Structural patterns

Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

- Adapter
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adapter

Problem

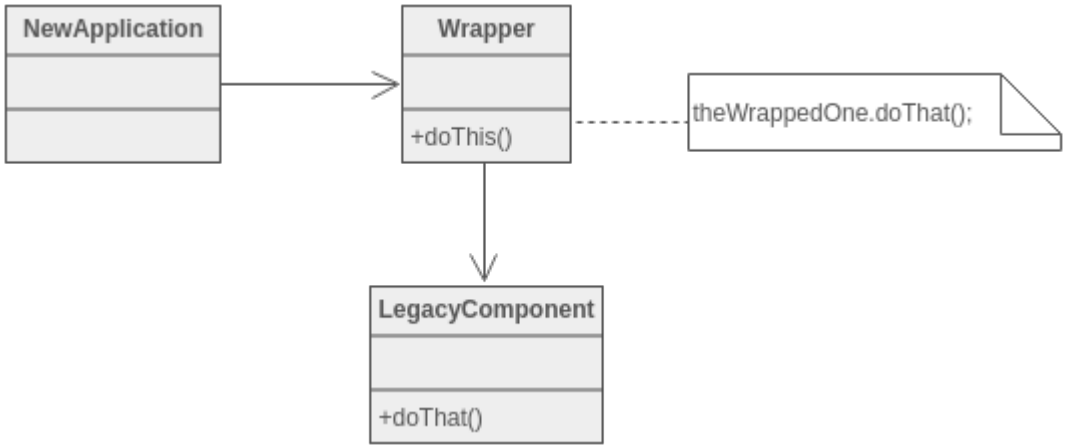
Be able to reuse components with different interface.

Solution

Create an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Diagram


NewApplication
Wrapper
+doThis()
theWrappedOne.doThat();
LegacyComponent
+doThat()

Composite

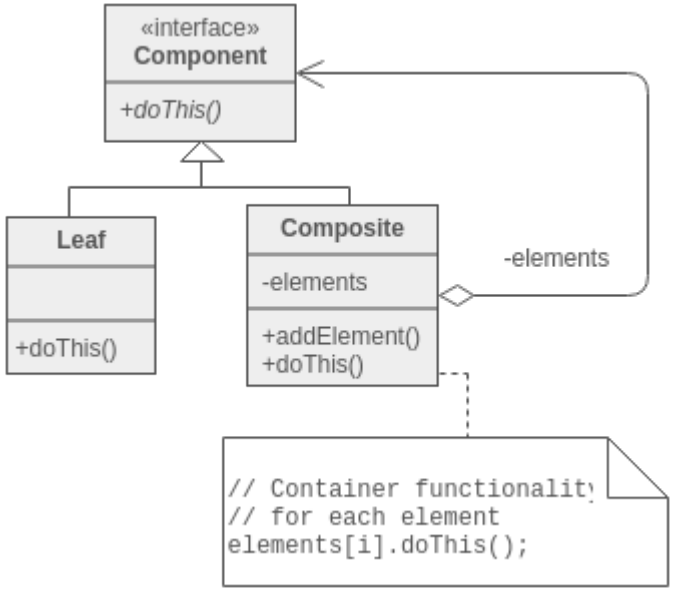
Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Manipulation with the composite objects must be the same as for primitive objects.

Solution

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

Diagram


«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement()
+doThis()
-elements
// Container functionality
// for each element
elements[i].doThis();

Decorator

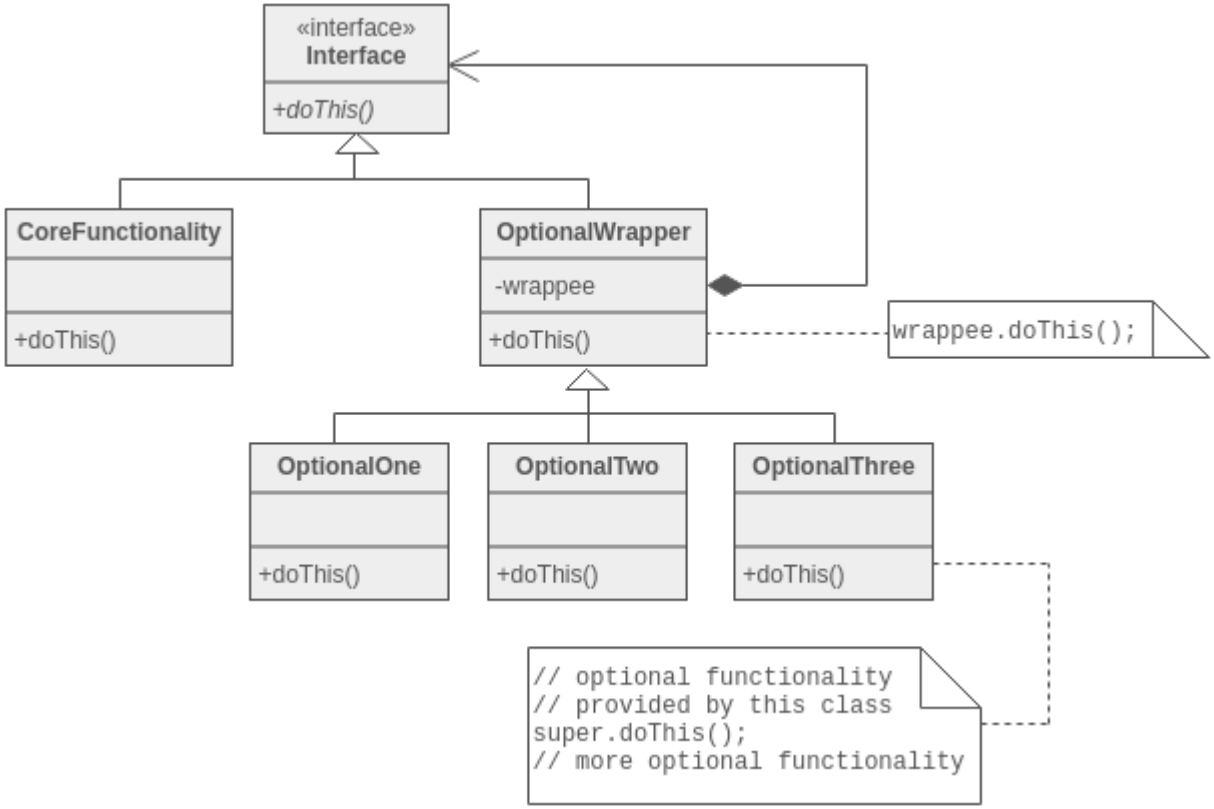
Problem

Dynamically add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Solution

Encapsulate the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Diagram


«interface»
Interface
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
wrappee.doThis();
OptionalOne
+doThis()
OptionalTwo
+doThis()
OptionalThree
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

Facade

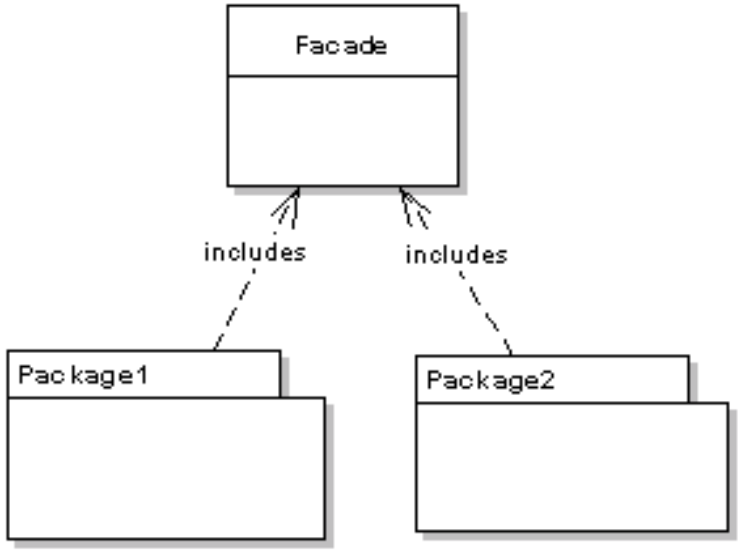
Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Solution

Encapsulate a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

Diagram



Flyweight

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

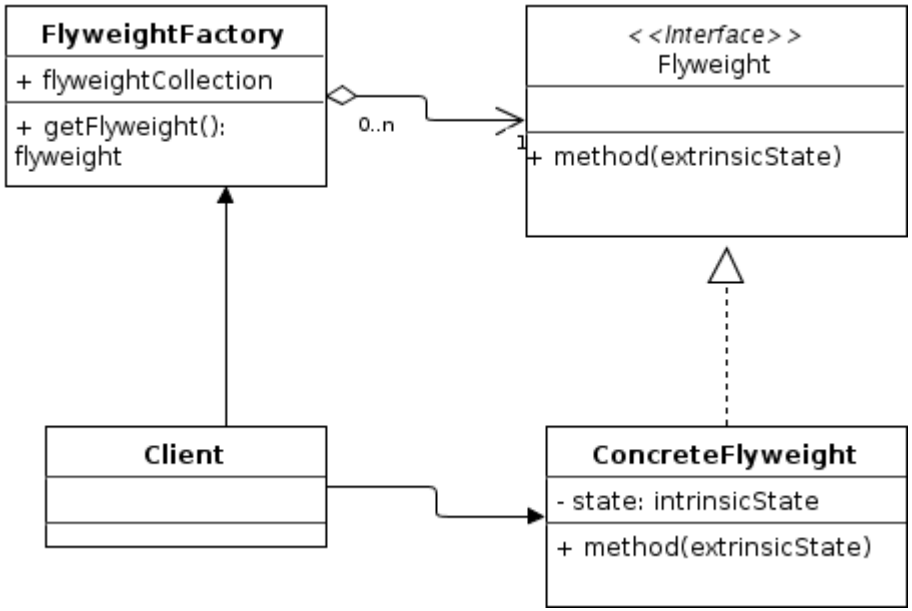
Solution

Divide each "flyweight" object into two pieces:

- the state-dependent (extrinsic) part,
- the state-independent (intrinsic) part.

Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

Diagram


FlyweightFactory
+ flyweightCollection
+ getFlyweight(): flyweight
0..n
1
<<Interface>>
Flyweight
+ method(extrinsicState)
Client
ConcreteFlyweight
- state: intrinsicState
+ method(extrinsicState)

Proxy

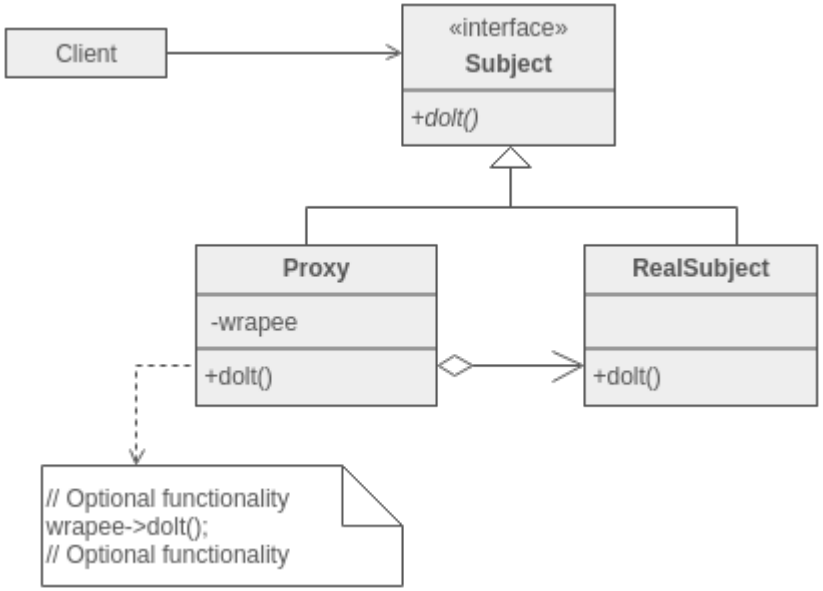
Problem

Hide real object and add additional functionality to methods’ call.

Solution

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Diagram


Client
«interface»
Subject
+dolt()
Proxy
-wrapee
+dolt()
RealSubject
+dolt()
// Optional functionality
wrapee->dolt();
// Optional functionality