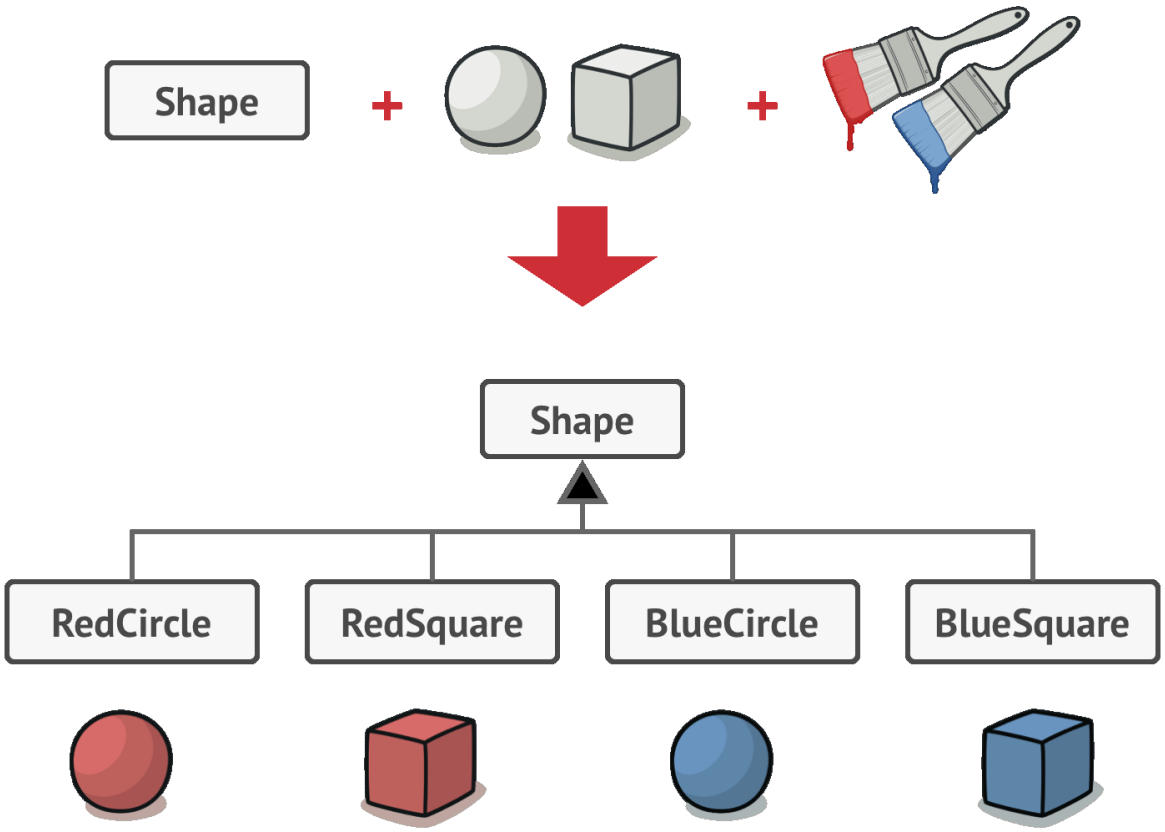
Bridge

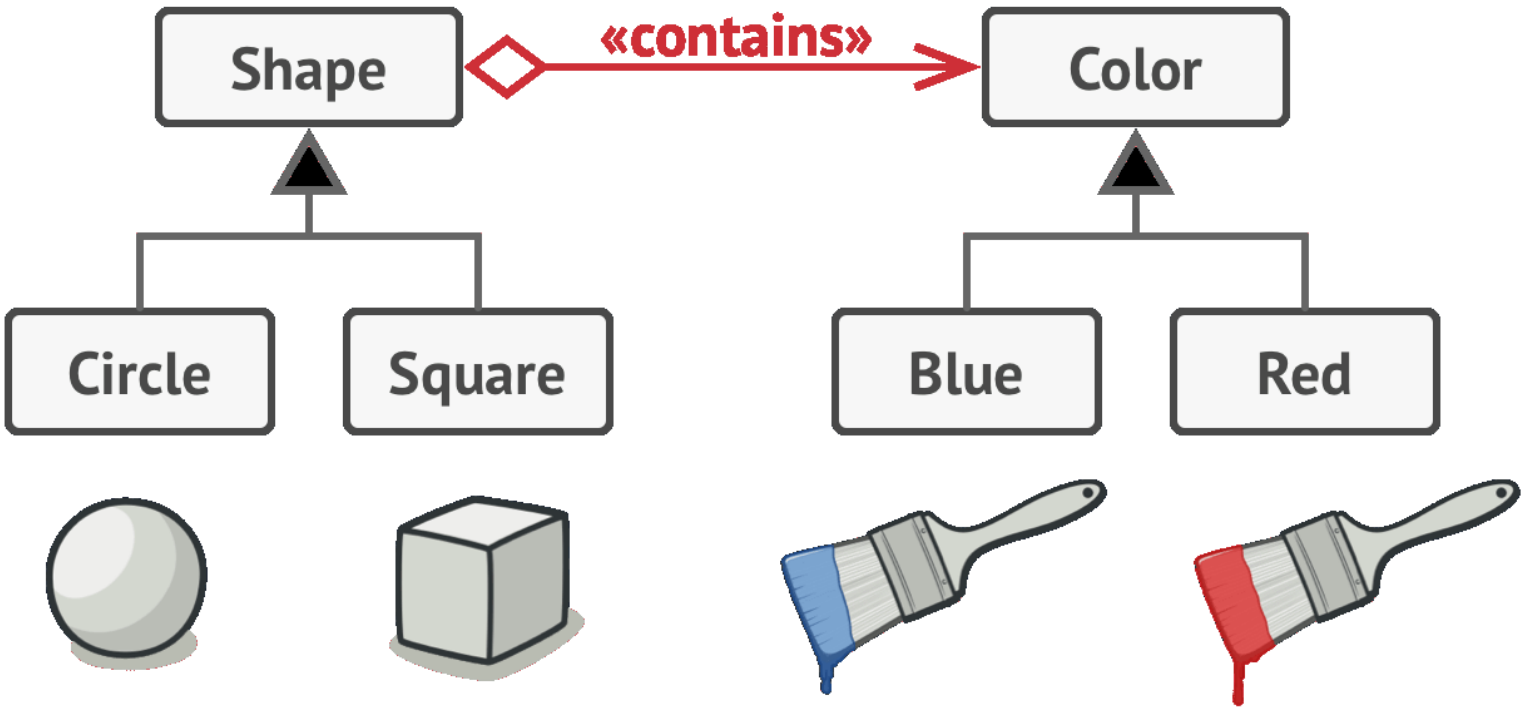
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation which can be developed independently of each other.

Problem



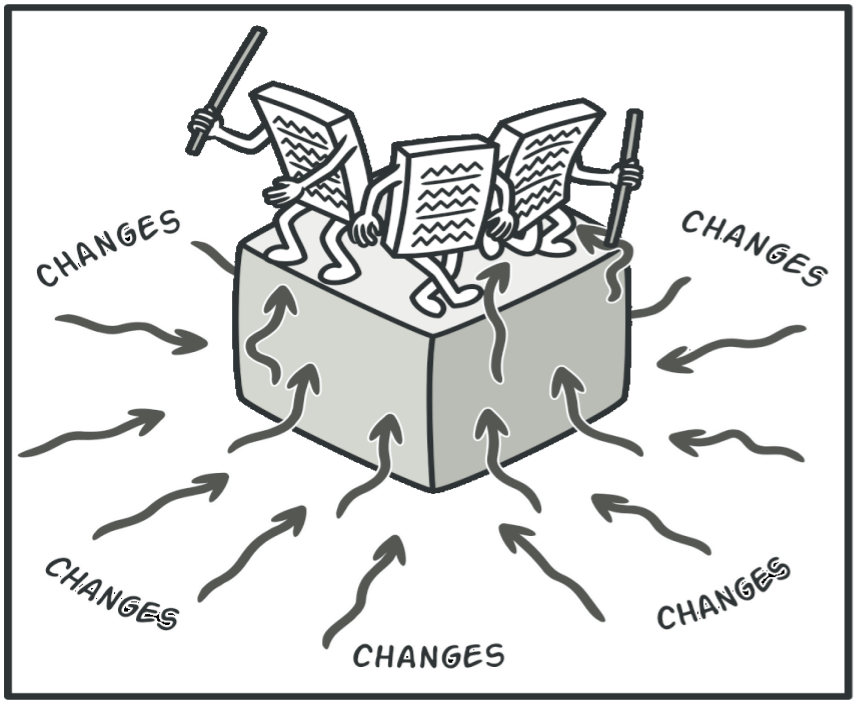
Number of class combinations grows in geometric progression

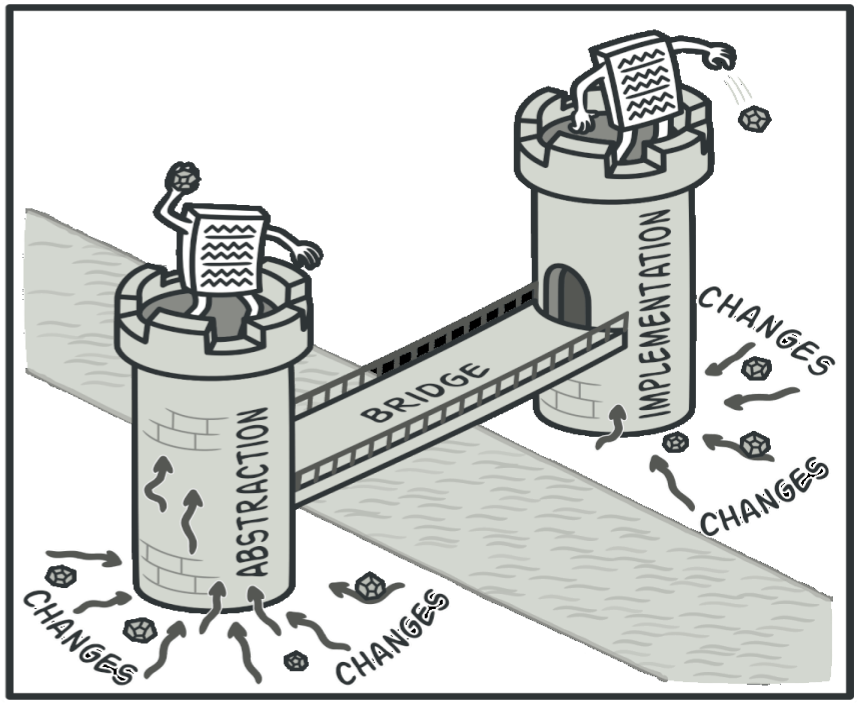
Solution



We can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

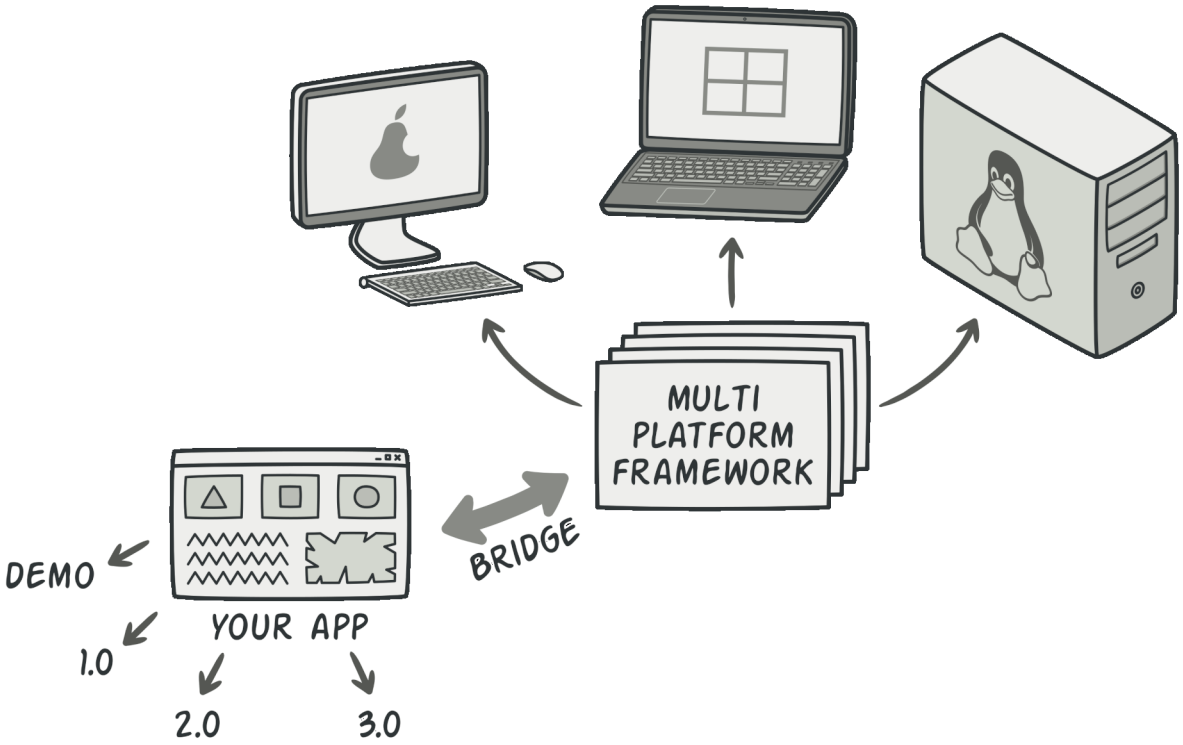
Contd.





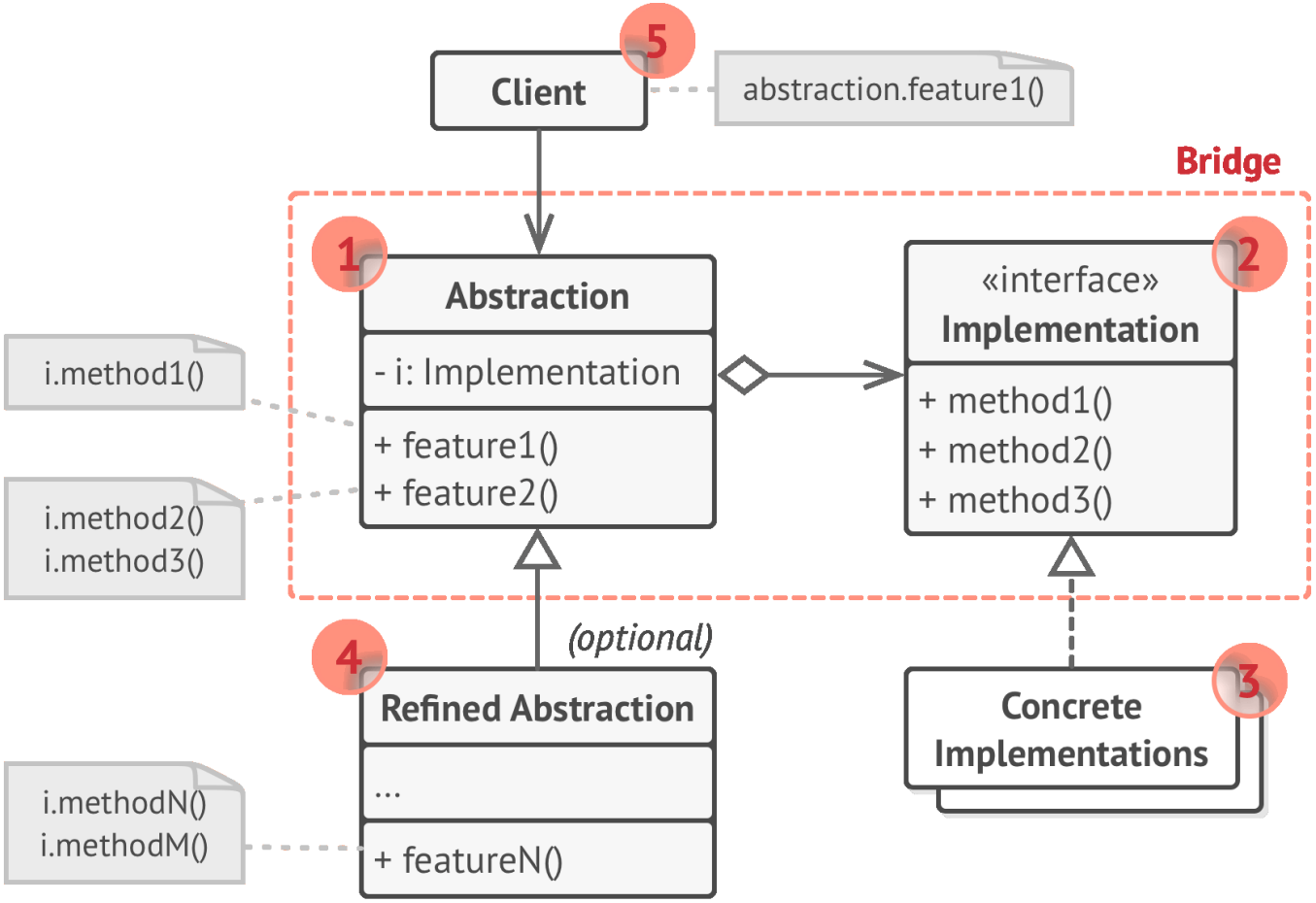
Making even a simple change to a monolithic codebase is pretty hard because we must understand the entire thing very well. Making changes to smaller, well-defined modules is much easier.

Contd.



One of the ways to structure a cross-platform application.

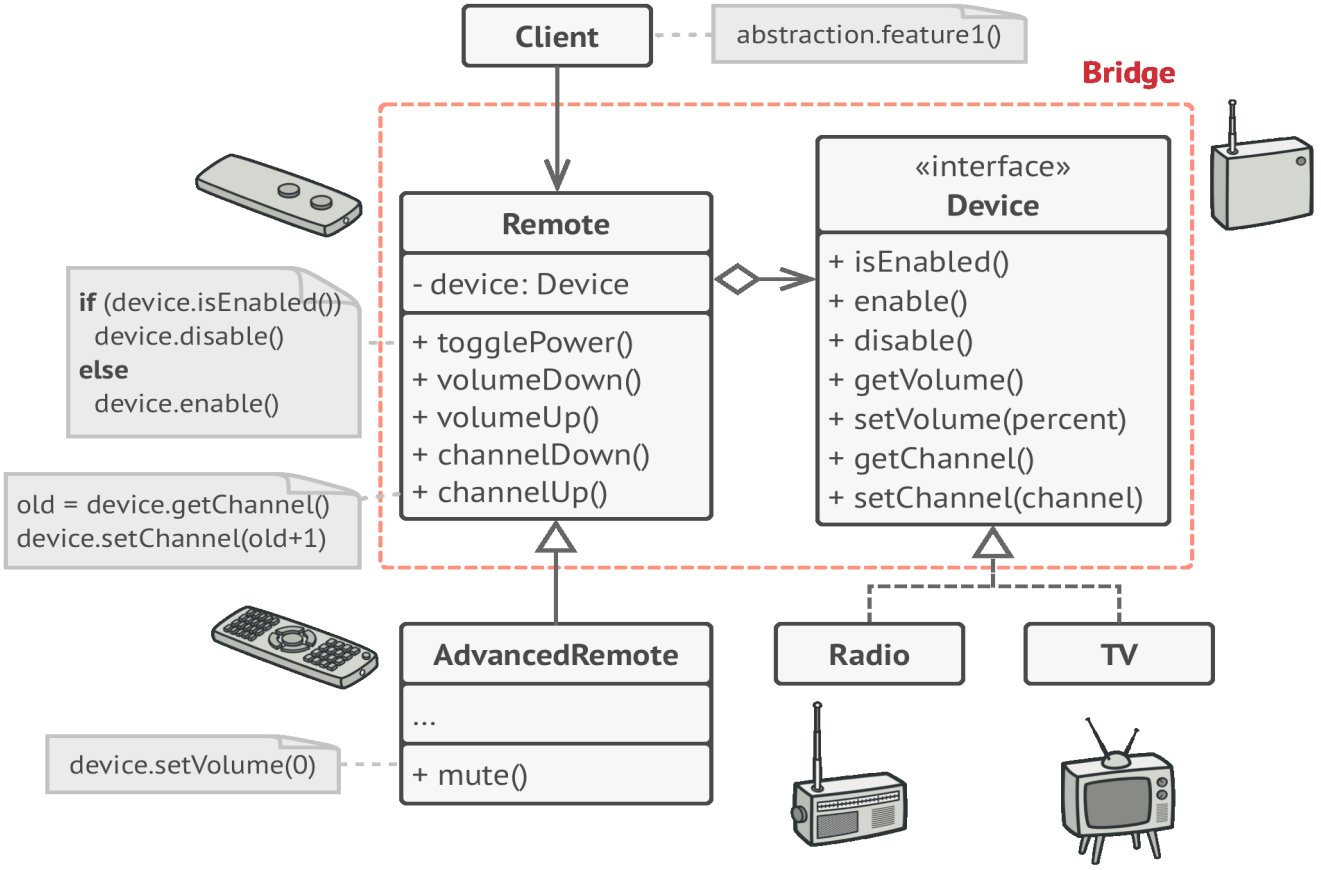
Structure


5
Client
abstraction.feature1()
Bridge
1
Abstraction
- i: Implementation
+ feature1()
+ feature2()
2
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(optional)
4
Refined Abstraction
...
+ featureN()
3
Concrete
Implementations
i.methodN()
i.methodM()

Contd.

- 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.
- **Concerte Implémentations** 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.

Pseudo code



The original class hierarchy is divided into two parts: devices and remote controls.

Applicability

- Use the Bridge pattern when we 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).
- Use the pattern when we need to extend a class in several orthogonal (independent) dimensions.
- Use the Bridge if you need to be able to switch implementations at runtime.

Pros and Cons

- We 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*. We can introduce new abstractions and implementations independently from each other.
- *Single Responsibility Principle*. We can focus on high-level logic in the abstraction and on platform details in the implementation.
- We might make the code more complicated by applying the pattern to a highly cohesive class.