

Bridge Pattern

Introduction

The Bridge Design Pattern is a structural design pattern that separates two things so they can change independently. Imagine you have different types of remote controls and different devices like TV and Radio. Without the Bridge pattern you would need separate classes for every combination - BasicTVRemote, AdvancedTVRemote, BasicRadioRemote etc. This creates too many classes. With 5 remote types and 10 devices you would need 50 separate classes, leading to class explosion and massive code duplication. Bridge pattern solves this by keeping remotes in one group and devices in another group and then connecting them through composition. Instead of creating a class for every combination, it creates two independent hierarchies that work together. The remote classes hold a reference to device objects, and this reference acts as the bridge connecting the two hierarchies. This way adding a new remote type or device doesn't require creating many new classes. Just like how a physical bridge connects two separate pieces of land allowing traffic to flow between them, the Bridge pattern connects two separate hierarchies allowing them to work together. The pattern is called Bridge because it connects two separate parts of your code allowing both to grow independently while working together.

Problem Statement

Consider a scenario where you need to create classes for different combinations of two varying dimensions. Using a remote control example with 2 remote types i.e. Basic and Advanced and 3 devices TV, Radio and AC you need 6 separate classes. With 5 remote types and 10 devices you need 50 classes. Without Bridge Pattern you need to create separate classes like BasicTVRemote, BasicRadioRemote, AdvancedTVRemote, AdvancedRadioRemote etc. This approach leads to massive code duplication because similar logic is repeated across multiple classes. For example all TV remote classes will have similar TV related code and all basic remote classes will have similar basic functionality code. When you want to add a new device type, you must create new classes for every existing remote type. Similarly adding a new remote type requires creating classes for every existing device. This makes the system very difficult to extend and maintain.

If you need to change how a device works you must update that logic in multiple remote classes. If you need to modify remote functionality you must change it across multiple device specific classes. This tight coupling between remote types and device types makes the entire system rigid and fragile. The system becomes difficult to test because

each combination class needs separate testing. Maintenance becomes a nightmare as any change to device behavior or remote features requires updating multiple classes, increasing the risk of bugs and inconsistencies.

Solution

The Bridge Design Pattern solves this problem by separating the abstraction from the implementation. Instead of creating a class for every combination, it creates two independent hierarchies that work together through composition. In the remote control example one hierarchy is created for remotes i.e. Basic Remote and Advanced Remote and another separate hierarchy for devices i.e. TV, Radio and AC.

The remote classes don't directly inherit from device classes. Instead each remote holds a reference to a device object. This reference acts as the bridge connecting the two hierarchies. The remote classes don't directly implement device-specific operations. Instead each remote class only holds a reference to a device object and defines what operations are available i.e. power on, volume up and mute. The device classes define how those operations are actually performed for each specific device. When a button is pressed on the remote it delegates the actual work to the device object it's connected to. For example, `BasicRemote.powerOn()` calls `device.powerOn()`, and the device object (TV, Radio, or AC) performs the actual power on operation specific to that device type. The remote doesn't need to know which specific device it's controlling, it just calls methods on the device interface.

This separation allows adding new remote types by creating new remote classes without touching device classes, and adding new devices by creating new device classes without modifying remote classes. The device a remote controls can even be switched at runtime giving flexibility that pure inheritance cannot provide. The pattern uses composition over inheritance. Each remote has-a device rather than is-a specific device remote. This loose coupling makes the system easy to extend, maintain and modify. Changes to device behavior only require updating the device classes and changes to remote features only affect remote classes. Related remotes and devices are grouped in separate hierarchies making the code more organized and maintainable.

Code

Code implementation of Chain of Responsibility pattern:

<https://github.com/prateek27/design-patterns-java/tree/main/design-patterns/src/main/java/org/prateek/StructuralPatterns/BridgePattern>

Use Cases

The Bridge Design Pattern is most useful when a system has two dimensions that can vary independently. If both the abstraction and implementation need to be extended separately without affecting each other, Bridge pattern provides the perfect solution. This is particularly valuable in systems where the abstraction layer and implementation layer evolve at different rates or have different requirements. The pattern allows each dimension to be modified independently, following the Open/Closed Principle where classes are open for extension but closed for modification.

Bridge pattern should be used when there is a need to avoid permanent binding between abstraction and implementation. This is particularly important when the implementation needs to be selected or switched at runtime. Consider a payment processing system where different payment methods need to work with different payment gateways. The payment method abstraction can work with any payment gateway implementation, and the gateway can be selected or changed at runtime based on merchant preferences or availability. This runtime flexibility is not possible with traditional inheritance where the relationship is fixed at compile time.

The pattern is highly beneficial when the number of classes would grow exponentially due to combinations. If a system has N abstractions and M implementations, traditional inheritance would require $N \times M$ classes, while Bridge pattern only needs $N + M$ classes. This dramatic reduction in class count prevents class explosion and keeps the codebase manageable. For example, with 5 remote types and 10 devices, traditional inheritance requires 50 classes, while Bridge pattern only needs 15 classes (5 remotes + 10 devices). This makes the system easier to understand, test, and maintain.

Bridge pattern works well when both hierarchies need to be extended by subclassing. If new types need to be added frequently, Bridge pattern allows adding them independently without touching existing code. New abstractions can be added by creating new abstraction classes without modifying implementation classes, and new implementations can be added by creating new implementation classes without touching abstraction classes. This independent extension capability makes the system highly flexible and adaptable to changing requirements.

However, the Bridge pattern should not be used when there is only one implementation. If a system will always have just one way of doing things, the added complexity is unnecessary. Simple inheritance would be more appropriate. The pattern adds unnecessary abstraction layers when there is no need for multiple implementations, making the code harder to understand without providing any benefits. Introducing

Bridge pattern in such scenarios creates overhead without any real advantage, complicating the codebase unnecessarily.

The pattern is also not suitable when the abstraction and implementation are unlikely to change. If the system is stable and both dimensions are fixed, introducing the Bridge pattern adds unnecessary complexity. The overhead of maintaining two separate hierarchies is not justified if they will never vary independently. In such cases, a simpler design with direct inheritance or composition would be more straightforward and easier to maintain. The pattern should only be used when there is genuine need for independent variation of abstraction and implementation.

Applications

Database Drivers: JDBC (Java Database Connectivity) is a classic example of Bridge pattern. The JDBC API serves as the abstraction that defines standard database operations. Different database vendors like MySQL, PostgreSQL, Oracle provide their own driver implementations. Applications can switch between databases by simply changing the driver without modifying the application code. This allows developers to write database-agnostic code that works with any database vendor, providing flexibility and portability across different database systems.

GUI Frameworks: Cross-platform GUI frameworks like Java Swing and JavaFX use Bridge pattern. The framework provides abstraction for UI components like buttons, windows and menus. Each operating system (Windows, macOS, Linux) provides its own implementation for rendering these components. The same application code can run on different platforms without changes. This enables developers to create platform-independent applications while leveraging native rendering capabilities of each operating system for optimal performance and user experience.

Graphics Rendering Systems: Graphics applications often use Bridge pattern to separate drawing logic from rendering platforms. The abstraction defines shapes like circles, rectangles and lines. The implementation can be OpenGL, DirectX, SVG or Canvas, allowing the same shapes to be rendered using different graphics libraries. This separation allows applications to switch between rendering backends based on platform capabilities or performance requirements without changing the core drawing logic.

Payment Processing Systems: E-commerce platforms use Bridge pattern for payment processing. Payment methods like credit card, debit card, UPI and wallet form the abstraction. Payment gateways like Stripe, PayPal, Razorpay and Square provide the

implementation. Merchants can support multiple payment methods across different gateways flexibly. This design allows businesses to add new payment methods or switch payment gateways without modifying the core payment processing logic, providing flexibility in payment integration.

Messaging Systems: Notification systems implement Bridge pattern where message types (email, SMS, push notification) form the abstraction and delivery channels (SendGrid, Twilio, Firebase) provide the implementation. New message types or delivery channels can be added independently. This architecture enables systems to support multiple notification channels and message types while allowing easy addition of new providers or message formats without affecting existing functionality.

Device Drivers: Operating systems use Bridge pattern for device drivers. The OS provides abstract interfaces for devices like printers, scanners and storage devices. Hardware manufacturers provide specific driver implementations for their devices, allowing the OS to support various hardware without knowing implementation details. This abstraction enables the operating system to work with diverse hardware from different manufacturers while maintaining a consistent interface for device management and communication.

Advantages and Disadvantages

Advantages

Bridge pattern provides strong decoupling between abstraction and implementation. This separation allows both to be developed, tested and maintained independently. Changes in the implementation do not affect the abstraction and vice versa, making the codebase more stable and easier to manage.

The pattern enables independent extension of both hierarchies. New abstractions can be added without modifying implementations, and new implementations can be added without touching abstractions. This follows the Open/Closed Principle where classes are open for extension but closed for modification.

Runtime flexibility is a major advantage of Bridge pattern. The implementation can be selected or switched at runtime based on requirements or conditions. This is not possible with traditional inheritance where the relationship is fixed at compile time.

Bridge pattern dramatically reduces the number of classes needed. Instead of creating $N \times M$ classes for N abstractions and M implementations, only $N + M$ classes are

required. This prevents class explosion and keeps the codebase manageable and organized.

The pattern improves code reusability as implementations can be shared across multiple abstractions. It also hides implementation details from clients, providing better encapsulation and protecting the internal structure from external dependencies. Implementations can be reused in different contexts, and abstractions can work with different implementations without code duplication, making the system more efficient and maintainable.

Disadvantages

Bridge pattern increases overall complexity of the codebase. It introduces additional classes and interfaces which can make the system harder to understand initially. For simple scenarios, this added complexity may not be justified.

The pattern adds a level of indirection through the bridge connection. This can make the code slightly harder to follow and debug as method calls pass through multiple layers. There is also a minor performance overhead due to this indirection. When debugging, developers need to trace method calls through the abstraction layer, then through the bridge, and finally to the implementation layer, which can make troubleshooting more time-consuming.

Bridge pattern requires careful upfront design to identify the right abstraction and implementation boundaries. If these boundaries are not correctly identified, the pattern may not provide the intended benefits and could make the system more complicated than necessary. The abstraction and implementation must be clearly separated, and their responsibilities must be well-defined.

For systems with only one implementation or implementations that rarely change, Bridge pattern is overkill. The overhead of maintaining two separate hierarchies is not worth it when simpler approaches would suffice. The pattern should only be used when there is genuine need for independent variation. Introducing Bridge pattern in such scenarios adds unnecessary abstraction layers that complicate the code without providing any real benefits, making it harder for developers to understand and maintain the system.