## Some patterns

- Memento
- Mediator
- Observer
- Adapter
- Composite
- Decorator
- Abstract Factory
- Object Pool
- Singleton

## Memento

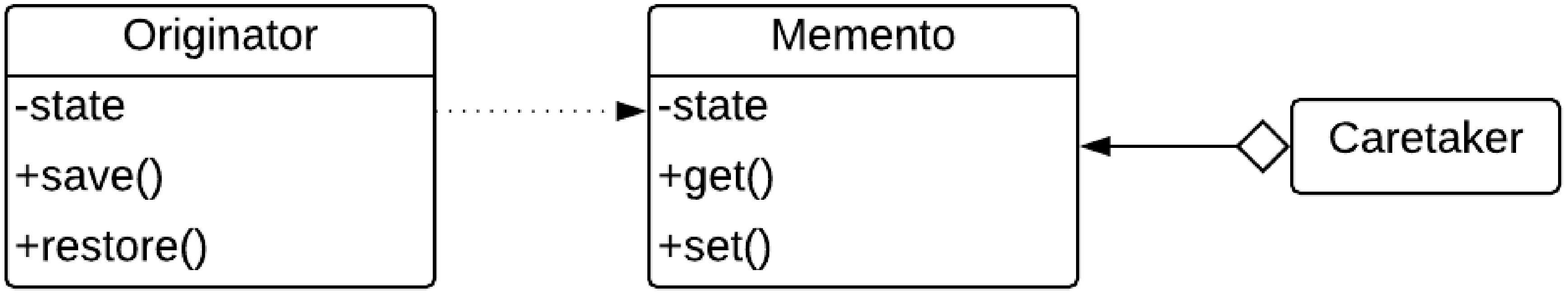
### Problem

Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).

### Solution

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

**Diagram**


Originator
-state
+save()
+restore()
Memento
-state
+get()
+set()
Caretaker

## Mediator

### Problem

Direct interaction between many objects can become very complex (each object must know about all others).

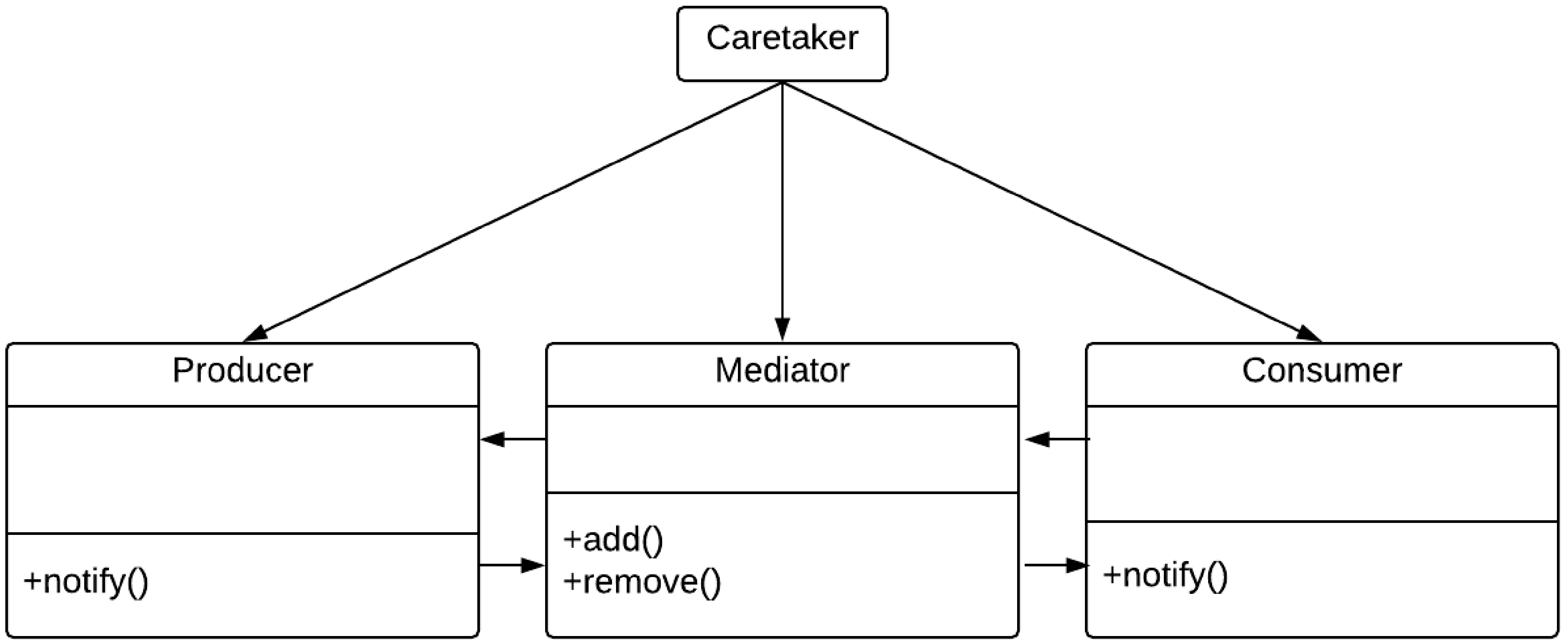
Too many dependencies between classes.

### Solution

Define an object that encapsulates how a set of objects can interact.

Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

**Diagram**


Caretaker
Producer
+notify()
Mediator
+add()
+remove()
Consumer
+notify()

## Observer

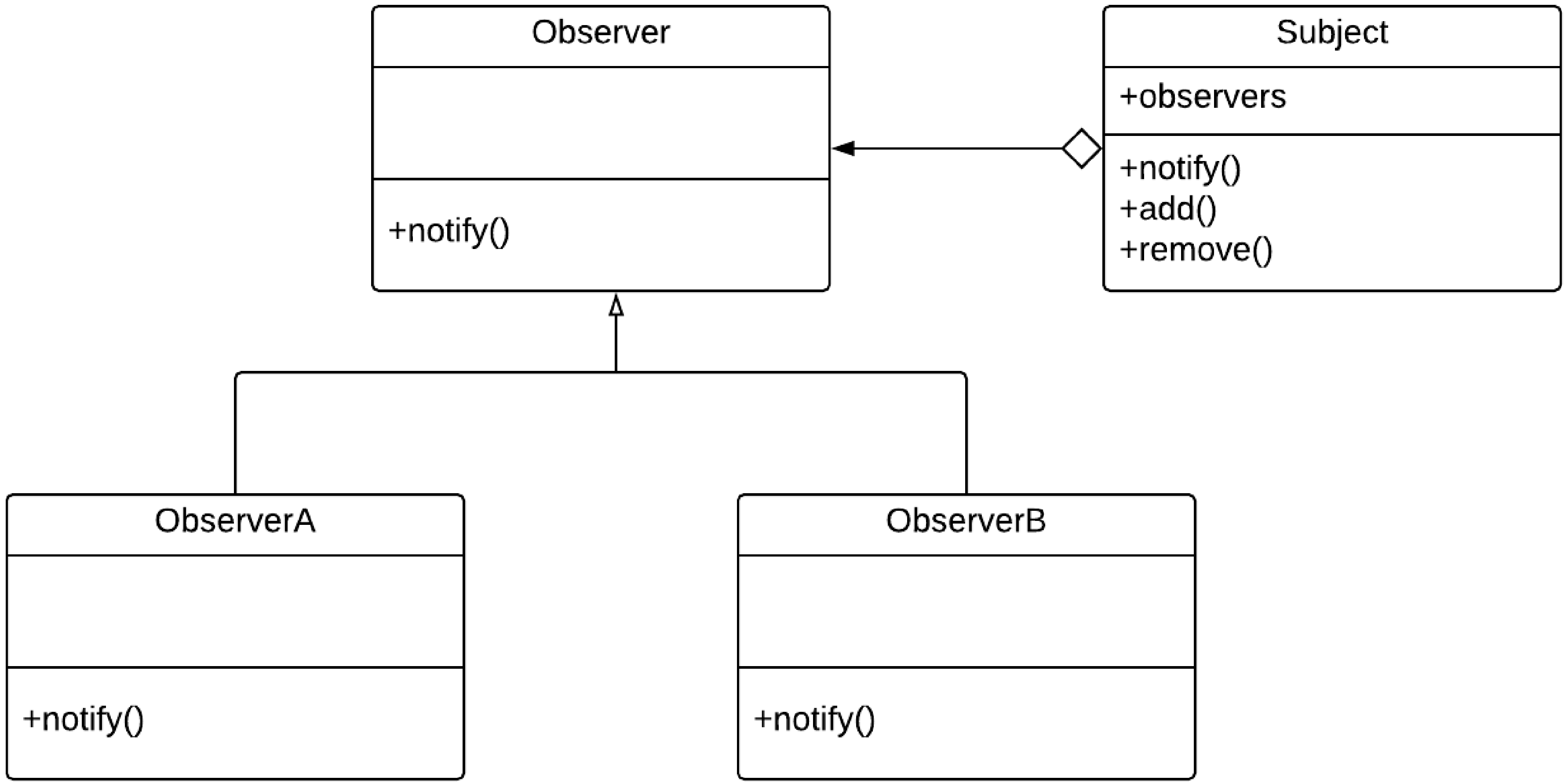
### Problem

Many objects want to “know” about new events.

### Solution

Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

**Diagram**


Observer
+notify()
Subject
+observers
+notify()
+add()
+remove()
ObserverA
+notify()
ObserverB
+notify()

## Adapter

### Problem

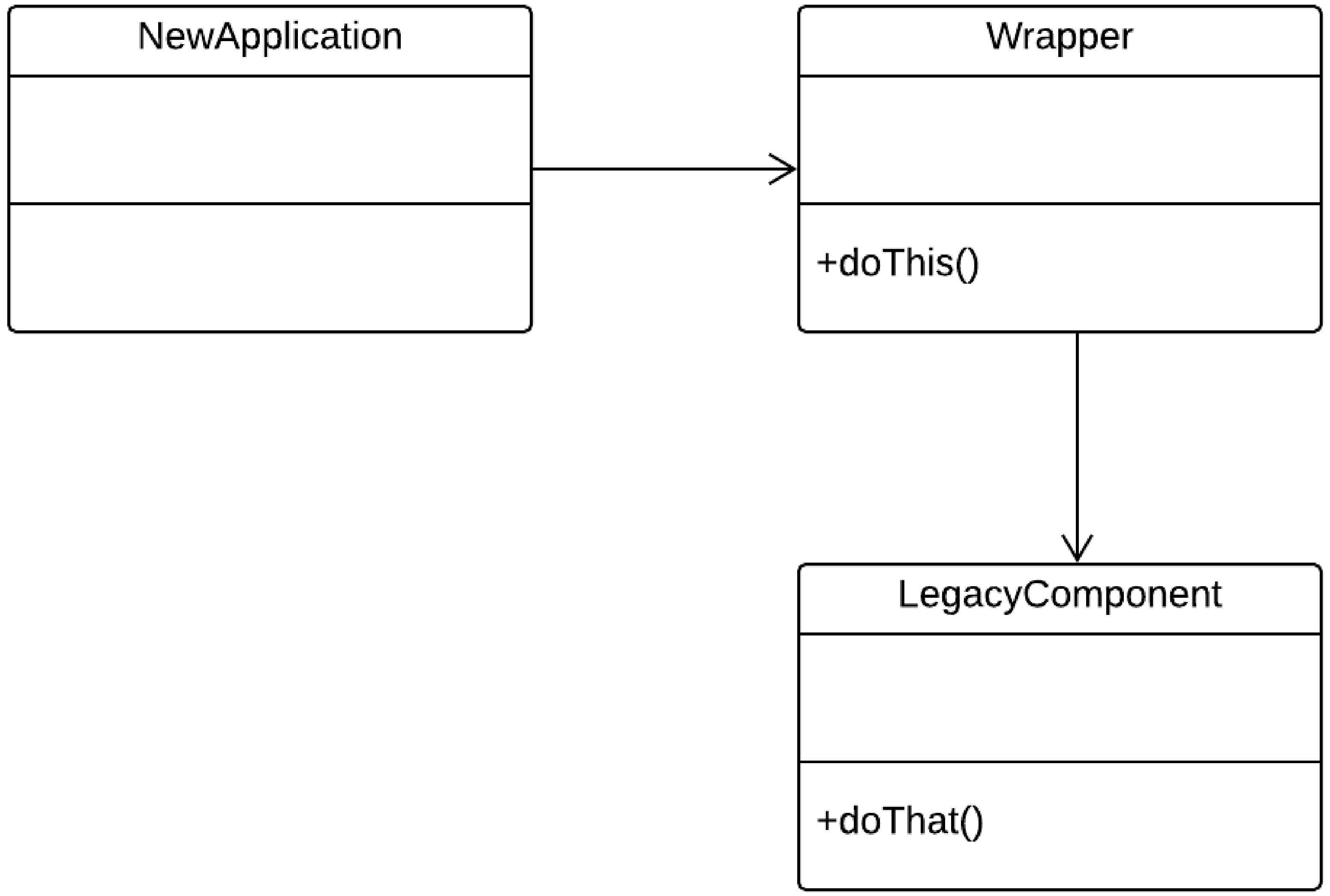
Be able to reuse components with different interface.

### 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**



## Composite

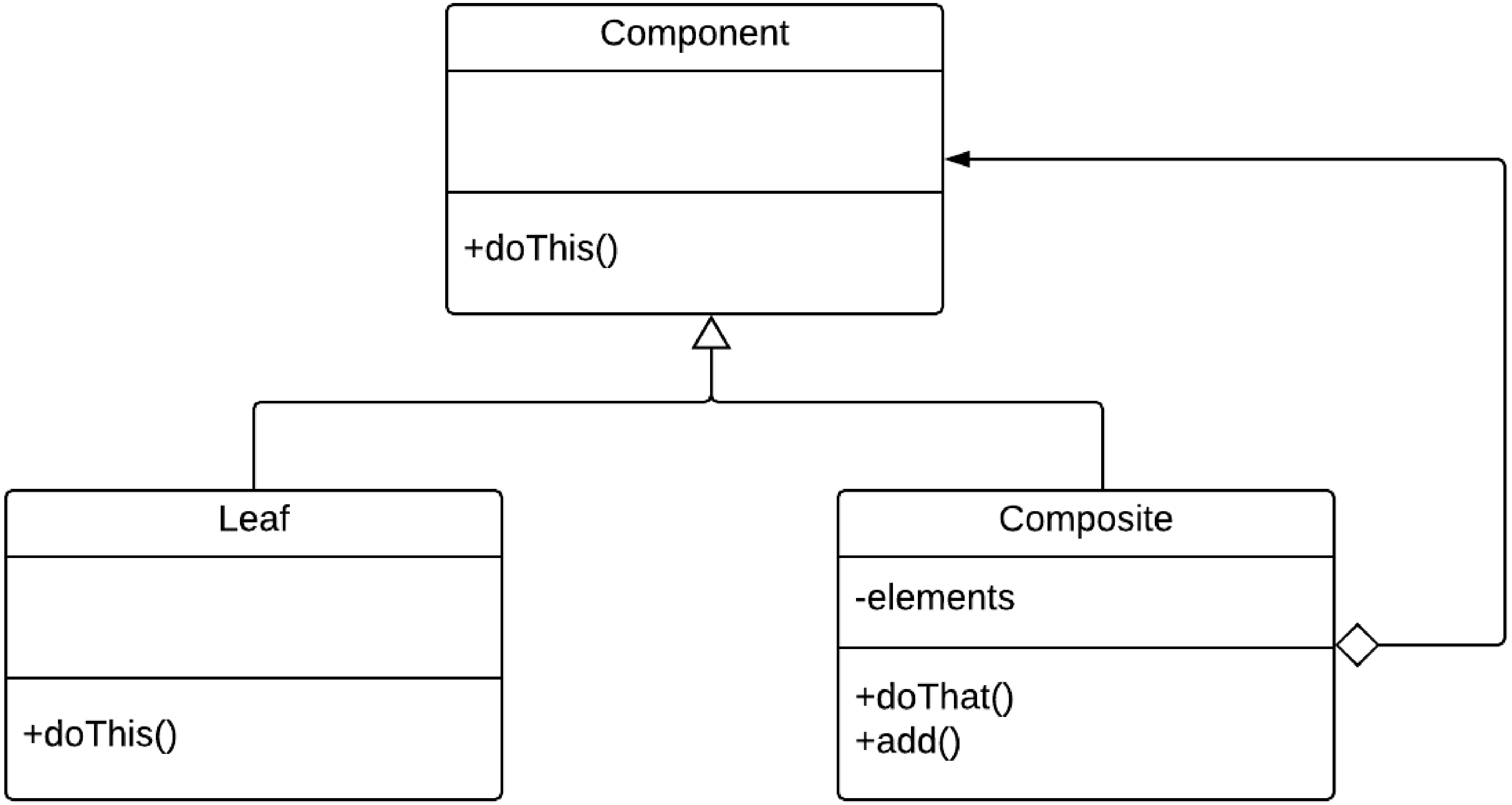
### Problem

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**


Component
+doThis()
Leaf
+doThis()
Composite
-elements
+doThat()
+add()

## Decorator

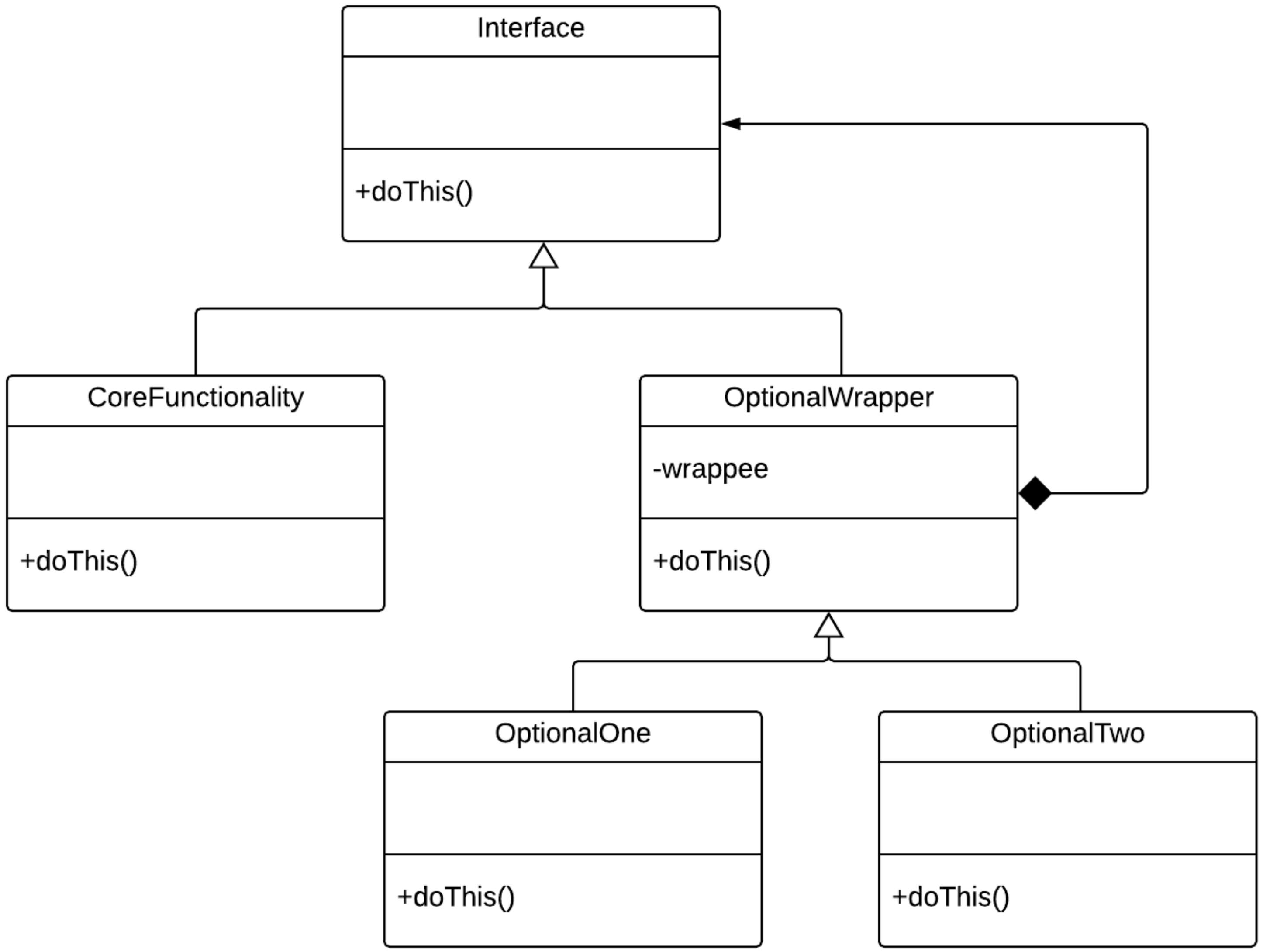
### Problem

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
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
-wrappee
+doThis()
OptionalOne
+doThis()
OptionalTwo
+doThis()

## Abstract Factory

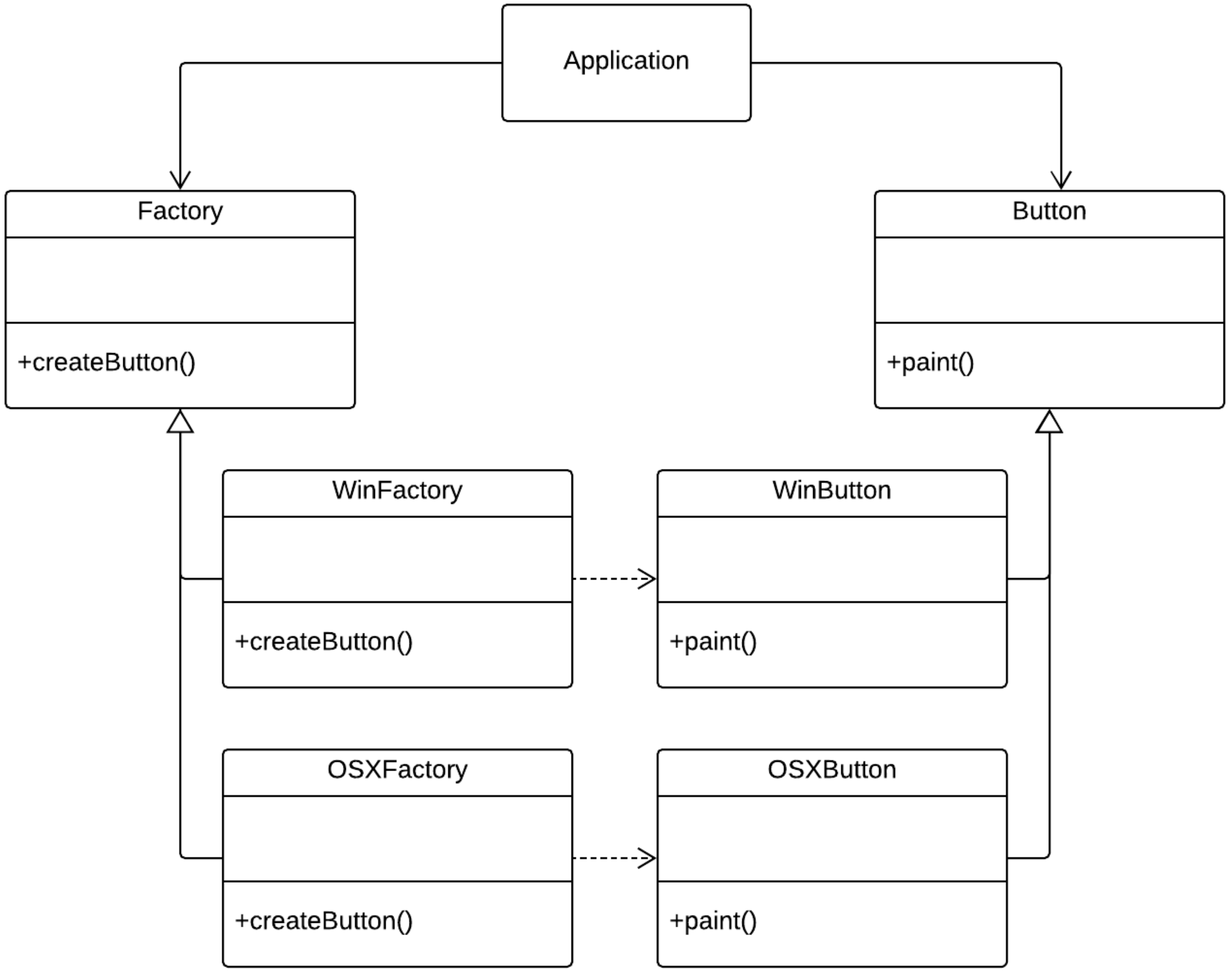
### Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Encapsulation must work without if-else cases.

### Solution

Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

## Diagram


Application
Factory
+createButton()
Button
+paint()
WinFactory
+createButton()
WinButton
+paint()
OSXFactory
+createButton()
OSXButton
+paint()

## Object pool

### Problem

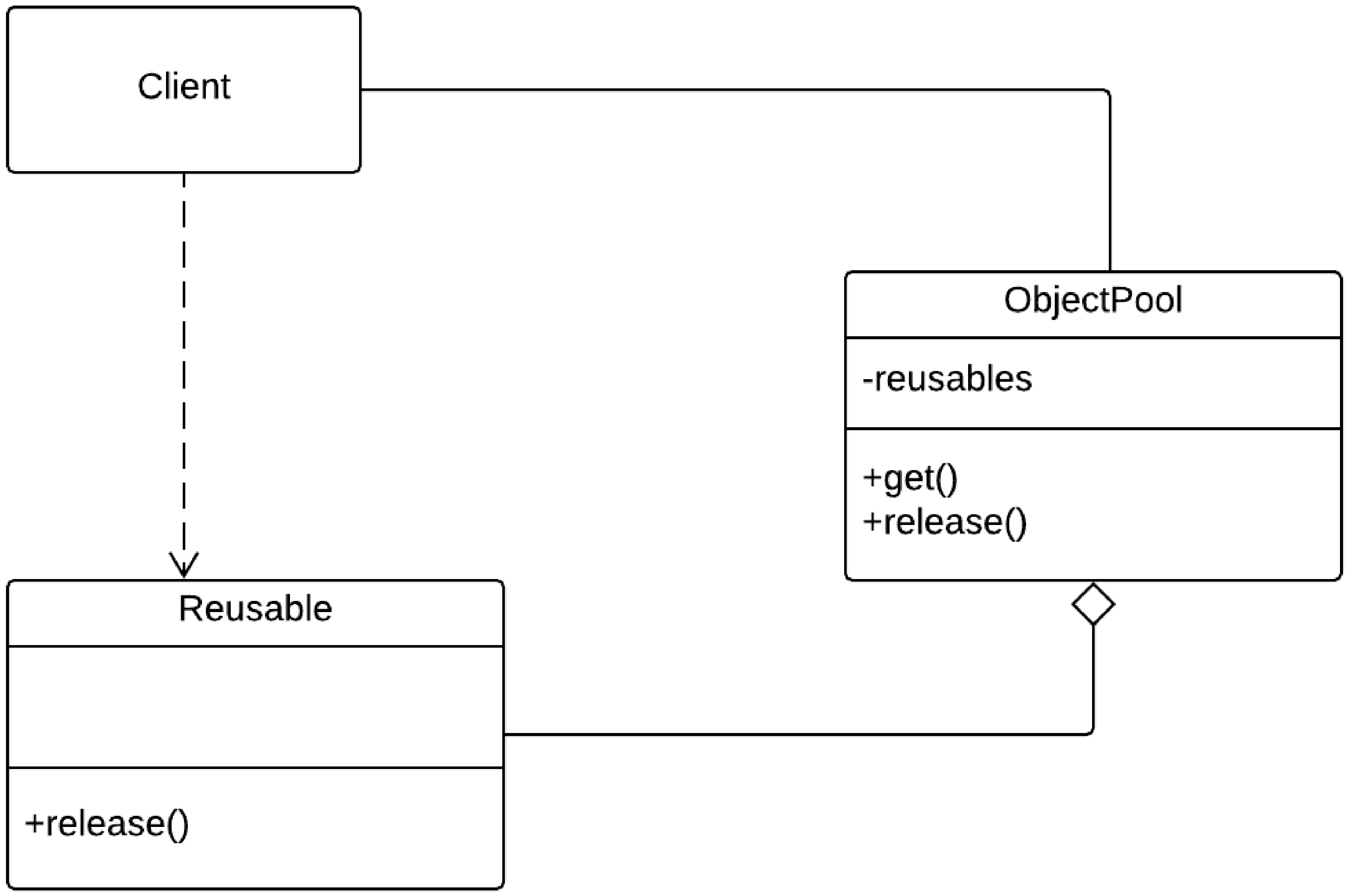
Reuse objects instead of deleting and creating new.

### Solution

Ask the pool for the object that has already been instantiated instead.

Grow pool if no more objects or restrict the number of objects created.

**Diagram**


Client
ObjectPool
-reusables
+get()
+release()
Reusable
+release()

## Singleton

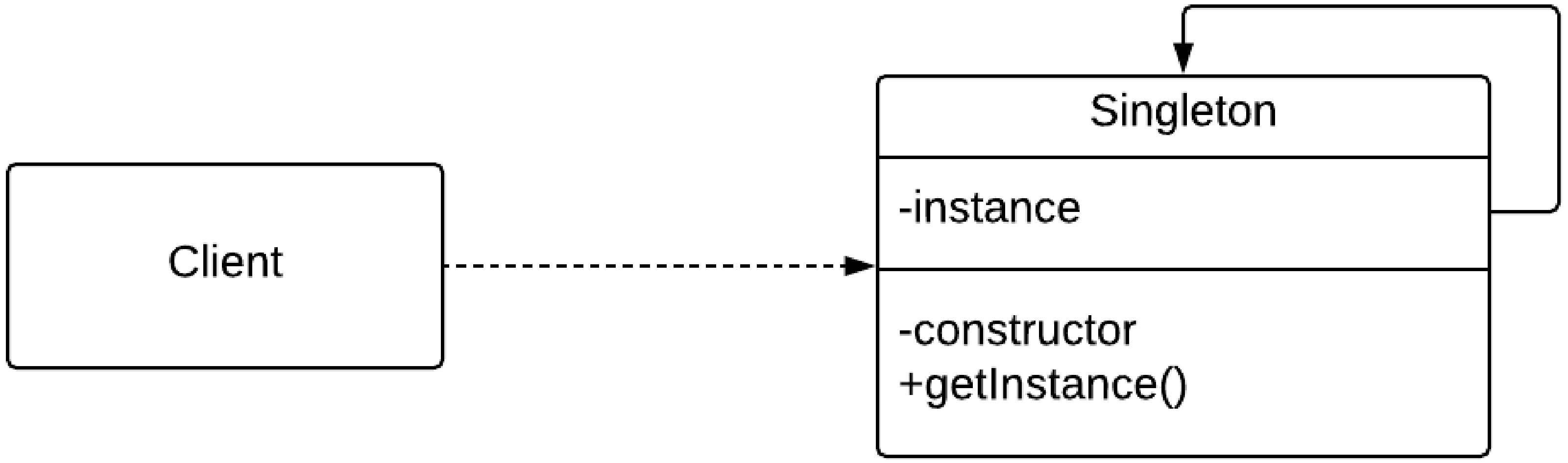
### Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

### Solution

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

**Diagram**


Client
Singleton
-instance
-constructor
+getInstance()