

STRUCTURAL DESIGN PATTERNS



Structural Design Patterns

Structural Design Patterns

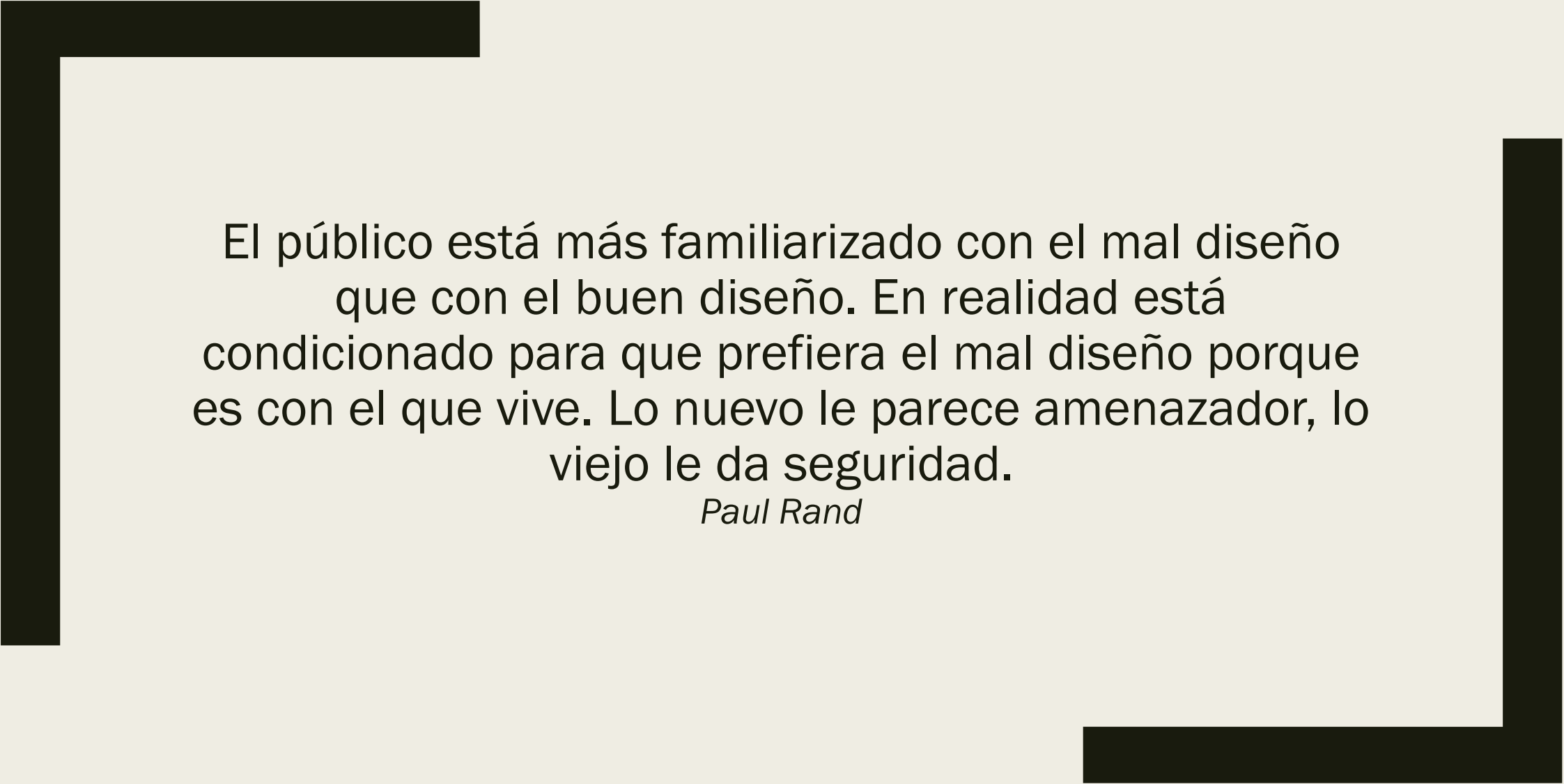
Each of these patterns will help you create a structure for your application architecture in a variety of scenarios.

You will learn how to create composite objects, how to connect to orthogonal class hierarchies together, how to enhance an existing object.

The structural patterns are based on the way in which a set of classes are related to each other to provide a complex functionality, providing a structure to achieve that goal.

Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Proxy



El público está más familiarizado con el mal diseño
que con el buen diseño. En realidad está
condicionado para que prefiera el mal diseño porque
es con el que vive. Lo nuevo le parece amenazador, lo
viejo le da seguridad.

Paul Rand

Adapter



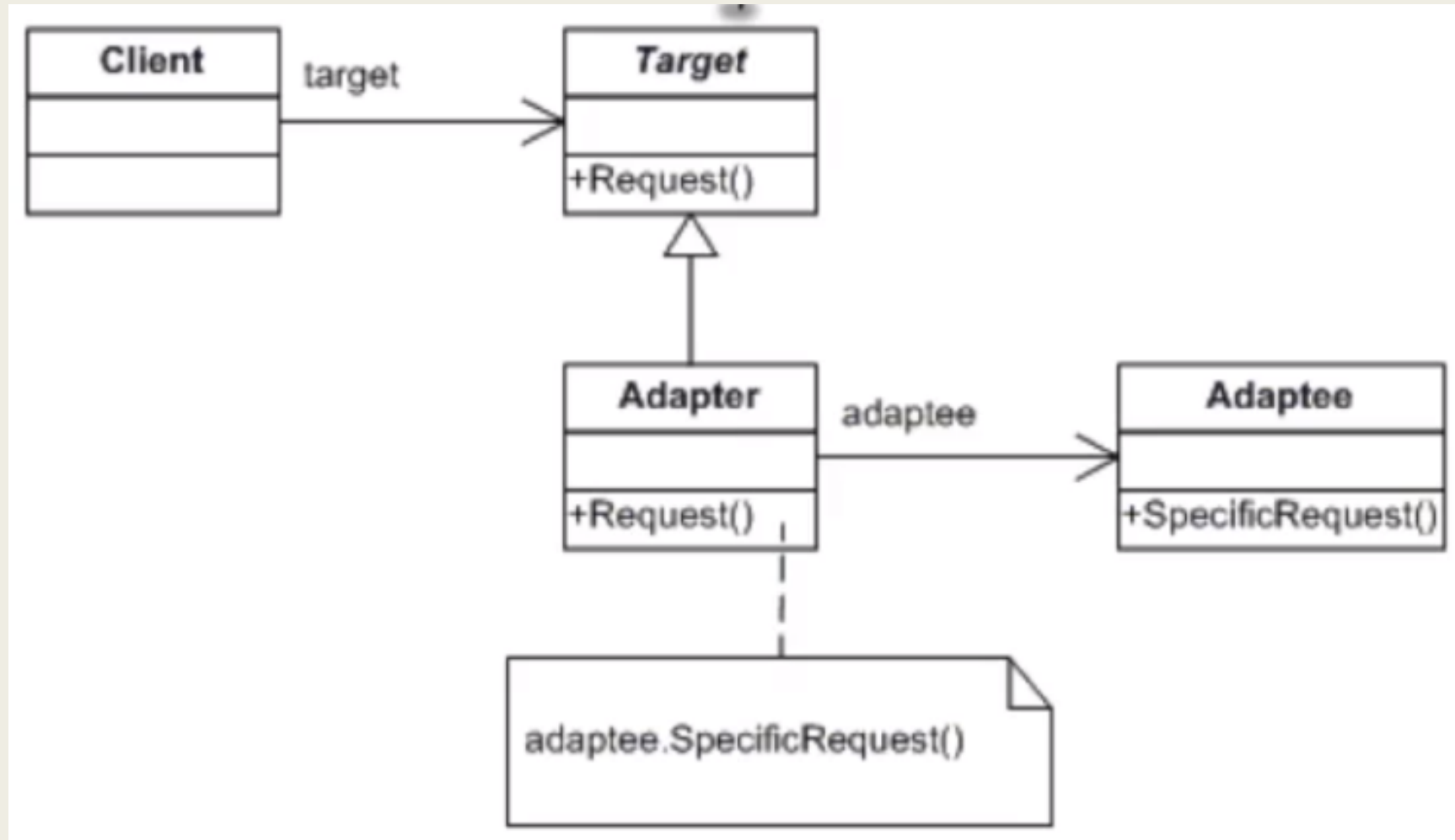
Intent

- Convert the interface of one class into another
- Wrap an existing class with a new interface
- Introduce a legacy component into a new system

Benefits

- Adapters greatly improve code reuse
- You can reuse code across different platforms

UML



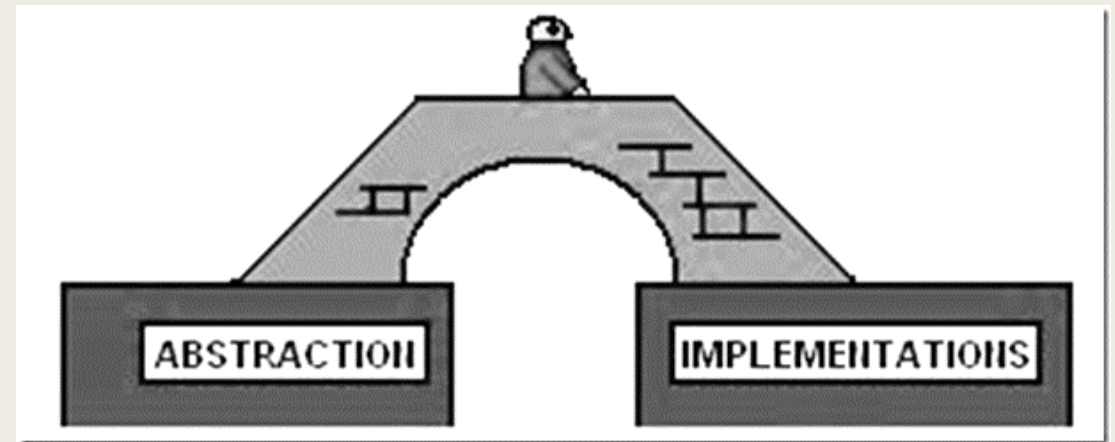
Checklist

- Identify all three actors in this pattern: the client, the adaptee and the adapter.
- Identify the interface that the client requires, and put it in an abstract target class
- Derive a concrete adapter class from the abstract target base class.
- Write the adapter code to map the client interface to the adaptee interface, and place it in the adapter class
- Modify the client to use the adapter every time it needs to access the adaptee

Final Comments

- An Adapter make things work after they are designed, a Bridge make things work before they are designed
- An Adapter expose the same interface, and a Decorator exposes an enhanced interface
- An Adapter changes the interface of an existing object, a Decorator enhances an object without changing its interface.
- A Facade defines a new interface, an Adapter reuses an existing interface.

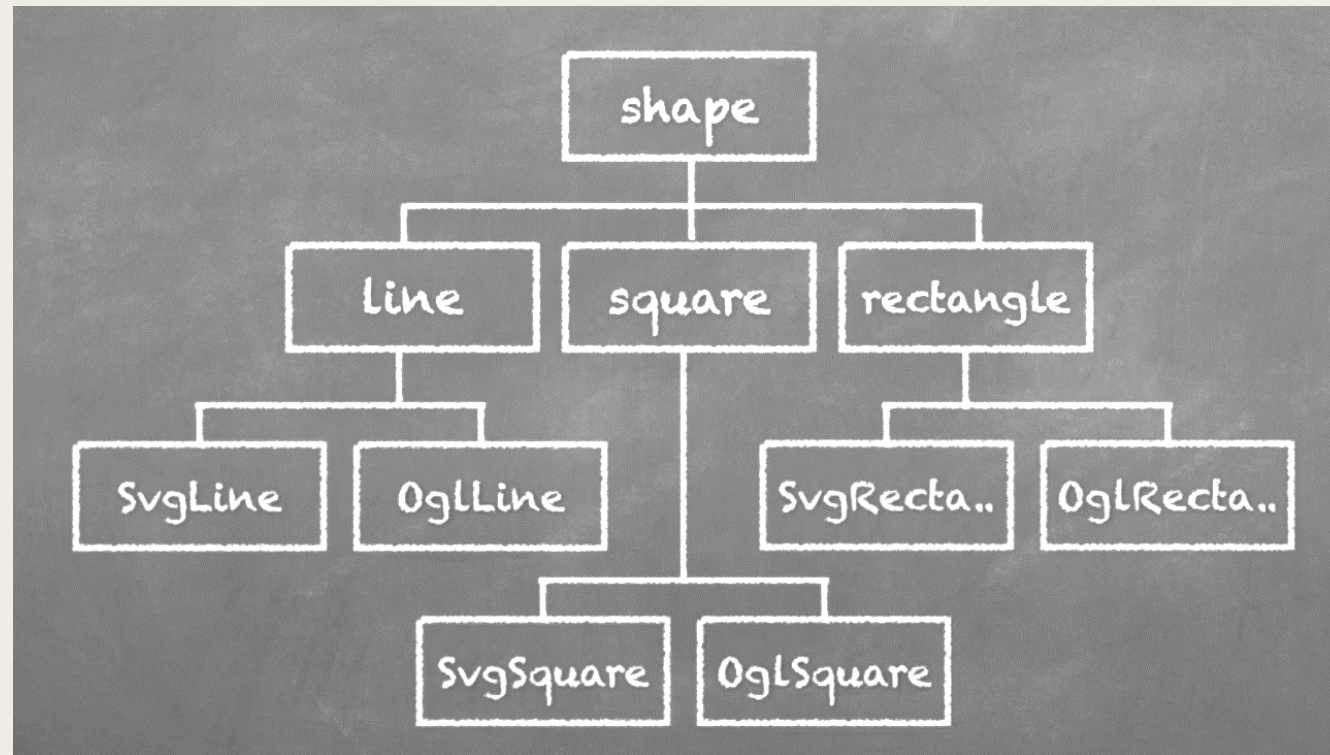
Bridge Pattern



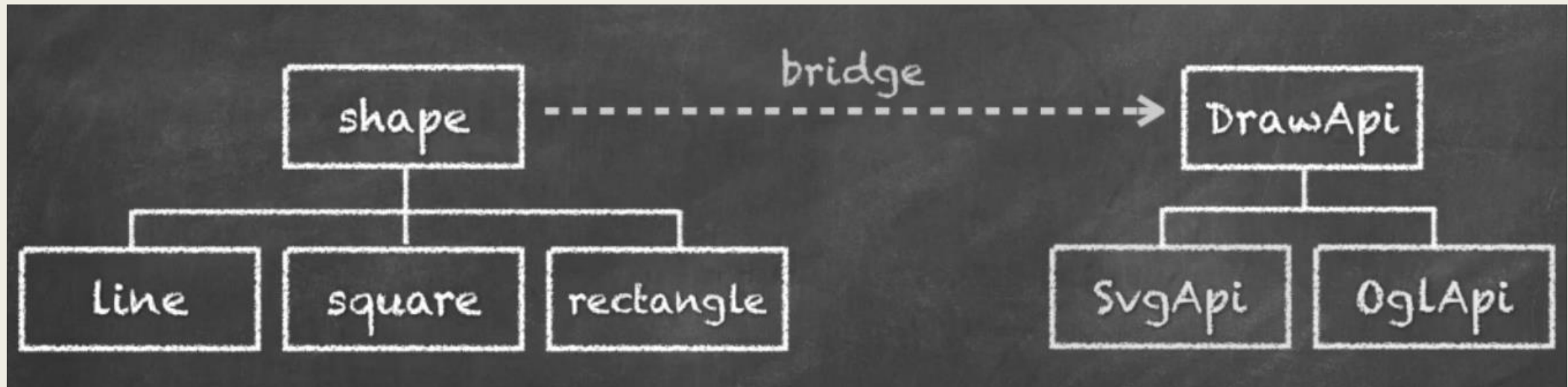
Intent

- Combine two or more orthogonal class hierarchies
- Bind an implementation to a class at runtime
- Clean up a proliferation of classes resulting from an interface coupled with lots of implementations
- Share an implementation among multiple objects

Orthogonal classes hierarchies



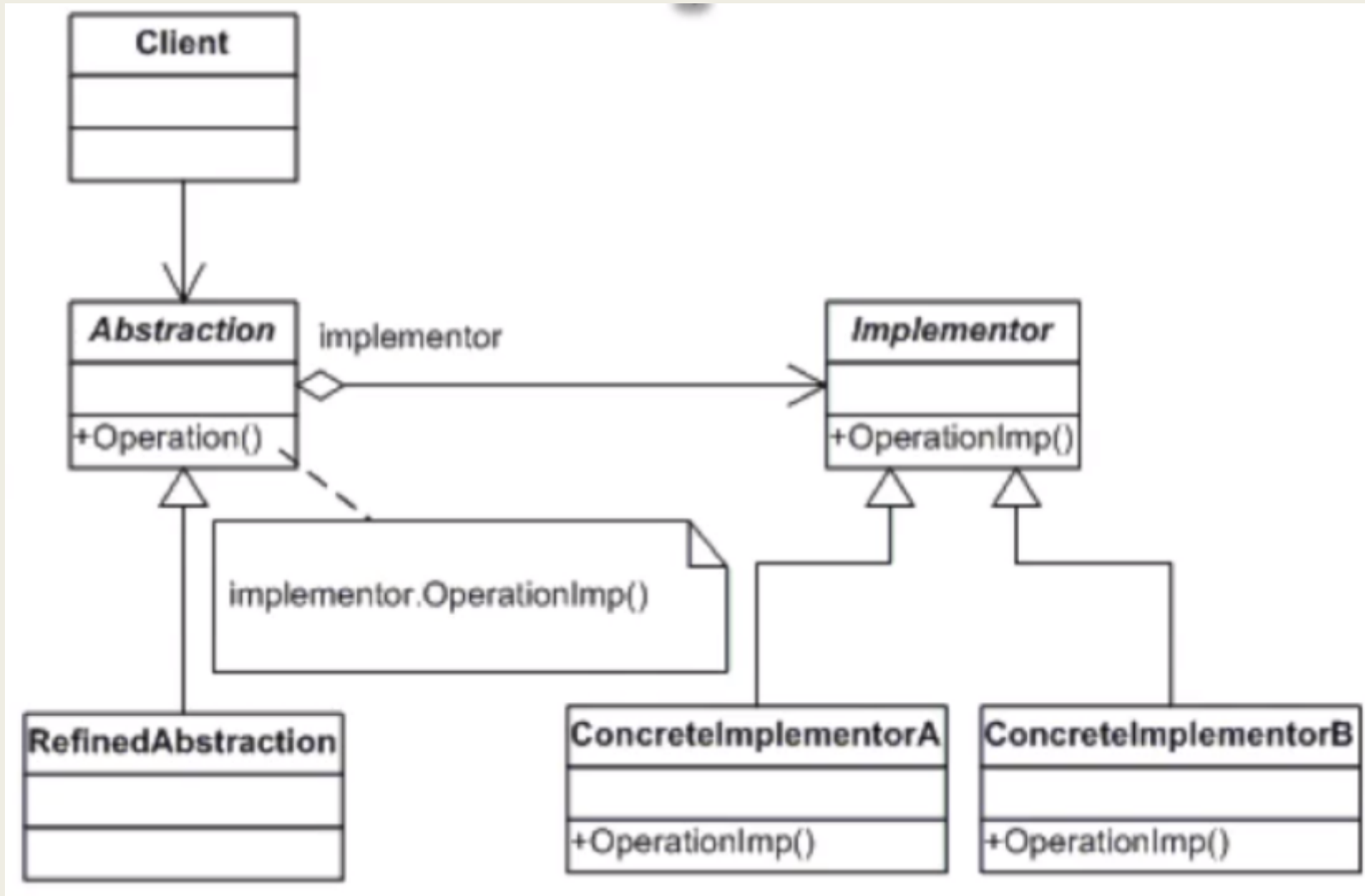
Bridge Pattern



Benefits

- Reduces the number of classes
- Improves code maintainability
- Define an interface at compile time and specify the actual implementation at runtime
- Provides the freedom to make future changes.

UML



Checklist

- Identify two or more orthogonal class hierarchies
- Design an abstract client oriented interface that defines what the client wants
- Design an abstract platform oriented interface that defines what the platform provides.
- Add derived specialization classes for each concrete abstraction and map the client interface to the platform interface.
- Add derived classes for each concrete platform

Final Comments

- An adapter make things work after they're designed, a Bridge makes them work before they are.
- A bridge is designed upfront to let abstraction and implementation vary independently, an Adapter is retrofitted to make unrelated classes work together.
- You can use the Abstract Factory pattern to create implementation objects

Bridge Pattern

- The Bridge pattern lets you combine two or more orthogonal class hierarchies.
- The pattern reduces the number of classes in your code and concentrates all implementation code in its own class hierarchy.
- The pattern provides the freedom to make changes to interface and implementation without breaking the architecture.
- You can provide the implementation at runtime by using an abstract factory to create the implementation objects.
- The pattern is similar to the Adapter pattern but is intended for decoupling instead of adapting.

Composite Pattern

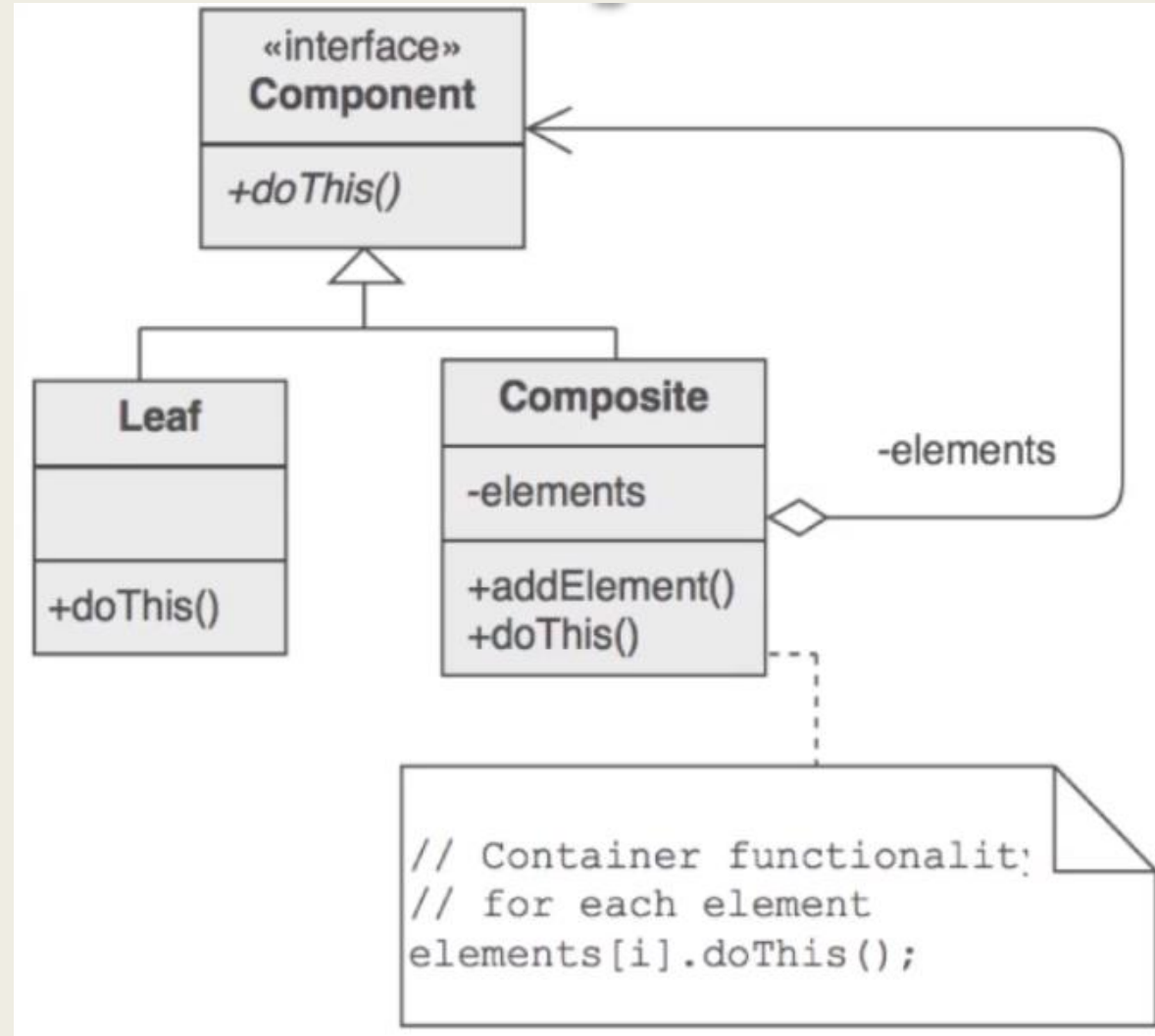
Intent

- You need a tree structure to represent a part of hierarchy
- You want to compose objects out of one or more child objects
- You want clients to treat individual objects and compositions of objects in exactly the same way

Benefits

- Set up a hierarchy of components
- Calling code doesn't need to know which components have children and which do not.
- Ideal if you are applying a sequence of operations on a tree-like data structure and the calling code doesn't care if the operation gets applied to a composite or a leaf object.

UML



Checklist

- Identify all composite objects that are composed of nested child components
- Create an abstract Component base class that contains the lowest common denominator interface
- Add a derived Leaf class and add any leaf management methods to his class.
- Add a derived Composite class and add any child management methods to this class.
- Implement the component API for the Leaf and Composite classes
- Add as many subclasses of Leaf and Composite as you need

Decorator Pattern



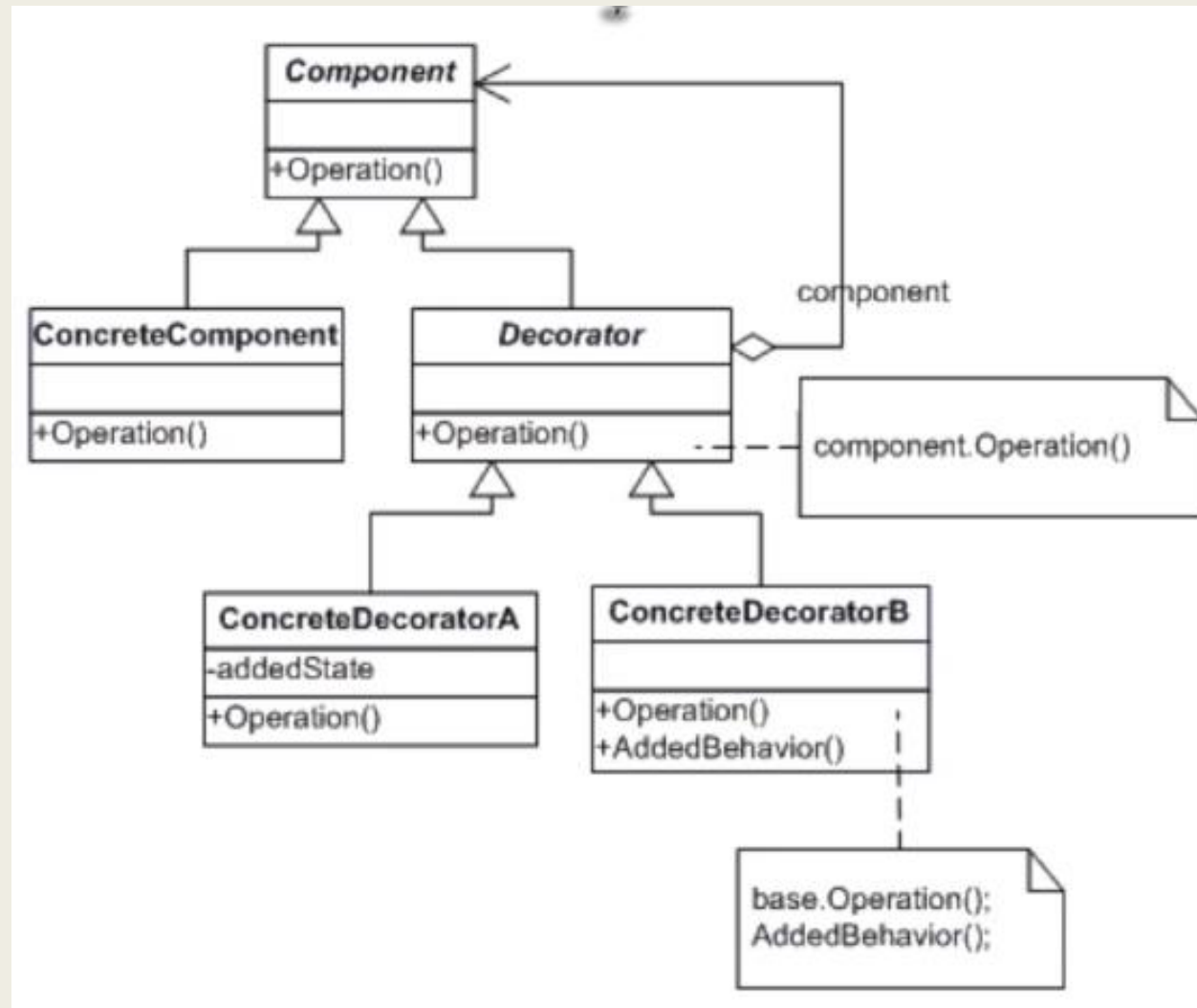
Intent

- Attach functionality to an object dynamically
- You are enable to use subclassing

Benefits

- Allows you to enhance functionality of a method without having to use subclassing
- Side-steps the multiple inheritance limitation in C#
- Hides the original object beneath layers of decorators

UML



Checklist

- Make sure you need to add functionality to an existing class that you cannot subclass and that there is an interface that is common to all.
- Create an abstract class with the lowest common denominator component interface
- Derive the existing class from the component base class
- Define a decorator for each additional piece of functionality
- Client defines type and ordering of component and decorator objects

Final Comments

- You can view Decorator as a Composite with a single child at each level. But a decorator is intended for adding functionality not object aggregation.
- You can combine Decorator and Composite patterns
- An Adapter provides a different interface to its subjects, a Proxy provides the same interface, and a Decorator provides an enhanced interface.

Decorator Pattern

- A good choice when you want to attach additional functionality to an object dynamically and you are unable to subclass the objects
- The pattern treats components and decorators in exactly the same way
- The pattern is ideal if you want to apply layers of enhancements to an object operation.
- You can combine the Composite and Decorator patterns.

Facade Pattern

Intent

- Provide a unified high-level interface to a set of low-level interfaces in a subsystem
- Make the subsystem easier to use
- Wrap a complicated subsystem with a simpler interface

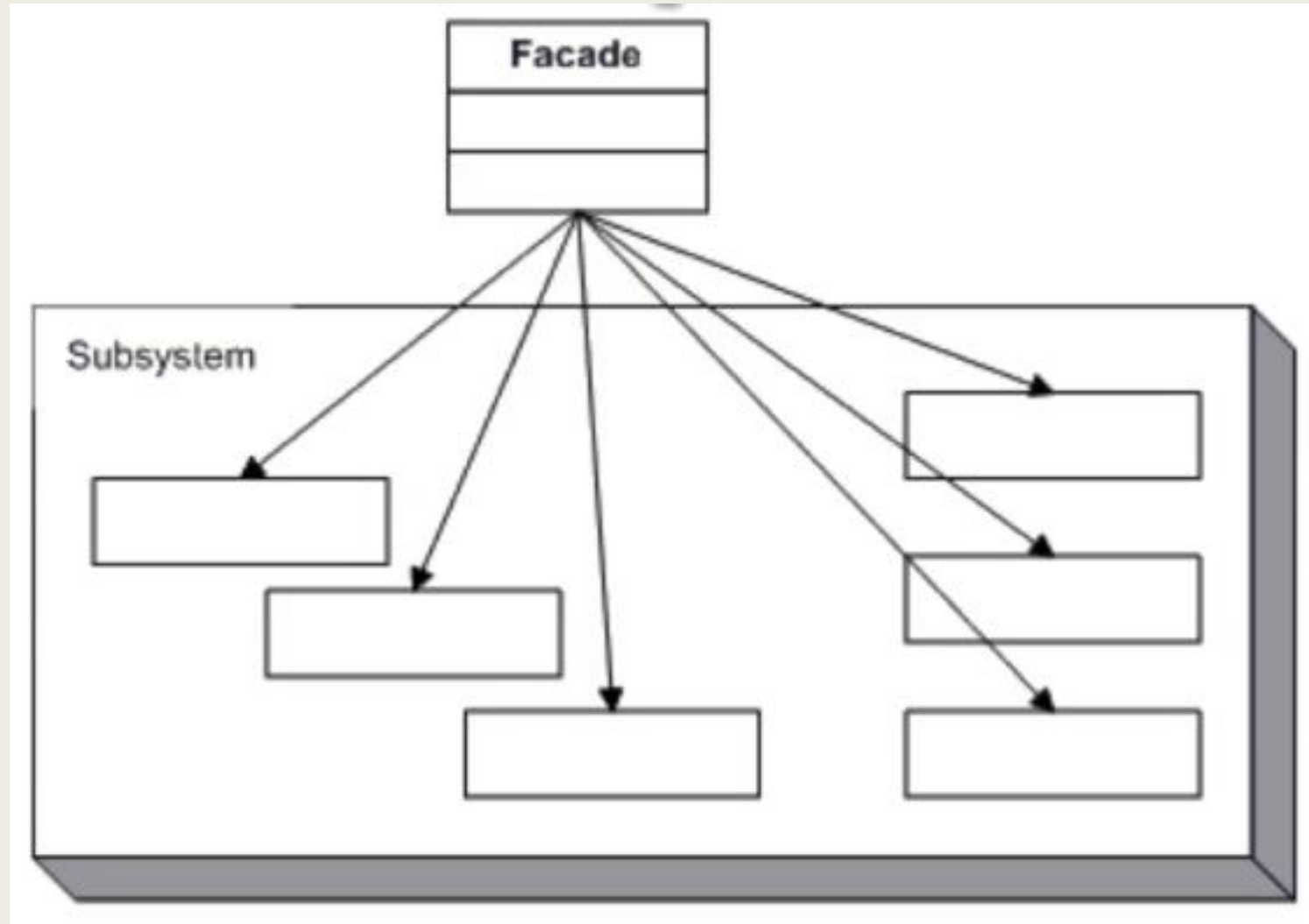
Benefits

- Dramatically reduces the learning curve to successfully interact with the subsystem
- The pattern also promotes the decoupling of the subsystem from its many clients

Risks

- Power and flexibility is gone when the façade is the only access point of the subsystem.
- Freedom to change the subsystem API is significantly reduced if the subsystem remains exposed to clients.
- A faced can turn into the God Object anti-pattern.

UML



Checklist

- Identify a complex subsystem in your application architecture that you want to simplify
- Design a simple unified interface for the subsystem.
- Create a façade class with the simple unified interface
- Have the façade call the appropriate low-level methods to implement the interface
- Modify the client to always use the facade

Final Comments

- A Façade is often a singleton, Adapters are usually instantiated for every component they wrap.
- An Abstract factory is a façade for object creation.

Facade Pattern

- Facade provides a unified high-level interface to a complex low-level subsystem to make the subsystem easier to use
- The pattern promotes decoupling of the subsystem from its clients, but denying clients access to the low-level interface also reduces power and flexibility.
- Facades can evolve into a God Object
- Adapter makes legacy components compatible, Facade reduces the learning curve of a complex subsystem
- Facades are often singletons, and you can sometimes use an Abstract Factory in a place of Facade.