decorator patterns for instances where we want to add authentication checks to certain functions or methods without modifying their core functionality. For instance, decorating a resource-fetching function with authentication checks before making the actual request.

Dynamic Configuration

We can use these patterns to dynamically configure objects with different settings without altering their original structure.

Event Handling

We can also use decorator patterns to dynamically attach or detach event handlers to objects without modifying their existing event handling code. Decorating a DOM element with additional event listeners based on user interactions. This could allow us to work independently of the element’s implementation and not affect it.

Pros and Cons

Let’s look at the advantages of implementing the Decorator pattern in our JavaScript code:

* We can extend an object’s behaviour without making a new subclass specifically for its demand.
* We can add or remove responsibilities from the object at runtime.
* Thanks to decorator patterns, we can combine several behaviours on to an object by using multiple decorators.
* The Decorator pattern adheres to the Single Responsibility Principle by addressing the challenge of extending an object's behaviour without necessitating the creation of multiple new classes for various permutations and combinations.

However, like every design pattern, the Decorator pattern also has its fair share of disadvantages listed down below:

* Ensuring that a decorator's behaviour remains independent of the order in the decorators stack can be challenging during implementation.
* The initial configuration code of the various layers can tend to look ugly and complex.

Façade design pattern

The Façade design pattern is a structural design pattern that provides a simple interface to a set of interfaces in a subsystem. It involves creating a higher-level interface that makes it easier to use a complex system or set of interfaces by providing a unified interface. In JavaScript, the Façade pattern is often used to hide the complexities of interacting with multiple components or APIs behind a single, simplified interface.

Problem Scenario

To understand the efficacy of the façade design pattern, let’s consider the example of a web application that needs to fetch and display the real-time weather information for a given location. To achieve this, we would require a probable integration of multiple weather APIs to gather comprehensive data like current temperature, humidity, wind speed, and forecasts.

However, this introduces a series of challenges. For instance, each weather API might have its own unique set of endpoints, request formats, and authentication mechanisms. Additionally, some APIs may necessitate additional processing or data transformation before it can be consistently presented in your application.

However, the challenges persist. The codebase might become cluttered with API-specific logic, leading to difficulties in maintenance and extension. If the third-party API undergoes changes in its implementation, such as adding or removing endpoints or modifying the authentication mechanism, it would necessitate widespread modifications throughout the application. Managing API keys, authentication tokens, and error handling for each API becomes a complex and error-prone task.

We would require a solution that makes it easier to manage the complexities arising with such scenarios.

Implementation

Let's break down the implementation of the Façade design pattern into distinct steps. The initial step, even before diving into the actual implementation logic, is to understand and identify the complex subsystem or set of interfaces that need simplification. Once the subsystem requiring modification is identified, the implementation process begins:

Step 1: Create the subsystem classes

The very first thing we need to do is to create the classes representing the subsystem components. These classes will handle the interactions with the respective subsystems.

Step 2: Create the façade classes

Next, we create a façade class that provides a simplified interface for the client code. This class encapsulates the interactions with the subsystem components. Its primary purpose is to simplify the interactions between the client code and the complex subsystem.

Step 3: Encapsulate the subsystem logic into the façade

Now, we need to encapsulate the logic and interactions with the subsystem components. The client code should interact with the facade and not need to know the details of the individual implementation logics.

Step 4: Expose simplified methods

Expose simplified methods in the Facade class that the client code can use. These methods should provide a coherent and easy-to-understand API, hiding the intricate details of the subsystem components. The Facade should handle any necessary coordination or translation between subsystem calls.

Step 5: Client code uses the façade

In the client code, instantiate the Façade class and use its simplified methods to perform tasks. The client code interacts solely with the Façade, unaware of the intricacies of the underlying subsystem components. This separation enhances code readability, maintainability, and flexibility.  
Let’s understand this better with a coding example in the next segment.

Code Example

Let's break down the implementation of the Facade design pattern in JavaScript using the example of an application that allows sharing to multiple social media platforms, namely Facebook, Instagram, and Twitter. Our main objective is to simplify the process of handling multiple API integrations without impacting our codebase.

The initial step before delving into the implementation of the Facade pattern is to comprehend the essence of the problem. In our scenario, the issue arises from the complexity stemming from different integrations with each social media platform. Variations in endpoints, authentication mechanisms, and intricate implementation logic pose challenges when integrating with these APIs.

Consider a component that needs to use all three APIs simultaneously; handling all three API integrations within a single component becomes a daunting task. The code would be cluttered with complex logic, including conditionals to manage each API integration. This problem intensifies when similar integrations are required for other components, leading to code repetition and the need for modification in those components. Additionally, introducing a new social media platform into the system would be a complex and time-consuming process. Essentially, the Facade design pattern aims to address and simplify these challenges.

Step 1: Create the subsystem classes

Our very first job would be to create classes that represent each of the subsystem. In our example, we'll create classes for interacting with Facebook, Twitter, and Instagram APIs.

1. class FacebookAPI {
2. postToFacebook() {
3. *// Implementation specific to posting on Facebook*
4. }
5. }
6. class TwitterAPI {
7. tweet() {
8. *// Implementation specific to tweeting on Twitter*
9. }
10. }
11. class InstagramAPI {
12. shareOnInstagram() {
13. *// Implementation specific to sharing on Instagram*
14. }
15. }

In the provided code example, three distinct classes, namely FacebookAPI, TwitterAPI, and InstagramAPI, have been established for the three subsystems. Each class is equipped with a unique method for sharing content on its respective platform. Implementation details, such as public keys or authentication mechanisms, can be encapsulated within these functions based on the specific requirements of each platform.

Step 2: Create the façade classes

Now, as our implementation dictates that we create a façade class that provides a simplified interface for the client code. Let’s create that:

1. class SocialMediaFacade {
2. constructor() {
3. this.facebookAPI = new FacebookAPI();
4. this.twitterAPI = new TwitterAPI();
5. this.instagramAPI = new InstagramAPI();
6. }
7. *// Methods to be exposed to the client will be implemented here*
8. }

In the provided code snippet, a class named SocialMediaFacade has been introduced to encapsulate interactions with the subsystem components. Three properties (facebookAPI, twitterAPI, and instagramAPI) have been defined to store instances of the respective subsystems (FacebookAPI, TwitterAPI, and InstagramAPI). This section of the logic primarily involves the instantiation of subsystem instances.

Step 3: Encapsulate the subsystem logic into the façade

Next, we need to add the subsystem logic into the façade. Our goal, we should remember, is to give the client code a single point of access for the integration, so that most of the implementation logic is encapsulated. Essentially, the client code should be interacting with the façade and not the individual social media APIs.

1. class SocialMediaFacade {
2. constructor() {
3. this.facebookAPI = new FacebookAPI();
4. this.twitterAPI = new TwitterAPI();
5. this.instagramAPI = new InstagramAPI();
6. }
7. *// Encapsulate the logic for sharing on all platforms*
8. shareOnAllPlatforms() {
9. const facebookResult = this.facebookAPI.postToFacebook();
10. const twitterResult = this.twitterAPI.tweet();
11. const instagramResult = this.instagramAPI.shareOnInstagram();
12. return { facebookResult, twitterResult, instagramResult };
13. }
14. }

In the provided code snippet, a shareOnAllPlatforms method has been added to the previously created SocialMediaFacade class. This method contains the implementation logic for interacting with individual APIs. For example, it calls the postToFacebook method of the facebookAPI instance and stores the result in the variable facebookResult. Similar actions are performed for Twitter and Instagram. The method returns an object containing the results, allowing the client code to receive the outcomes upon invocation.

Step 4: Expose simplified methods

In our example scenario, the previous step has covered both steps 3 and 4. By creating the method shareOnAllPlatforms in the previous example, we have also exposed it to be used by the client code on instantiation. By using only the shareOnAllPlatforms method, we make sure that the client is not exposed to the internal logic for implementing the APIs.

Step 5: Client code uses the façade

In the final step of our implementation, we consolidate the logic we've developed so far. This involves allowing the client code to utilize the façade for simplified interaction with multiple social media platforms.

1. const socialMediaFacade = new SocialMediaFacade();
2. const results = socialMediaFacade.shareOnAllPlatforms();
3. console.log(results);

Indeed, by using the SocialMediaFacade class, the client code benefits from a simplified and unified interface to interact with multiple social media platforms. We simply invoke the shareOnAllPlatforms method instead of handling all the different API integrations. This shields the client from the complexities of individual APIs and provides a convenient way to share content across various platforms.

Applications

The Facade pattern in JavaScript can be beneficial in various scenarios where you want to simplify and provide a unified interface to a complex subsystem or set of functionalities. Here are a few scenarios where the Facade pattern can be useful:

Working with multiple APIs

As we have seen in the coding example above, the Façade design pattern is perfect for scenarios where we need to work with multiple external APIs, each with its own set of endpoints, authentication mechanisms, and response formats. The Facade can hide the details of API communication, making it easier for the client code to interact with diverse services.

Browser Compatibility Issues

When dealing with cross-browser compatibility issues, the Facade can abstract away the differences in browser-specific implementations, providing a consistent interface for the client code to work with.

Legacy Code Integration

Integrating new features or modules with existing legacy code can be challenging. A Facade can be used to create a modern and simplified interface for the legacy code, allowing new components to interact with the old system seamlessly.

Resource Management

When dealing with resource management, such as handling connections to databases, caches, or file systems. The Facade can encapsulate the initialization, configuration, and clean-up processes, providing a straightforward interface for resource management.

Third-party Library Integration

Integrating third-party libraries that have intricate setups and configurations. The Facade can encapsulate the necessary steps to initialize and use the library, exposing a simplified interface for the client code.

In these scenarios, the Facade pattern helps improve code organization, reduce dependencies, and enhance maintainability by providing a clean and simplified interface for the client code.

Pros and Cons

The advantage of using the Façade design pattern in our code is that it can isolate our code from the complexity of the system.

At the same time, the major disadvantage here is that it can also end up becoming a God object, thereby introducing tight coupling in our code and that can prove to be disastrous to our code.

**Flyweight design pattern**

The Flyweight design pattern is a structural design pattern that is used to reduce the memory footprint or computational expense of an object by sharing as much as possible with related objects. This pattern is particularly useful when a large number of similar objects need to be created, and the overhead of creating and managing each individual instance is too high.

In the context of JavaScript, this pattern can be implemented using a combination of shared and non-shared data, where the shared data represents the part of an object that can be shared among multiple instances, while the non-shared data in unique to every instance.

Problem Scenario

Let's examine a scenario that can shed light on the types of issues that can be addressed by the Flyweight design pattern. Let’s consider the example of game development. In a video game scenario, such as the **Grand Theft Auto** (**GTA**) series, **numerous non-player characters** (**NPCs**) populate the virtual world. NPCs are characters that exist in the game environment but are not controlled by the player. They may include citizens going about their routines while the main character undertakes missions.

Each NPC in the game possesses common characteristics, like appearance, behaviour, or animations (considered intrinsic state), along with unique properties such as positions and health (considered extrinsic state). Creating a distinct object for every NPC that encompasses both intrinsic and extrinsic states could result in significant memory consumption, especially considering the large number of NPCs present in a typical game.

We would require a solution that can help us save memory by grouping and sharing the common properties. This separation of the common properties and the unique data of every object can help reduce memory consumption and improve the efficiency of the application.

Implementation

Let's go through the step-by-step process of implementing the flyweight design pattern in a JavaScript code:

Step 1: Identify the intrinsic and extrinsic states

The first step in implementing the Flyweight Design Pattern involves clearly identifying and differentiating between **intrinsic** and **extrinsic** properties. **Intrinsic properties** are those that remain constant across multiple instances and can be shared. These properties are stored within the flyweight object to minimize memory usage. **Extrinsic properties** are unique to each instance and cannot be shared. These are passed as parameters during method calls to maintain flexibility while keeping the shared object lightweight. For instance, imagine a text editor displaying a large document with various characters. The **intrinsic state** could include the character's font type, style, and size, which are shared across multiple characters of the same type. The **extrinsic state** could be the position (x, y coordinates) of each character on the screen, which is unique to every character instance.

Step 2: Create the flyweight interface

Next, we need to define an interface or the base class that will declare the method(s) that will be used by the concrete flyweights that will extend this class. This interface usually includes the operation method.

Step 3: Implement concrete flyweight class

Next, we need to create a concrete flyweight class that implements the flyweight interface. This class represents the shared (intrinsic) state.

Step 4: Create a flyweight factory

Now, we need to implement a flyweight factory that manages the creation and retrieval of flyweight instances. The factory ensures that instances are shared when possible.

Step 5: Client code

As the final step, we will create instances of the flyweight factory in the client code and use it to get or create flyweight instances.

An additional step can be added here where test the implementations to ensure that shared and unique states are handled correctly, and refine the design as needed based on testing and performance considerations.

To gain a better understanding of this implementation, let's explore a real-life coding example in the next section.

Code Example

Let's apply the example of a music-playing application to implement the steps we discussed in the previous section and incorporate the flyweight design pattern into our codebase. Imagine we are building a music streaming service where users can create playlists with a large number of songs. Each song has common properties like its title, artist, and duration (intrinsic state), but users can add songs to different playlists, each with its own order and play count (extrinsic state).

Step 1: Identify the intrinsic and extrinsic states

In the first step, we will identify and differentiate the intrinsic and extrinsic properties or states for our application. So, let’s go ahead and define that:

* **Intrinsic properties:** The properties that can be shared among multiple instances – title, artist, duration.
* **Extrinsic properties:** The properties that will be unique to each instance - order in playlist, play count.

Step 2: Create the flyweight interface

Now, we need to define the interface or the base class that will act as an interface for our flyweight objects.

1. *// Flyweight interface*
2. class SongFlyweight {
3. constructor() {
4. this.intrinsicState = null; *// Shared properties*
5. }
6. operation(extrinsicState) {
7. *// Shared behaviour using intrinsic and extrinsic states*
8. }
9. }

So, we have now created the class SongFlyWeight that will be implemented to create the flyweight objects. In the constructor, we have created a property intrinsicState that will include the shared properties and an operation method that accepts an argument extrinsicState to represent shared behaviour.

Step 3: Implement concrete flyweight class

Now that we've created the interface or the base class to derive a concrete class, let's examine its code implementation before delving into an understanding of the code:

1. *// Concrete Flyweight*
2. class ConcreteSongFlyweight extends SongFlyweight {
3. constructor(title, artist, duration) {
4. super();
5. this.intrinsicState = { title, artist, duration };
6. }
7. operation(extrinsicState) {
8. *// Implement shared behavior using intrinsic and extrinsic states*
9. }
10. }

In this example, we extended the SongFlyWeight class to create the concrete class ConcreteSongFlyweight. By using the super method, we inherited the properties defined in the SongFlyWeight class. Simultaneously, an instance of this class must provide the title, the artist, and the duration of the song, which we have designated as intrinsic properties—properties shared by the objects. We will let the operation method be used in the client code.

Step 4: Create a flyweight factory

Next for the sake of separation of concerns, we will create a flyweight factory that will be responsible for managing the creation and retrieval of flyweight objects:

1. class SongFlyweightFactory {
2. constructor() {
3. this.flyweights = {};
4. }
5. getSongFlyweight(title, artist, duration) {
6. const key = `${title}-${artist}-${duration}`;
7. if (!this.flyweights[key]) {
8. this.flyweights[key] = new SongFlyweight(title, artist, duration);
9. }
10. return this.flyweights[key];
11. }
12. }

The **getSongFlyweight** method now constructs the key by including the duration along with the title and artist. This ensures that each unique combination of title, artist, and duration results in a distinct key, preventing collisions where songs with the same title and artist but different durations might incorrectly share the same flyweight object.

By doing so, the Flyweight pattern is implemented more robustly, accurately reflecting the uniqueness of each song instance.

Step 5: Client code

Now, let's apply the defined code to implement it in the client code:

1. *// Client code*
2. const songFlyweightFactory = new SongFlyweightFactory();
3. const song1 = songFlyweightFactory.getSongFlyweight('Song A', 'Artist X', '3:30');
4. const song2 = songFlyweightFactory.getSongFlyweight('Song B', 'Artist Y', '4:15');
5. song1.operation({ order: 1, playCount: 10 });
6. song2.operation({ order: 2, playCount: 5 });

In the client code, we create an instance of the **SongFlyweightFactory**. We then use the factory to get or create **SongFlyweight** instances (**song1** and **song2**) by providing unique intrinsic properties. Finally, we call the **operation** method on each instance, passing extrinsic properties like order in the playlist and play count.

By following these steps, you create a system where shared properties of songs are efficiently managed by flyweights, reducing memory overhead in a music playlist application. This allows you to handle a large number of similar objects with less memory consumption.

Applications

The Flyweight pattern can be effective in various scenarios in JavaScript, especially when dealing with a large number of similar objects where memory usage needs to be optimized. Here are some scenarios where the Flyweight pattern can be beneficial:

Rendering engine in a game

In a game development scenario, if we have a multitude of similar objects (e.g., trees, rocks) with shared characteristics (e.g., texture, colour), we can use the Flyweight pattern to optimize memory usage by sharing common properties among instances.

UI components in a web application

In a web application with numerous UI components (e.g., buttons, dropdowns) that share common styles, we can use the Flyweight pattern to manage the common styling properties centrally, reducing redundancy and improving performance.

Document Object Model (DOM) nodes

When manipulating the DOM in a web application, if we have a large number of similar elements with shared properties (e.g., class names, styles), we can use the Flyweight pattern to efficiently manage and share these common properties.

Collaborative editing tools

In a collaborative editing tool, if we have multiple users working on a document and using similar text formatting options, we can use the Flyweight pattern to share the formatting information to minimize the memory footprint.

In each of these scenarios, the Flyweight pattern can help optimize memory usage by sharing common properties among similar objects, leading to improved performance and reduced resource consumption.

Pros and Cons

The advantage of using the Flyweight design pattern in our code is that it can save lots of RAM, provided the program has lots of similar objects.

However, there are two major disadvantages of the Flyweight design pattern:

* We might be trading RAM for CPU cycles when recalculation would happen every time someone calls a flyweight method.
* The code becomes far too complicated, if not handled correctly.

Proxy design pattern

The Proxy design pattern is a structural design pattern that provides surrogate or a placeholder for another object. A proxy controls the access to the original object, allowing us to add extra behaviour or restrictions. This can be useful for various scenarios like lazy loading, access control, and so on.

Problem Scenario

To understand the problem solved by the proxy design patterns, let’s take a very relevant example of a photo-sharing website where users can upload and view images. As users scroll through their feed, the application needs to load and display images. However, some images might be very large in size, leading to a slower loading time and a poor user experience.

In this scenario, loading all the images at once, especially the larger ones, can result in a very sluggish performance. Users generally do not like to wait for the images to load, and they will end up being frustrated and stop using the application altogether. Additionally, downloading large images can consume a significant amount of bandwidth, impacting both server and client resources.

The challenge is to optimize the image loading process, ensuring that only the images currently in the user's viewport are loaded. Loading images only when necessary can improve the application's performance and reduce the strain on network resources.

Implementation

Let's go through the step-by-step process of implementing the Proxy design pattern in our codebase:

Step 1: Identify the subject (real object)

The very first thing that we need to do is to identify the original object (subject) for which we want to control access. This is the object that the proxy will act as a surrogate for.

Step 2: Create the Proxy

Next, we will implement a proxy class that will act as a substitute or placeholder for the real object. The proxy controls access to the real object and can add additional behaviour.

Step 3: Define the proxy interface

Now, we need to create a proxy interface, i.e., a common interface that both the real object and the proxy will implement. This ensures that the proxy can be used interchangeably with the real object.

Step 4: Implement the real object

Now, we need to develop the class for the real object, which contains the actual functionality that the proxy will control access to.

Step 5: Implement the proxy

Now, we will create the proxy class, which wraps around the real object. The proxy intercepts requests and can add additional functionality or control access to the real object.

Step 6: Utilize the proxy in client code

As a final step, we will replace instances in our application where we use the real object, with the proxy. This allows the proxy to control access to the real object and apply the additional behaviour.

While it’s fairly easy to comprehend these steps, let’s have a look at a coding example in the next section to help us understand better.

Code Example

To understand the implementation of a proxy pattern effectively, let's explore the concept of lazy-loading in web applications. We will break down the process into steps for a clearer understanding and implementation.

Step 1: Identify the subject (real object)

In our example, the subject will be the image loader. So, let’s create a class surrounding it:

1. *// Real Object (Subject)*
2. class ImageLoader {
3. loadImage(url) {
4. console.log(`ImageLoader: Loading image from ${url}`);
5. *// Actual image loading logic*
6. }
7. }

In this example, we have defined a class ImageLoader with a single method loadImage that takes a url as a parameter. The loadImage method simulates loading an image by printing a message indicating the URL from which the image is being loaded. This serves as a basic representation of a loading logic in a real-world scenario.

Step 2: Create the Proxy

In the subsequent step, we aim to create a proxy for our main subject, the ImageLoader class. This proxy will be designed to intercept and control access to the loadImage method. Let's proceed with the implementation of this proxy:

1. *// Proxy*
2. class ImageProxy {
3. constructor() {
4. this.imageLoader = new ImageLoader();
5. }
6. loadImage(url) {
7. *// Proxy logic for lazy-loading*
8. console.log(`ImageProxy: Request received for image from ${url}`);
9. *// Additional proxy logic for lazy-loading*
10. this.imageLoader.loadImage(url);
11. }
12. }

In this example, we have effectively implemented a proxy class for our ImageLoader. The proxy, represented by the ImageProxy class, contains an instance of the ImageLoader (stored in the imageLoader property). We've replicated the loadImage method from the ImageLoader and introduced a statement to indicate that this is the image proxy. Additionally, we've incorporated an extra logic for lazy-loading. The proxy utilizes the loadImage method of the imageLoader instance to load the actual image when needed.

Step 3: Define the proxy interface

Now, let's proceed with the crucial step of defining the proxy interface that both the real object and the proxy can implement. Let's go ahead and create that:

1. *// Interface for both Real Object and Proxy*
2. class ImageInterface {
3. loadImage(url) {
4. *// Common method for image loading*
5. }
6. }

In the example above, we've introduced the class ImageInterface, which features a singular method – loadImage. This method is shared between both the proxy and the original subject.

Step 4: Implement the real object

In this step, we need to make the real object implement the proxy interface:

1. *// Real Object (Subject)*
2. class ImageLoader extends ImageInterface {
3. loadImage(url) {
4. console.log(`ImageLoader: Loading image from ${url}`);
5. *// Actual image loading logic*
6. }
7. }

Here, we have modified our original class to implement the ImageInterface that we created in step 3.

Step 5: Implement the proxy

Now, we will make sure the proxy also implements the proxy interface:

1. *// Proxy*
2. class ImageProxy extends ImageInterface {
3. constructor() {
4. super();
5. this.imageLoader = new ImageLoader();
6. }
7. loadImage(url) {
8. *// Proxy logic for lazy-loading*
9. console.log(`ImageProxy: Request received for image from ${url}`);
10. *// Additional proxy logic for lazy-loading*
11. this.imageLoader.loadImage(url);
12. }
13. }

In the code example above, we've ensured that the proxy class ImageProxy also extends the ImageInterface. Additionally, we've included a super method in the constructor to guarantee the inheritance of all properties from the proxy interface.

Step 6: Utilize the proxy in client code

Now that all components are in place, let's proceed to implement the proxy code to manage the lazy-loading of images.

1. *// Client code*
2. const imageLoader = new ImageProxy();
3. imageLoader.loadImage("example.jpg");

For the final step, we have now ensured that we use the ImageProxy instead of the ImageLoader to load our images. If we run this code, this is the result that we will see in the browser console:

1. ImageProxy: Request received for image from example.jpg
2. ImageLoader: Loading image from example.jpg

As evident from the above outcome, the image proxy is consistently invoked first, subsequently initiating the request for our image loader to load the image.

This behaviour demonstrates the essence of the Proxy design pattern, where the proxy controls access to the real object and can add additional functionality. The lazy-loading logic can be implemented in the proxy to optimize the loading process based on specific requirements.

Applications

The Proxy pattern in JavaScript can be useful in various scenarios to provide control, optimization, and additional functionalities. Here are several instances where the Proxy pattern can be beneficial:

Lazy Loading

As we have already seen in the example, proxy design pattern can be used to implement lazy loading for resources such as images, scripts, or data. The proxy can delay the actual loading of resources until they are explicitly requested, improving initial page load times.

Access Control

We can employ a proxy to control access to sensitive operations or resources. For example, we can use a protection proxy to restrict access to certain methods or properties based on user roles or permissions..

Caching

We can implement a proxy to cache the results of expensive operations. The proxy can store the results and return them if the same operation is requested again, reducing the need for redundant computations.

Validation

We can use a proxy to validate input parameters or data before it reaches the actual object. This ensures that only valid data is processed by the real object, preventing potential issues.

Remote Proxy (AJAX Requests)

We can use a proxy to manage and optimize remote calls, such as AJAX requests. The proxy can handle tasks like authentication, caching, or batching requests before forwarding them to the actual service.

Pros and Cons

There are several advantages of using the Proxy design pattern in JavaScript. A few of them are as follows:

* We can control the service object and manage its lifecycle without the clients knowing about it and in scenarios, where the lifecycle doesn’t matter to the client.
* The best part about having a proxy is that the service object is not required to be available or ready. The proxy can take all the requests and wait until the service object is ready for implementation.
* Finally, it follows the Open/Closed Principle, where we can add new proxies without changing the service object or client.

The few disadvantages associated with the Proxy design pattern are as follows:

* Introduction of new classes could potentially complicate the code, if not handled or envisioned correctly.
* Moreover, the proxy merely serves as a waiting room until the actual service object can respond. So, there is always a chance that the service might get delayed.

Pattern selection tips

In this chapter, we explored seven types of structural design patterns along with their diverse use-cases and scenarios. When confronted with a real-world problem, determining the most suitable pattern might initially seem perplexing. Therefore, this section will serve as a guide to help us choose the right pattern for our specific use-case.

Adapter Pattern

As we know, Adapter pattern allows objects with incompatible interfaces to collaborate. With that in view, here are certain scenarios where this design pattern will be beneficial to use:

* When integrating a new component with an existing system.
* When the interface of an existing class needs to be compatible with a client's expectations.
* When working with legacy code that has an incompatible interface.

Bridge Pattern

Bridge pattern allows us to split a large class of closely related classes into two independent hierarchies. Therefore, Bridge pattern can work wonders for scenarios where:

* When we want to separate abstraction from implementation.
* When changes in the implementation should not affect the abstraction.
* When there is a need to extend both abstraction and implementation independently.

Composite Pattern

Composite pattern permits us to compose objects into discernible tree structures. This design pattern can help us:

* When we want to treat both individual objects and compositions of objects uniformly.
* When clients should be able to ignore the difference between compositions of objects and individual objects.
* When we need to represent part-whole hierarchies.

Decorator Pattern

Decorator design pattern, as its name suggests, helps us to attach new behaviours to objects by placing these objects inside special wrappers. This can prove beneficial in the following scenarios:

* When we want to add new functionalities to an object dynamically without altering its structure.
* When there is a need for a flexible and reusable way to extend behaviour.
* When we want to avoid subclassing to extend behaviour.

Façade Pattern

Facade pattern provides a simplified interface to a library, a framework, or any other complex set of classes. Therefore, it can be used extensively for the following scenarios:

* When we need to provide a simplified interface to a complex subsystem.
* When client code needs to interact with a subsystem, but we want to shield it from the complexities of that subsystem.
* When we want to decouple the client code from the implementation details of a subsystem.

Flyweight Pattern

Flyweight pattern helps us fit more objects into the available amount of RAM by sharing common parts of the state between multiple objects. Its utility can be beneficial in the following cases:

* When there are a large number of similar objects that can be shared to reduce memory usage.
* When the application uses a large number of objects that have similar properties.
* When the overhead of creating and maintaining a large number of similar objects is significant.

Proxy Pattern

Proxy pattern helps us by providing a substitute or a placeholder for another object. This can be useful in cases where:

* When we want to control access to an object.
* When we need to add functionality before or after the actual processing of requests.
* When we want to implement lazy loading of an expensive object.

Conclusion

In conclusion, this chapter provided an in-depth exploration of structural design patterns, which play a crucial role in organizing code for the development of flexible and scalable systems. Throughout the chapter, we delved into seven distinct types of structural design patterns. We started with the Adapter design pattern, beneficial for integrating newer code with legacy systems. The Bridge design pattern was next, creating a bridge to separate the abstraction and implementation aspects of complex problems. Following that, the Composite design pattern emerged as a solution for tree-based structures with both leaf and composite elements. The Decorator pattern was discussed, offering a way to add new functionalities to existing objects. Façade pattern simplifies the handling of various integrations by creating a unified interface. Flyweight pattern optimizes memory usage by sharing common components. Lastly, the Proxy pattern enables the creation of a surrogate that takes control before accessing the original object, allowing for the addition of functionalities or behaviours.

A common thread among all these patterns is that structural design patterns provide a systematic approach to addressing code complexity through effective code structuring and organization. When applied appropriately, these patterns serve as powerful tools for constructing systems that are not only scalable but also reliable. In the ever-evolving landscape of software development, the understanding and mastery of structural design patterns are indispensable. As we move forward, armed with the insights gained from this chapter, it is imperative for developers to leverage these patterns judiciously, adapting them to the unique requirements of each project. By doing so, we contribute to the creation of robust, flexible, and sustainable software architectures that stand the test of time.

Points to remember

* Structural design patterns can be likened to assembling individual objects like LEGO pieces. Just as arranging and structuring these pieces can result in masterpieces, appropriately organizing and structuring objects can lead to the development of robust code that stands the test of time.
* The Adapter design pattern involves creating an interface capable of accommodating newer methods of implementation. It provides a suitable adapter or modification that serves as a bridge between the new and different method and the traditional legacy approach of a code.
* The Bridge design pattern advocates separating the implementation part of the code from the abstraction part. This separation allows encapsulating the implementation code, facilitating the creation of cross-platform applications.
* The Composite design pattern is well-suited for code with a hierarchical or tree structure. It effectively addresses complexities arising from components that can have children, accommodating both leaf nodes and composite nodes in a structured manner.
* The Decorator design pattern enables the addition of functionalities or behaviours to code at runtime without modifying the original code base.
* The Façade pattern acts as a literal façade, hiding the complex implementation logic of integrating multiple components. Instead, it provides a simplified interface for the client code to interact without being aware of the underlying complexities.
* The Flyweight design pattern is useful in reducing memory usage by efficiently managing shared and non-shared components, particularly in scenarios involving a large number of objects.
* The Proxy design pattern creates a surrogate that can access the actual object and add behaviours or functionalities to it before the actual object can be accessed.

Exercises

1. Which structural design pattern can effectively handle the scenario of integrating multiple APIs into a common codebase?
   1. Adapter
   2. Proxy
   3. Decorator
   4. Façade
2. Which structural design pattern is described by the following statement – “splitting large classes or closely related classes into two parts – abstraction and its implementation, allowing them to be developed independently of each other”?
   1. Composite
   2. Bridge
   3. Adapter
   4. Singleton
3. Which of the following applications define the correct use-case for a Flyweight design pattern?
   1. Lazy Loading
   2. Access Control
   3. Data Validation
   4. Integration to legacy code.

**Answers:** 1 – d, 2 – b, 3 - a