What is the Bridge pattern, and what problem does it solve in software design?

O padrão Bridge separa a abstração da implementação, permitindo que elas variem independentemente. Resolve problemas de complexidade e promove flexibilidade no design de software, especialmente quando há múltiplas dimensões de variação. Evita o acoplamento excessivo entre classes, facilitando a manutenção e extensão do sistema.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other. The primary advantage of the Bridge pattern is that it decouples the Abstraction and the Implementation, allowing them to evolve independently. This separation makes it easier to modify or extend both the Abstraction and the Implementation without affecting each other.

O padrão Bridge é uma abordagem de design estrutural utilizada para desacoplar uma abstração da sua implementação, permitindo que ambas possam variar de forma independente. Este padrão permite que a abstração e a sua implementação sejam desenvolvidas independentemente e depois combinadas.
O principal problema que o padrão Bridge resolve no design de software é o problema de uma hierarquia de classes rígida.

Bridge permite dividir uma classe grande ou um conjunto de classes intimamente relacionadas em duas hierarquias separadas – abstração e implementação – que podem ser desenvolvidas independentemente uma da outra.

O problema que o padrão Bridge resolve está relacionado à necessidade de desacoplar uma abstração de sua implementação. Em muitos casos, você pode ter diferentes tipos de abstrações que precisam ser combinadas com diferentes implementações. Se essas abstrações e implementações estiverem diretamente ligadas, o código pode se tornar complexo e difícil de manter. Além disso, qualquer alteração em uma abstração ou implementação pode exigir alterações em outras partes do código, o que viola os princípios de design sólidos, como o Princípio Aberto/Fechado (Open/Closed Principle) e o Princípio da Responsabilidade Única (Single Responsibility Principle).

Describe the structure of the Bridge pattern. What are the main components involved?

O padrão Bridge consiste em:

1. Abstração: Define a interface de alto nível.
2. Implementador: Define a interface de baixo nível.
3. Abstração Refinada: Estende a funcionalidade da Abstração.
4. Implementador Concreto: Classes que implementam o Implementador.

Em resumo, o padrão Bridge divide o sistema em duas hierarquias separadas, permitindo flexibilidade e manutenção mais fácil.

Structure of the Bridge Pattern
Abstraction: This is the core of the Bridge pattern. It defines the interface for the abstraction. It contains a reference to the implementer.
Refined Abstraction: Extends the abstraction to take the finer detail one level below. It hides the finer elements from implementers.
Implementer: Defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different.
Concrete Implementation: Implements the implementer by providing the concrete implementation.

The Bridge Pattern operates by splitting the functionalities into two separate hierarchies:
Abstraction hierarchy: This represents the high-level part of the code, encompassing interfaces or abstract classes that define and control the abstraction.
Implementation hierarchy: This contains the concrete implementations that the abstraction can utilize.
The main components are the following:
Abstraction: An interface or abstract class that represents the high-level control operations. This typically maintains a reference to the Implementor.
RefinedAbstraction: Extends the Abstraction to provide more concrete or variant controls.
Implementor: An interface or abstract class defining the low-level operational methods. This acts as a base for the concrete implementation.
ConcreteImplementor: Provides concrete implementations for the methods defined in the Implementor.

Abstração:
Define a interface para a abstração de alto nível. Mantém uma referência a um objeto de implementação e delega todo comportamento específico da implementação a ele.

Abstração Refinada:
Estende a abstração adicionando métodos adicionais ou refinando os existentes. É uma subclasse da abstração.

Implementação:
Define a interface para as classes de implementação de nível inferior. Normalmente fornece operações básicas que a abstração precisa executar, mas de uma maneira que pode variar entre diferentes implementações.

Implementação Concreta:
Esta é a implementação real da interface de implementação. As classes de implementação concretas fornecem implementações específicas para as operações definidas pela interface de implementação.

What are the benefits of using the Bridge pattern in software development? Provide some practical examples.

Usar o padrão Bridge no desenvolvimento de software oferece benefícios como desacoplamento entre abstração e implementação, flexibilidade para trocar implementações, extensibilidade fácil, testes simplificados e redução de duplicação de código. Exemplos incluem kits de ferramentas GUI para diferentes sistemas operacionais, camadas de persistência de dados independentes do banco de dados específico e aplicativos de controle remoto que podem se comunicar com vários dispositivos usando protocolos diferentes.

Benefits:
- You can create platform-independent classes and apps.
- The client code works with high-level abstractions. It isn't exposed to the platform details.
- Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
- Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.
Examples of the Bridge Pattern
Window System: A classic example is a window system that supports different window managers (e.g., X Window Manager and IBM's Presentation Manager). The window system can be the abstraction, and the window managers can be the implementers. By using the Bridge pattern, you can create different types of windows (e.g., icon windows, transient windows) that can work with any window manager without needing to create a separate class for each combination of window type and window manager.
Vehicle and Workshop: Another example is a vehicle assembly system where vehicles can be assembled in different workshops. The vehicle can be the abstraction, and the workshops can be the implementers. The Bridge pattern allows you to change the workshop at runtime without affecting the vehicle's assembly process

O padrão Bridge em desenvolvimento de software oferece vários benefícios, tais como o desacoplamento entre abstração e implementação, facilitando a manutenção e extensibilidade do código. Além disso, ele promove uma melhor organização do código, tornando-o mais modular e fácil de entender. Dois exemplos práticos são: frameworks de GUI, onde formas podem ser renderizadas em diferentes dispositivos; e controles remotos, que separam o controle das diferentes funções dos dispositivos controlados. Esses exemplos mostram como o padrão Bridge pode simplificar a gestão da complexidade em sistemas de software.

Podemos criar classes e aplicativos independentes de plataforma.

O código do cliente trabalha com abstrações de alto nível. Não é exposto aos detalhes da plataforma.

Princípio Aberto/Fechado. Pode introduzir novas abstrações e implementações independentemente uma da outra.

Princípio da Responsabilidade Única. Pode focar na lógica de alto nível na abstração e nos detalhes da plataforma na implementação.

O padrão Bridge é um padrão de design estrutural que separa uma abstração de sua implementação, permitindo que ambas variem independentemente. Ele resolve o problema de ter uma hierarquia de classes rígida e inflexível, tornando as abstrações e implementações independentes umas das outras, facilitando a extensão e a evolução do sistema

A estrutura do padrão Bridge é composta pelos seguintes componentes principais:

Abstraction: Define a interface abstrata para os clientes interagirem. Pode conter referências para objetos da interface Implementor.

Implementor: Define a interface para as classes de implementação concreta. As classes Implementador fornecem implementações concretas para as operações definidas na interface abstrata.

Concrete Implementor: Implementa a interface definida pelo Implementador. Fornece implementações específicas para os métodos definidos pela interface Implementador.

Refined Abstraction: Uma extensão da Abstração que pode adicionar funcionalidades adicionais, mas ainda depende da interface do Implementador para realizar suas operações.

O padrão Bridge oferece vários benefícios em desenvolvimento de software, incluindo:

Desacoplamento de abstrações e implementações: Permite que as abstrações e as implementações variem independentemente umas das outras. Isso significa que você pode alterar ou estender uma sem precisar modificar a outra, resultando em um sistema mais flexível e de fácil manutenção.

Facilita a extensibilidade: Como abstrações e implementações são separadas, é mais fácil adicionar novas funcionalidades ou comportamentos sem modificar o código existente. Isso é particularmente útil em sistemas onde existem múltiplas variações de funcionalidades.

Promove o princípio da composição sobre a herança: Em vez de depender da herança para estender funcionalidades, o padrão Bridge favorece a composição, o que geralmente é considerado uma prática mais flexível e menos suscetível a problemas de hierarquia de classes.

Favorece a reutilização de código: Como a implementação é separada da abstração, as implementações podem ser reutilizadas em várias abstrações, e vice-versa, promovendo o reuso de código e reduzindo a duplicação.

Um interruptor doméstico que controla luzes, ventiladores de teto, etc., são exemplos que podem ser adaptados para demonstrar o padrão Bridge.

The Bridge pattern is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. It solves the problem of the "cartesian product" complexity that arises when you have multiple dimensions of variation in your software design.

Abstraction: This is the high-level interface that defines the abstract methods or operations that the clients will use.

Implementor: This is the interface or abstract class that defines the methods that the concrete implementors must implement.

Concrete Abstraction: These are concrete classes that extend the Abstraction and use an Implementor to perform specific operations.

Concrete Implementor: These are concrete classes that implement the Implementor interface and provide actual implementations of the methods defined in the Implementor.

Decoupling of Abstraction and Implementation, Enhanced Extensibility, Improved Maintainability, Platform Independence.

For example, in a drawing application that supports different shapes and different rendering methods, instead of having a complex hierarchy where ach shape class is tightly coupled with a specific rendering method, you can use the Bridge pattern to separate the shape hierarchy from the rendering hierarchy.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies (abstraction and implementation) which can be developed independently of each other. The Bridge pattern attempts to solve the problem of managing complexity and promoting flexibility in software design.

The Bridge pattern is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently.

The main components involved in the Bridge pattern are:

Abstraction: This is an interface or abstract class that defines the abstract interface.
Refined Abstraction: This is a class that extends the interface defined by the Abstraction.
Implementer: This is an interface that defines the interface for implementation classes.
Concrete Implementor: This is a class that implements the Implementer interface.

The Bridge pattern offers several benefits in software development, primarily focusing on flexibility, maintainability, and scalability, such as: Decoupling of Abstraction and Implementation, Increased Flexibility and Maintainability by separating the abstraction from its implementation, Scalability by allowing the addition of new abstractions and implementations without modifying existing code.

The bridge pattern states that you should change the components of an object into different classes as to easily change the properties of an eventual future object

In the bridge pattern you have the abstraction and the implementation

The bridge pattern allows the abstraction of an object to be different from it's implementation, making it easier to create different objects that share some similarities.

The Bridge pattern decouples abstraction from implementation, mitigating Cartesian product complexity by separating class hierarchies for dimensions like Shape and Color. It enhances extensibility by allowing independent addition of new abstractions and implementations. By hiding implementation details from the client, it fosters flexibility and scalability in software design, facilitating seamless combination of different abstractions and implementations without the need for excessive class definitions.

The Bridge pattern comprises Abstraction, which defines high-level operations and maintains a reference to an Implementor; RefinedAbstraction, extending Abstraction with specific operations; Implementor, an interface delegated operations; and ConcreteImplementor, providing actual implementations. This structure decouples abstraction from implementation, facilitating independent variations and reducing complexity in software design.

The Bridge pattern is a structural design pattern that separates an abstraction from its implementation, allowing both to vary independently. This pattern is useful when there are multiple dimensions of variability in your classes, and you want to avoid ending up with a combinatorial explosion of subclasses to cover every combination of these dimensions.

Key components of the Bridge pattern are the Abstraction, RefinedAbstraction, Implementor, and ConcreteImplementor. The Abstraction defines the high-level operations, which are implemented in the Implementor. The RefinedAbstraction provides more specific operations, and the ConcreteImplementor provides the actual implementation.

Benefits of the Bridge pattern include decoupling of abstraction and implementation, improved extensibility, and hiding implementation details from clients. It's commonly used in device drivers, cross-platform UIs, networking, and database connectivity.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other. The Bridge pattern solves the problem of having a rigid coupling between abstraction and implementation.	The main components of the Bridge pattern are the Abstraction, Refined Abstraction, Implementor, and Concrete Implementor. These components work together to decouple abstraction from implemmentation, promoting flexibility and extensibility in software design	The benefits are: You can create platform-independent classes and apps. The client code works with high-level abstractions. It isn't exposed to the platform details. Open/Closed Principle: you can introduce new abstractions and implementations independently from each other. Single Responsibility Principle: you can focus on high-level logic in the abstraction and on platform details in the implementation. One example are Drawing Programs: Imagine a drawing program that supports multiple drawing tools (e.g., pencil, brush, eraser) and multiple drawing surfaces (e.g., canvas, whiteboard, image editor). The Bridge pattern can be used to separate the drawing tools (abstraction) from the drawing surfaces (implementor). This allows users to switch between different drawing tools and surfaces seamlessly without modifying the drawing tool or surface classes directly.
Structural design pattern that allows you to split a large class into two separated hierarchies. It allows for decoupling abstractions from implementations. Promotes code reusability. Facilitates evolution and maintenance.	1 - Abstraction: provides a high-level control logic. It relies on the object to do the low-level work. 2 - Implementation: Interface that is common to all concrete implementations. The abstraction can only communicate with the object by the methods declared in this interface. 3 - Concrete Implementations: Contains platform specific code. 4 - Refined Abstractions: Provides a variant of the control logic. They work with different implementations extending from the abstraction itself. 5 - Client: Class that is only interested with working with the abstraction. Its job is to link the abstraction object with the implementation object.	Open/Close Principle: Introduce new abstractions and implementations easily. Single Responsability Principle: Focus on high-level logic in the abstraction. Create platform-independent classes and apps. Client code works with high-level abstractions, being hidden from the platform details. An example is a TV and its stereos and we can use a Bridge pattern to implement new functionalities into the remote.
Bridge is a structural design pattern that divides business logic or huge class into separate class hierarchies that can be developed independently basically it lets you split the monolithic class into several class hierarchies. This is good because you can change the classes in each hierarchy independently of the classes in the others so it simplifies code maintenance and minimizes the risk of breaking existing code.	The main components are abstraction and implementation. Abstraction (also called interface) is a high-level control layer for some entity. This layer isn't supposed to do any real work on its own. It should delegate the work to the implementation layer (also called platform).	Bridge benefits are huge, it allows you to basically not grow exponential in terms of classes/subclasses allowing you to separate one of the dimensions into other separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.
O padrão de design Bridge é uma técnica que permite separar uma abstração da sua implementação, de modo que ambas possam ser modificadas independentemente uma da outra. O padrão Bridge resolve o problema do acoplamento rígido entre uma abstração e as suas implementações concretas, permitindo que elas evoluam independentemente. Isso simplifica a manutenção do código e aumenta a flexibilidade do sistema.	A Abstração fornece a lógica de controle de alto nível. Ela depende do objeto de implementação para fazer o verdadeiro trabalho de baixo nível. Na implementação, declara-se a interface comum para todas as implementações concretas. A abstração só pode interagir com a implementação através dos métodos aqui declarados. A abstração pode ter métodos similares aos da implementação, mas normalmente declara comportamentos mais complexos que dependem de operações primitivas da implementação. As Implementações Concretas contêm código específico da plataforma. As Abstrações Refinadas oferecem variações para controlar a lógica. Elas interagem com diferentes implementações através da interface geral de implementação, assim como sua classe superior. Normalmente, o Cliente está interessado apenas na abstração. No entanto, cabe ao cliente associar o objeto de abstração a uma das implementações concretas.	O padrão Bridge permite criar classes e aplicações independentes de plataforma, mantendo o código cliente isolado dos detalhes específicos da plataforma. Isso segue os princípios aberto/fechado e de responsabilidade única, pois possibilita a introdução de novas abstrações e implementações de forma independente, facilitando a manutenção e focando na lógica de alto nível na abstração e nos detalhes de plataforma na implementação. Exemplos Práticos: App de desenho, Abstração (funcionalidades para desenhar), Implementação (Forma como os desenhos são mostrados na tela) App de Música, Abstração (Controlo das funções de reprodução (play, pause)), Implementação (Como a música é reproduzida (com altifalantes ou fones de ouvido).
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other. It solves the fact that when creating shapes of different colors you need to create various different new classes.	The Abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work. The Implementation declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here. The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation. Concrete Implementations contain platform-specific code. Refined Abstractions provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface. Usually, the Client is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.	The pros are: - You can create platform-independent classes and apps. - The client code works with high-level abstractions. It isn't exposed to the platform details. - Open/Closed Principle. You can introduce new abstractions and implementations independently from each other. - Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.

O padrão Bridge é projetado para separar a abstração da implementação, permitindo que ambos variem independentemente. Ele soluciona problemas de rigidez em software onde uma mudança leva a uma cascata de alterações em outras partes do código.

A estrutura do padrão Bridge envolve duas hierarquias principais: uma para abstrações e outra para implementações.
Abstração (Abstraction): Define a interface abstrata e mantém uma referência a um objeto do tipo implementador. Pode conter um método que será definido por uma extensão.
Implementação (Implementor): É uma interface que define como a abstração vai interagir com diferentes implementações, que podem variar sem que haja impacto na abstração.
Refinamento da Abstração (Refined Abstraction): Estende a interface definida pela Abstração para casos específicos.
Implementação Concreta (Concrete Implementor): Classes que herdam da interface Implementador e fornecem implementações específicas para os métodos definidos.

Os benefícios incluem maior flexibilidade, promoção do princípio de responsabilidade única e melhor manutenção. Por exemplo, pode-se alterar a UI e os mecanismos de armazenamento de dados de uma aplicação de forma independente usando o padrão Bridge.

The Bridge pattern is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. It achieves this by providing an interface hierarchy (abstraction) and a separate implementation hierarchy. The Bridge pattern allows the implementation and abstraction to vary independently and be extended or modified without affecting each other.
The main problem the Bridge pattern solves is the situation where a class has multiple dimensions of variation, and the variations are independent of each other. In such cases, using inheritance to handle these variations can lead to an explosion of subclasses and complexity. The Bridge pattern addresses this issue by separating the abstraction from its implementation, thus allowing both to vary independently.

The main components of the bridge pattern are: the abstraction, the implementation, the concrete implementations, the refined abstraction and the client. The abstraction provides high-level control logic, it relie on the implementation object to do the actual low-level work; the implementation declares the interface that's common for all concrete implementations, an abstraction can only communicate with an implementation object via methods that are declared here, concrete implemnetation contain platform-specific code; refined abstractions provide variants of control logic ans the client is only interested in workinh with the abstraction

Decoupling Abstraction from Implementation - The Bridge pattern separates the abstraction (interface) from its implementation, allowing them to vary independently
Open/Closed Principle - The Bridge pattern follows the Open/Closed Principle, allowing new abstractions and implementations to be added without modifying existing code
Reduced Complexity - By breaking down a large class hierarchy into smaller, more manageable components, the Bridge pattern reduces complexity and makes the codebase easier to understand and maintain
Platform Independence - The Bridge pattern is particularly useful for developing platform-independent software.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

The Abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work.

The Implementation declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

Concrete Implementations contain platform-specific code.

Refined Abstractions provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

Usually, the Client is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

You can create platform-independent classes and apps.
The client code works with high-level abstractions. It isn't exposed to the platform details.
Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.

The bridge pattern separates an interface from its implementation, putting them in seperate hierarchies.

Abstraction - the interface
Implementor - the implementation seperate from the interface

It allows for more flexibility with implementations of the interface, since they are not obligated to implement every single method

O padrão Bridge é usado para separar a abstração da implementação, permitindo que ambos variem independentemente.
Ele resolve o problema de ter muitas de classes causado pela herança tradicional ao lidar com várias dimensões de variação. Para resolver esse problema é introduzindo dois níveis de abstração: a abstração, que define a interface de alto nível, e a implementação, que fornece as implementações concretas.

Para o funcionamento independente da abstração e da implementação são implementados 4 componentes principais:

Abstraction: Interface de alto nível usada pelos clientes, mantendo uma referência à implementação.

Refined Abstraction: Extensões da abstração que fornecem variações ou melhorias específicas. É opcional.

Implementor: Interface para as classes de implementação, permitindo variação independente da abstração.

Concrete Implementation: Classes que fornecem implementações específicas da interface Implementor.

Vantagens:
-Podemos criar classes e aplicações independentes de plataforma.

-O código cliente trabalha com abstrações em alto nível. Ele não fica exposto a detalhes de plataforma.

-Princípio aberto/fechado. Você pode introduzir novas abstrações e implementações independentemente uma das outras.

-Princípio de responsabilidade única. Podemos focar na lógica de alto nível na abstração e em detalhes de plataforma na implementação.

Exemplo pratico:
Por exemplo o padrão Bridge resolve a complexidade em sistemas de eletrônicos, como o controle universal de diversos dispositivos (TVs, rádios). Sem o Bridge, tínhamos uma complicada hierarquia de classes para cada dispositivo e controle. Com o Bridge, separa-se a interface do controle remoto (abstração) da lógica de controle dos dispositivos (implementação), permitindo que operem independentemente. Isso é feito através de abstrações (Controle Remoto) e implementações concretas (TV, Rádio), simplificando a extensão de funcionalidades sem afetar o outro lado, como adicionar novos dispositivos ou controles.

O padrão Bridge é um padrão de design estrutural que separa uma abstração de sua implementação, permitindo que ambas variem independentemente. Ele resolve o problema de ter uma hierarquia de classes com múltiplas dimensões de variação.

- Abstraction fornece controlo lógico de alto nível
- Implementation declara a interface comum para todas as implementações completas.
- Concrete Implementations contém código especifico da plataforma
- Refined Abstraction forence variantes do controlo lógico
- Cliente que apenas está interessado em trabalhar com a Abstraction

Poder criar classes independentes da plataforma
O cliente apenas trabalha com abstrações de alto nível
Respeita, da mesma forma que o Adapter, o Open/Closed Principle e Single Responsibility Principle

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies which can be developed independently of each other.
It helps reducing the complexity level of classes

Bridge:
1. Abstration - provides high-level control logic. It relies on the implementation object to do the actual low-level work.
2. Implementation - declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.
3. Concrete Implementations - contain platform-specific code.
4. Refined Abstractions - provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface. (Optional)
5. Client - is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

You can create platform-independent classes and apps.
The client code works with high-level abstractions. It isn't exposed to the platform details.
Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.

Bridge é um structural pattern que tem como objetivo dividir uma grande classe (ou estrutura hierárquica) em duas estruturas mais simples. Se tivermos duas classes (ou famílias de classes) que são caracterizadas por duas propriedades fechadas distintas, com n e m valores possíveis respetivamente, ao invés de termos uma estrutura com n x m possíveis formas, podemos ter duas estruturas diferentes onde cada uma é caracterizada por apenas das propriedades.

O bridge pattern tem um cliente que trabalha com a abstração, esta é um controlo de alto nível que precisa da implementação do objeto para trabalhar o low-level. A Implementation é uma interface que é comum para todas as implementações concretas. Existem também abstrações redefinidas que dão variantes na lógica de controlo.

Benefícios do Bridge Pattern:

- Permite criar classes e aplicações independentes da plataforma.
- O código do cliente trabalha com abstrações de alto nível. Não está exposto aos detalhes da plataforma.
- Princípio Aberto/Fechado: Pode introduzir novas abstrações e implementações independentemente umas das outras.
- Princípio da Responsabilidade Única: Pode focar na lógica de alto nível na abstração e nos detalhes da plataforma na implementação.

Quanto a um exemplo prático, possuimos o seguinte caso: Temos um remote control que pode controlar diferentes dispositivos, como televisões e rádios. O remote control possui métodos para ligar/desligar, ajustar o volume e trocar de canal. Para permitir que este funcione com diferentes tipos de dispositivos, implementamos o padrão Bridge. Isso separa a lógica de controlo do remote control (abstração) da implementação específica de cada tipo de dispositivo (implementação).

The Bridge design pattern allows you to separate the abstraction from the implementation.
It decouples the Abstraction and the Implementation, allowing them to evolve independently.

The main components are the abstraction and the Implementation
The abstraction defines the interface with which the client interacts, while the implementation provides the concrete realization of the abstraction. The two parts are then connected to each other via a bridge object.

The advantages of a bridge are that abstraction and implementation be decoupled. The implementation is also changed dynamically at run time and the extensibility of abstraction and implementation is improved.

Bridge é um padrão estrutural que permite separar uma abstração de sua implementação, possibilitando que ambas sejam desenvolvidas de forma independente. Isso é feito através da encapsulação de uma classe de implementação dentro de uma classe de interface, permitindo que o código cliente interaja apenas com a parte de abstração, sem se preocupar com a parte de implementação.
O problema que o padrão Bridge resolve no design de software é a necessidade de desacoplar uma abstração de sua implementação para que ambas possam variar independentemente. Isso é particularmente útil quando a abstração e a implementação precisam ser desenvolvidas e estendidas de forma independente, e quando a implementação precisa ser selecionada em tempo de execução, em vez de ser vinculada em tempo de compilação.

A implementação do padrão Bridge geralmente envolve:
Abstração: Uma interface ou classe abstrata que define a interface para a abstração.
Implementador: Uma interface ou classe abstrata que define a interface para a implementação.
Implementação Concreta: Implementa a interface do implementador, fornecendo a implementação concreta.
Abstração Refinada: Uma classe concreta que estende a abstração e delega algumas de suas operações para a implementação concreta.

- Criar classes e aps independentes da plataforma.
- O código cliente trabalha com abstrações de alto nível. Não será exposto aos detalhes da plataforma.
- É possível introduzir novas abstrações e implementações independentes uma da outra.
- É possível focar em lógica de alto nível na abstração e em detalhes da plataforma na implementação.

O padrão Bridge é um design estrutural que ajuda a separar a abstração da sua implementação de modo a que possam variar independentemente

-Abstraction: Define uma interface/classe abstrata
- Implementor: fornece uma interface independente da plataforma para implementar as funcionalidades específicas.
-Concrete Implementation: Implementa a interface definida pelo Implementor
-Refined Abstraction: Estende a abstração básica

Desacoplamento da abstração da implementação, melhor manutenção e maior flexibilidade, facilita a independência da plataforma

Bridge pattern is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies.
Imagine you have the class shape and you have the circle and square, and now you can implement colors, you would get 4 different combinations, but if you add another color or shape you would exponentially grow the number of objects, so you create a separation and now you have a shape that bridges "contains" the color.

The abstraction: Provides high level control logic.
The implementation: Declares an interface that is common to all the concrete implementations.
Concrete implementations: Contains platform's specific codes.
Refined abstraction: Provide variants of control logic.
Client: Usually it's only interested in working with abstraction.

You can create platform independent classes and apps
The client code works with high-level abstraction, it isn't exposed to the platform detail.
Follows the open closed principle, because you can introduce new abstractions and implementations independently from each other.
It also follows the single responsibility principle, because you can focus on high level logic in the abstraction and on platform details in the implementation.
Imagine we have a device class that acts as the implementation, and remote is the abstraction, you can develop the remote control classes independently from the device classes. A basic remote control might only have two buttons, but you could extend it with additional features, such as an extra battery or touchscreen.

Para criar implementações alternativas para uma classe uma hipótese é criar subclasses. No entanto, esta solução não é escalável, aumentando a sua complexidade exponencialmente com o número de variáveis envolvidas. O padrão Bridge permite a divisão de uma classe ou um conjunto de classes relacionadas em duas hierarquias: abstração e implementação, para que ambas possam ser desenvolvidas de forma independente. A proposta deste padrão é deixar a herança e apostar na composição, permitindo assim separar as dimensões em duas hierarquias de classes distintas.	A Abstração fornece lógica de controlo de alto nível. Ele depende do objeto de implementação para realizar o trabalho real de baixo nível. A Implementação declara a interface que é comum para todas as implementações concretas. Uma abstração só pode se comunicar com um objeto de implementação por meio de métodos declarados aqui. A abstração pode listar os mesmos métodos da implementação, mas geralmente a abstração declara alguns comportamentos complexos que dependem de uma ampla variedade de operações primitivas declaradas pela implementação. Implementações concretas contêm código específico da plataforma. Abstrações Refinadas fornecem variantes de lógica de controle. Assim como seu pai, eles trabalham com diferentes implementações por meio da interface geral de implementação. Normalmente o Cliente só está interessado em trabalhar com a abstração. No entanto, é função do cliente vincular o objeto de abstração a um dos objetos de implementação.	-Podemos criar classes e aplicativos independentes de plataforma. -O código do cliente funciona com abstrações de alto nível. Não está exposto aos detalhes da plataforma. -Princípio Aberto/Fechado (Open/Closed). Podemos introduzir novas abstrações e implementações independentemente umas das outras. -Princípio da Responsabilidade Única. Você pode se concentrar na lógica de alto nível na abstração e nos detalhes da plataforma na implementação. Exemplo: Sistema de notificações que pode enviar mensagens através de diferentes canais, como e-mail ou SMS. Mas o sistema tem que ser flexível para adicionar novos canais no futuro sem alterar o código existente. Assim, se você quiser adicionar um novo canal de notificação, como notificações push, simplesmente cria-se uma nova implementação para esse canal (por exemplo, EnviarPush) que implementa a interface CanalDeNotificacao(interface comum a envio por sms e e-mail). A classe de notificação não precisa ser alterada, apenas a nova implementação é adicionada.
Extract one of the dimensions into a separate class hierarchy, so that the original classes will reference a object of the new hierarchy, instead of having all of its state and behaviors within one class. Without the bridge pattern, the more combinations are possible, the more complex the class would be	The implementation declares an interface that is common to all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.	It complies with the Single Responsibility Principle and the Open/Close Principle. The client code works with high level abstractions, and isn't exposed to the platform details
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies.	Abstraction class: provides high level control logic and relies on the implementation object to the actual low level work. Implementation interfaces: declares the interface that's common for all concrete implementation. Concrete implementation: contains platform specific code. Refined abstractions: provide variance to control logic, they work with different implementation via the general implementation interface.	You can create platform independent classes and apps; client code works with high level abstractions and it is not exposed to the platform details; Single Responsibility Principle; Open/Closed Principle. A practical example would be when you need to extend a class several orthogonal dimensions and another example would be if you need to be able to switch implementations at run time.
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other. Problem: imagine you have a class with two subclasses. if you want to extend this class to add a new feature, you have to create new subclasses that incorporate the existing subclasses with each one of the new features. This way, adding new features will make the hierarchy grow exponentially. solution: The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.	1 - The Abstraction: provides high-level control logic. It relies on the implementation object to do the actual low-level work. 2 - Implementation: Declares the interface that's common for all concrete implementation. An abstraction can only communicate with an implementation object via methods that are declared here. 3 - Concrete implementations: contain platform-specific code. 4 - Refined Abstractions: provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface. 5 - Client: usually the client is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.	It allow us to create platform-independent classes and apps, the client node works with high-level abstractions. It also allow us to introduce new abstractions and implementations independently from each other, and focus opn high-level logic in the abstraction and on platform details in the implementation. For example, a furniture factory that produces different types of furniture, with Bridge pattern we can separate the types of furniture from the materials (of the furniture), avoiding the explosion of the class hierarchy. Another example could bel for a program that allows combinations of elements, and bridge will allows you to create new combinations of elements without changing the existing code.
O Bridge pattern é um padrao que permite dividir uma classe em diferentes hierarquias - abstração e implementação, que podem ser desenvolvidos independentemente para tornar mais simples o seu desenvolvimento.	Vamos ter uma classe abstrata que esta dependente da interface da implentação, esta vai ter todos os metodos comuns á implementação concreta.	Os beneficios sao: poder criar classes independentes da plataforma, o codigo do cliente funciona com alto nivel de abstração, seguindo "Open/Closed Principle." Pois introduz novas abstracoes e implementacoes independete entre elas, e o "Single Responsibility Principle" pois se foca na logica de alto nivel na abstração e nos detalhes da plataforma na implementação.

O padrão Bridge é um padrão de design estrutural que tem como objetivo desacoplar uma abstração da sua implementação, de forma que possam variar independentemente. Isto é "fazendo uma ponde " entre a abstração e a implementação.

O probelema é muitas vezes nos deparamos com sistemas que precisam ser extendidos em duas dimensões independentes. Por exemplo, você pode ter uma aplicação que precisa trabalhar com diferentes sistemas de banco de dados e, ao mesmo tempo, suportar vários formatos de relatório. Uma abordagem comum seria estender a aplicação em duas hierarquias de classes separadas - uma para cada dimensão de extensão. No entanto, isso rapidamente leva a uma explosão do número de classes, tornando o sistema difícil de entender e manter.

O padrão Bridge permite separar a interface de utilizador da lógica de implementação através de uma "ponte", possibilitando que ambos os lados evoluam independentemente sem se afetarem, aumentando a flexibilidade e facilitando a manutenção do código

Esse padrão esta dividido em:

Abstraction: É uma interface abstrata que mantém uma referência a um objeto do tipo Implementador. A Abstração define uma interface de alto nível que irá interagir com o cliente.

Refined Abstraction: Uma ou várias implementações da interface Abstração que refinam e estendem a interface inicial. Essas classes estendem a Abstração, fornecendo formas específicas de implementar a funcionalidade definida na Abstração.

Implementor: Define a interface para as classes de implementação. Esta interface não precisa corresponder exatamente à interface da Abstração; no entanto, deve ser capaz de realizar as operações necessárias que a Abstração precisa. O objetivo principal é fornecer a base para a implementação concreta.

Concrete Implementor: Classes concretas que herdam ou implementam a interface Implementador. Essas classes fornecem implementações específicas das operações definidas na interface Implementador. Cada Implementação Concreta corresponde a uma implementação específica da abstração, que pode variar independentemente dela.

O objetivo do padrão Bridge é separar a abstração da sua implementação, de modo que os dois possam variar independentemente, o que permite que o código seja mais flexível e reutilizável

Você pode criar classes e aplicativos independentes de plataforma.
O código do cliente funciona com abstrações de alto nível. Não está exposto aos detalhes da plataforma.
Princípio Aberto/Fechado . Você pode introduzir novas abstrações e implementações independentemente umas das outras.
Princípio da Responsabilidade Única . Você pode se concentrar na lógica de alto nível na abstração e nos detalhes da plataforma na implementação

Exemplo um aplicativo que organiza eventos e quer notificar os usuários sobre atualizações importantes

Componentes do Padrão Bridge
Notification: Interface de notificação que declara o método notifyUser.
EventNotification: Implementa Notification para enviar notificações sobre eventos.
MessageSender: Interface que define como a mensagem será enviada.
EmailSender, SMSSender: Envia mensagens via e-mail ou SMS.

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

The Abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work.
The Implementation declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.
The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.
Concrete Implementations contain platform-specific code.
Refined Abstractions provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface. These are OPTIONAL.
Usually, the Client is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.

Allows to divide and organize monolithic classes that have several variants of some functionality into hierarchies, which in turn allows to change classes in each hierarchy independently of the other classes. Also allows to make different implementations of the same class used by others while following the same patterns as previous.

É um padrão de desenho estrutural que permite dividir uma classe grande ou um conjunto de classes relacionadas em duas hierarquias separadas - a abstração e implementação, que podem ser desenvolvidas independentes uma da outra.

Estrutura: Realizar a ponte entre o nível de abstração e implementação entre conceitos e famílias de objetos interligados. Permite desenvolver as partes da implementação independentes umas das outras.
Componentes: Uma interface comum para todas as implementações concretas. A abstração apenas comunica com implementações da interface através de métodos definidos na própria interface. Temos também classes concretas derivadas da abstrata e consequentemente derivam métodos da interface de implementação. O client apenas comunica com a classe abstrata.

Permite criar classes independentes da plataforma.
O client code trabalha com abstrações, não é exposto aos detalhes.
Pode-se introduzir mais abstrações e novas implementações independentes uma da outra.
Podemo-nos focar numa lógica de alto nível na abstração e nos detalhes da plataforma na implementação.
Por exemplo: Temos uma interface device que implementa vários métodos relativos a diversos tipos de dispositivos. Uma rádio, tv implementam essa interface. Temos uma classe abstrata Remote que contém um Device e tem métodos abstratos para o controlo de todo o tipo de dispositivos que integram essa interface device. Temos o advanceremote que é uma implementação concreta dessa classe abstrata Remote que pode ter métodos adicionais. O client apenas interage com a classe abstrata Remote.

The bridge pattern attempts to curb complexity by utilizing object composition as opposed to object inheritance, this works because inheritance when done in more than one dimension leads to a combinatorial explosion of implementations which can cause heavy maintenance burden and overall difficult the job of refactoring the code as the requirements evolve.

The structure of the bridge pattern consists of an Abstraction class, and an Implementation interface. The Abstraction class provides high level logic, while the Implementation interface declares the common operations between all the concrete implementations. As usual with interfaces, the abstraction may only communicate with the implementations using the methods declared on the interface. The implementations are included in the Abstraction by means of composition, that is, the Abstraction class contains as an attribute one or more concrete implementations of the Implementation interface. Finally the abstraction class itself can be subclassed by refined abstractions that enhance it according to the evolving requirements.

The bridge pattern allows for the creation of modular, platform-independent software that allows the client code to interface with several high-level abstractions. On top of that, the bridge pattern allows to more easily respect several SOLID principles, namely the Open/Close principle and the Single Responsibility principle.

The Bridge pattern is a structural design pattern that decouples an abstraction from its implementation so that the two can vary independently. It allows the abstraction and the implementation to change and evolve independently without requiring changes to each other. The main problem that the Bridge pattern solves in software design is the "class explosion" issue that arises when you have multiple variations of a class hierarchy. Without the Bridge pattern, you might end up with a large number of class combinations, where each subclass of the abstraction is paired with each subclass of the implementation, leading to a combinatorial explosion of classes.	Abstraction: Provides high-level control logic. It relies on the implementation object to do the actual low-level work. Implementation: Declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here. Concrete Implementations: Contain platform-specific code. Refined Abstractions: Provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.	Separation of concerns, expandability, flexibility, encapsulation, platform independence. Practical example: A household switch controlling lights, ceiling fans, etc.
Bridge is a structural design pattern that allows us to split a large class or a set of closely related classes into two separate hierarchies (abstraction classes and implementation classes), which can be developed independently of each other. It solves the geometric progression of class creation in case of a need to extend the already created classes' functionality.	The Abstraction provides high-level control logic. It relies on the implementation object to do the actual low-level work. The Implementation declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here. The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation. Concrete Implementations contain platform-specific code. Refined Abstractions provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface. Usually, the Client is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.	Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers). You can create platform-independent classes and apps. The client code works with high-level abstractions. It isn't exposed to the platform details. Open/Closed Principle. You can introduce new abstractions and implementations independently from each other. Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.
Padrão que permite a uma abstração ter várias implementações que podem mudar ao longo do tempo. Faz isto dividindo classes complexas em duas hierarquias mais pequenas, a abstração e a implementação	- Abstraction: Interface ou class abstrata que define as operações disponíveis para o cliente - Abstração refinada(opcional): pode haver mais que uma. Extendem a abstração atraves de override ou extendem o comportamento definido na Abstraction - Interface de implementação: interface ou class abstrata que a implementação concreta tem de implementar - Implementação concreta: implementações concretas da interface de implementação. Cada implementação concreta dá formas diferentes de implementar a interface	- Decoupling da interface e da implementação - A abstração e a implentação são independentes, fazendo com que possam existir mudanças em qualquer um dos dois sem que o outro não seja afetado - Single Responsibility Principle. Dá para focar na logica de abstração e nos detalhes da plataforma de implementação
O padrão Bridge permite dividir varias classes que relacionam entre si em duas hierarquias separadas que são abstração e implementação.	A Abstração fornece lógica de controle de alto nível e depende do objeto de implementação para realizar o trabalho de baixo nível. A Implementação declara a interface comum para todas as implementações concretas. Implementações concretas contêm código específico da plataforma. Abstrações Refinadas fornecem variantes de lógica de controle e trabalham com diferentes implementações por meio da interface geral de implementação. E o Cliente só está interessado em trabalhar com a abstração, mas ele também vincula o objeto de abstração a um dos objetos de implementação.	Os benefícios são poder criar classes e aplicativos independentes de plataforma, código do cliente irá funcionar com abstrações de alto nível, pode-se introduzir novas abstrações e implementações independentemente umas das outras e pode-se concentrar na lógica de alto nível na abstração e nos detalhes da plataforma na implementação.
The Bridge pattern is a structural design pattern that aims to decouple an abstraction from its implementation so that the two can vary independently. It achieves this by creating an interface (abstraction) that is maintained independently from the classes implementing it (implementation), thereby allowing the implementation to be changed without affecting the client code. In simpler terms, the Bridge pattern provides a way to separate the abstraction (what) from its implementation (how), allowing them to vary independently. The problem that the Bridge pattern solves in software design is the issue of tightly coupling abstractions with their implementations. Without the Bridge pattern, changes to the implementation often require changes to the abstraction, and vice versa. This can lead to code that is difficult to maintain, extend, and reuse.	The Bridge pattern is a structural design pattern that separates an abstraction from its implementation. This allows for independent variation of both without affecting the other. The main components are Abstraction, Implementation, Concrete Abstraction and Concrete Implementor. Abstraction: This is the interface or abstract class that defines the operations that concrete classes will implement. It acts as the bridge between the client code and the implementation. Client code interacts with the abstraction without needing to know the specifics of the implementation. Implementation (Implementor): This is the interface or abstract class that defines the implementation details of the functionality. Concrete implementor classes will inherit from this interface and provide specific implementations. Concrete Abstraction: This extends the abstraction class and implements the operations using a specific implementor. It creates a bridge between a particular abstraction and a concrete implementation. Concrete Implementor: This implements the implementation interface and provides the actual functionality. There can be multiple concrete implementors offering different ways to carry out the operations.	The Bridge pattern allows the creation of flexible, modular and easy-to-maintain systems. It promotes good software design practices such as abstraction, encapsulation and separation of concerns and can help your code better adapt to changing requirements and future code enhancements. For example, imagine we have Coffee class with 3 subclasses: Latte, Black and Cappuccino. We now want to extend this classs hierarchy to incorporate temperature (hot or cold), so we create Hot and Cold subclasses. However, since we already have 3 subclasses,we'll need to create six class combinations such as HotBlack and ColdLatte. But that's not all. If later we decided to add more coffee types, we would need to create 2 subclasses for each coffee type we add, leading to an unnecessarily high amount of subclasses. This can be avoided by implementing a Bridge service linking these two dimensions through object composition instead of class inheritance, making it substancially simpler.

Para criar implementações alternativas para uma classe uma hipótese é criar subclasses. No entanto, esta solução não é escalável, aumentando a sua complexidade exponencialmente com o número de variáveis envolvidas. O problema prende-se com a extensão das classes em duas dimensões. A proposta deste padrão é deixar a herança e apostar na composição, permitindo assim separar as dimensões em duas hierarquias de classes distintas.	Abstraction: fornece a lógica de controlo de alto nível. Ele depende do objeto de implementação para realizar o trabalho real de baixo nível. Implementation: declara a interface que é comum para todas as implementações concretas. Uma abstraction só pode se comunicar com um objeto de implementação por meio de métodos declarados aqui. A abstraction pode listar os mesmos métodos da implementação, mas geralmente a abstração declara alguns comportamentos complexos que dependem de uma ampla variedade de operações primitivas declaradas pela implementação. Refined Abstractions: fornece variantes da lógica de controle. Assim como seu pai, eles trabalham com diferentes implementações por meio da interface geral de implementação. Client: normalmente só está interessado em trabalhar com a abstração. No entanto, é função do cliente vincular o objeto de abstração a um dos objetos de implementação.	Facilita a extensão de abstrações e implementações de forma independente e separa a interface da implementação, permitindo mudanças em ambos os lados sem afetar o outro. Exemplo: Facilitar a comunicação com diferentes tipos de dispositivos (impressoras, scanners) através de uma interface comum, enquanto permite variações específicas de cada dispositivo.
É um padrão de design extrutural que permite você dividir uma grande classe ou um grupo de classes relacionadas em duas hierarquias separadas (abstração e implementação) que podem ser desenvolvidas independente uma da outra. Resolve o problema de criação de várias classes que se inter-relacionam.	Abstração: Providencia controle de lógica de alto nível. Depene do objeto "Implementação" para fazer o trabalho de baixo nível. Implementação: Decclara a ainterface que é comum para todas as implementações completas. Implementações concretas: Contém o código específico da plataforma. Abstrações refinadas: Concede variações de lógica de controle. Como seus pais (Abstraction) trabalham com diferentes implementações via interface de implementação. Cliente: Trabalha apenas com a abstração. É trabalho do cliente linkar o objeto abstração com um dos objetos implementação.	Pode criar classes e aplicativos independete de plataforma. O código cliente funciona com abstrações de alto nível. Você pode focar na lógica de alto nível na abstração e em detalhes da plataforma na implementação. Temos as classes mãe: Shape e color e queremos relacioná-las, ao invés de termos 4 classes com as possíveis variações, fazemos apenas duas classes com Bridge.
the bridge pattern allows the abstraction and implementation to be developed independently, with this the client code can access only the abstraction without the implementation	The abstraction, is the core of the pattern and contains the implementer; The refined abstraction, is an extension of the abstraction, it also hides some elements from the implementer; The implementer, is used to define the interface for the implementacion class, since we use abstractions it is not needed for this interface to be a 1 to 1 comparison with the abstract interface. The implementacion, is the implementation of the implementer.	This pattern is based on the phrase "prefer composition over inheritance", with that it offers more independence and decoupling between classes, which results in a more horizontal dependency tree instead of the usual vertical tree. This can be used for example in the creation of different objects, we can take ikea for example, without the bridge we separate each product and each product with each way of production and each way to assemble, with the bridge we separate the possible products with a overaching product abstract class, and another, for example, workshop abtract class, with this we can diferentiate the products under products and under workshop the components necessary and the way to assemble
O padrão Bridge, em software design, consiste na divisão de classes "grandes" ou de um conjunto de classes semelhantes e relacionadas em duas hierarquias diferentes, a abstração e a implementação, permitindo que sejam desenvolvidas de forma separada. Desta forma permite facilitar a expansão de classes seguindo uma estratégia de herança, criando classes abstratas que são posteriormente usadas para evitar a criação de subclasses em número desnecessário.	A estrutura está equipada com uma classe Abstraction que fornece um controlo lógico de alto nível e liga-se a uma interface chamada Implementation que declara os métodos comuns a todas as implementações concretas, a abstração só pode comunicar com esta interface através dos métodos declarados, estes componentes constroem a bridge que permite a conexão entre os vários objetos de código.	As vantagens fornecidas são a criação de classes e aplicação independentes da plataforma, o código do cliente trabalhar com altos níveis de abstração, não sendo exposto aos detalhes. Para além disso respeita o Open/Closed Principle e o Single Responsibility Principle. Um exemplo prático é, tendo uma classe Forma com várias subclasses como retângulo verde e retângulo azul, podemos criar uma classe Cor contida na forma, de modo a facilitar a expansão de novas formas (circulo, triângulo), sem precisarmos de criar mais subclasses para cada cor.
O padrão Bridge é um padrão de design estrutural que desacopla uma abstração de sua implementação, de modo que as duas possam variar independentemente. É usado para separar uma grande classe ou um conjunto de classes relacionadas de perto em duas hierarquias separadas.	Abstraction, é a interface ou classe abstrata que define a abstração; Implementation, é a interface ou classe abstrata que define a implementação;	Benefícios: desacoplamento, flexibilidade e reutilização; Exemplo: um aplicativo de editor gráfico que suporta diferentes formatos de arquivo (por exemplo, PNG, JPEG, BMP), precisa ser capaz de ler e escrever esses formatos, mas os detalhes específicos de implementação de cada formato são diferentes.
O Bridge é um padrão estrutural que desacopla a abstração da sua implementação, permitindo que ambas variem. Tudo istopara fazer com que a ligação entre a abstração e a sua implementação não seja tão rígida trazendo flexibilidade e permitindo reutilizar código.	Os componentes principais são: a abstração, algo de mais alto nível, a implementação, que completa a funcionalidade fornecendo os detalhes concretos.	Este padrão traz alguns benefícios, nomeadamente a desacopulação já que separa a abstração da implementação, a flexibilidade pois permite que tanto a extensão como a abstração seja feita sem se afetarem e a reutilização pois permite reutilizar e compartilhar implementações entre abstrações. Um exemplo disto é numa aplicação multiplataforma (Windows e Linux). Estes sistemas têm systecalls diferentes e, utilizando o padrão Bridge para separar a lógica da aplicação das chamadas específicas, permite que a lógica por trás seja a mesma ainda que a implementação varie de sistema para sistema.
O padrão Bridge é um padrão Estrutural que tem como objetivo separar uma abstração de sua implementação, com isso, pode se desenvolver as classes independentemente umas das outras. É útil quando existem classes numa hierarquia com muitas variações e é necessário desacoplar essas variações para evitar a criação de um grande número de subclasses. Nesse padrão, a herança é preterida em relação à composição.	Abstração: fornece a lógica de alto nível e depende da interface implementação para ser utilizada. Implementação: declara a interface comum e permite a comunicação entre a abstração e as implementações concretas. Implementações Concretas: responsável pelas implementações reais para os métodos definidos na Implementação.	Promove um baixo acoplamento entre as classes, que aumenta a flexibilidade do sistema e facilita a adição de novas funcionalidades. Além disso, permite a reutilização de código e simplifica a manutenção. Criação de Formas Geométricas: desenhar diferentes formas com diferentes atributos. Controle Remoto Universal: criação de um sistema para controlar dispositivos eletrônicos diversos.
O padrão Bridge é um padrão de design estrutural em engenharia de software que visa desacoplar uma abstração da sua implementação, de modo que os dois possam variar independentemente.	O padrão Bridge é estruturado em torno de quatro componentes principais: Abstração, Abstração refinada, Implementador e implementador concreto.	Os benefícios de usar o padrão Bridge no desenvolvimento de software incluem: Flexibilidade e Extensibilidade, Desacoplamento, Reutilização e Redução de classes.