Behavioral patterns II

- Mediator
- Observer
- Template method
- Visitor

Mediator

Problem

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

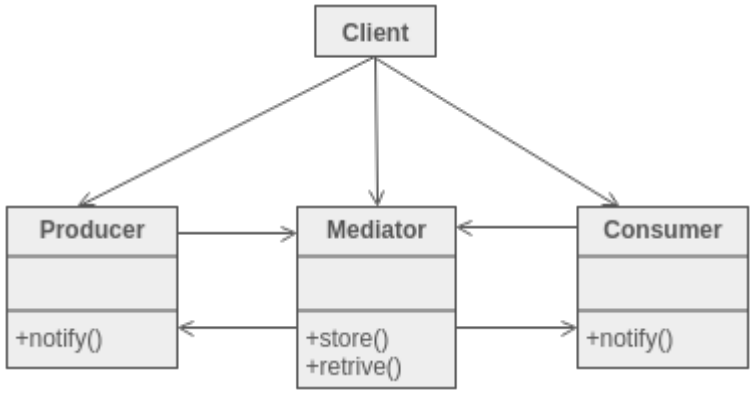
Too many dependencies between classes.

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


Client
Producer
+notify()
Mediator
+store()
+retrive()
Consumer
+notify()

Observer

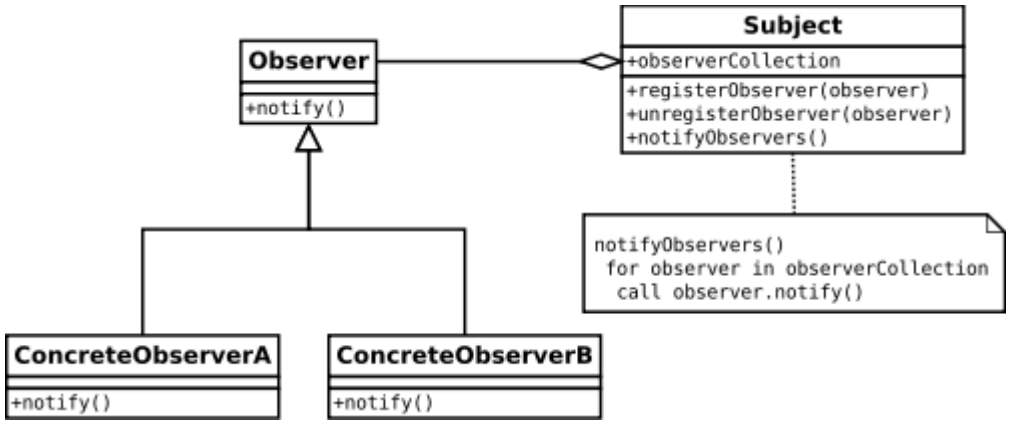
Problem

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
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

Template method

Problem

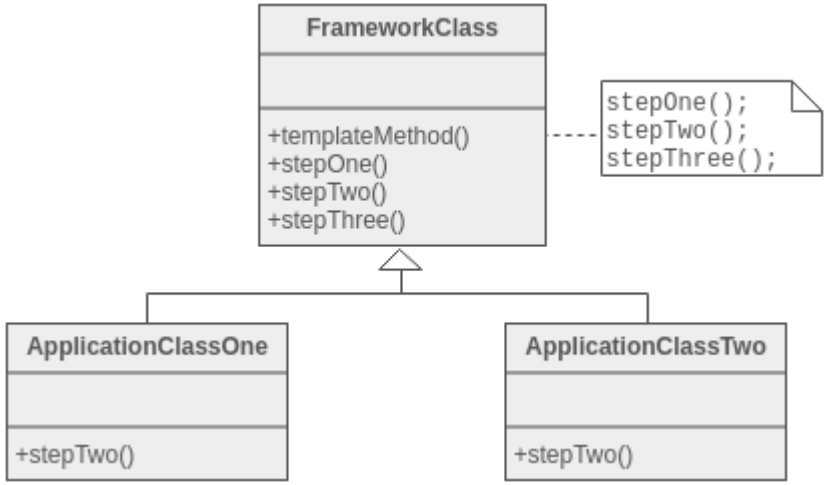
Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Solution

Place similarities to the super class with abstract method which is called in the super class.

Place differences to concrete implementations of the abstract method.

Diagram


FrameworkClass
+templateMethod()
+stepOne()
+stepTwo()
+stepThree()
stepOne();
stepTwo();
stepThree();
ApplicationClassOne
+stepTwo()
ApplicationClassTwo
+stepTwo()

Visitor

Problem

Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.

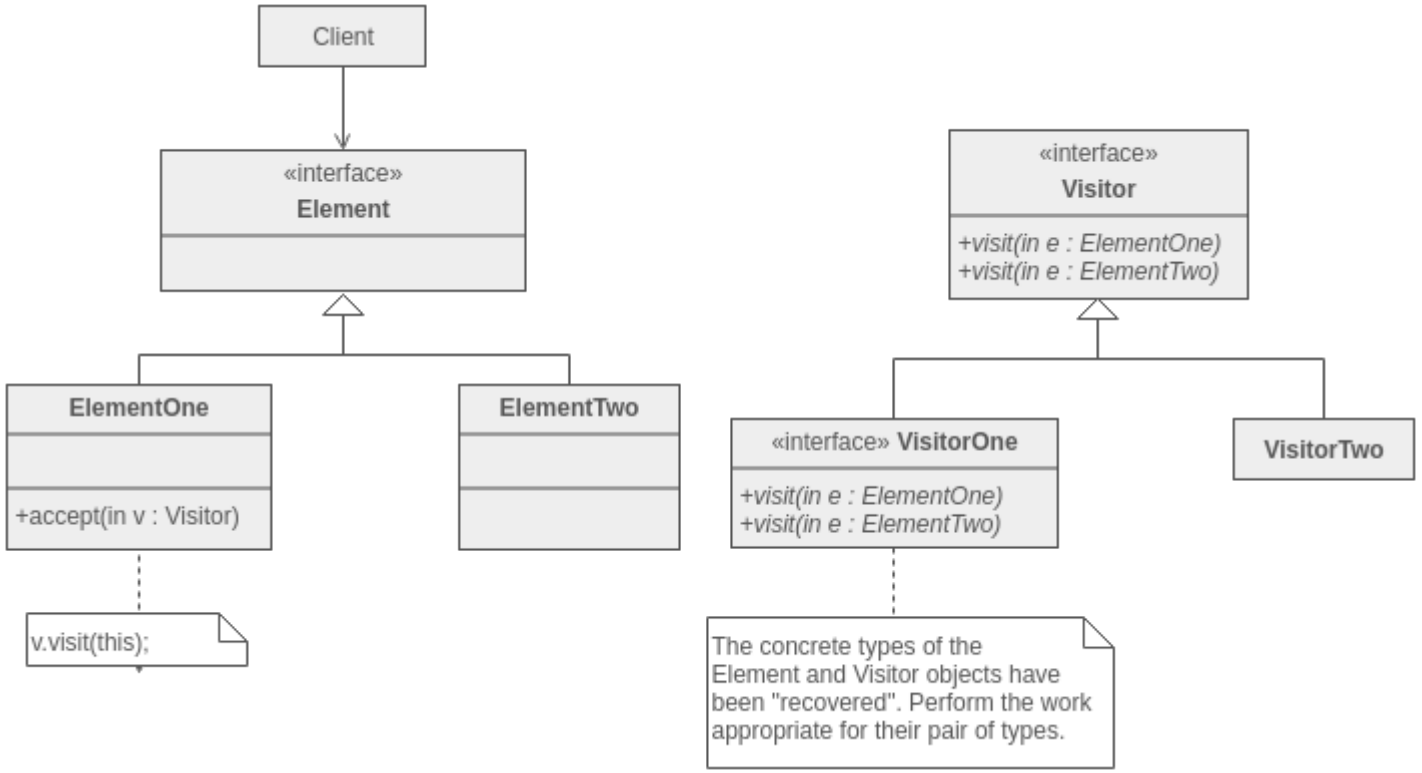
You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Solution

Create a Visitor class hierarchy that defines a visit() method in the interface for each concrete derived class in the aggregate node hierarchy. Each visit() method accepts a single argument - a reference to an original Element derived class.

Each operation to be supported is modelled with a concrete derived class of the Visitor hierarchy. The visit() methods declared in the Visitor base class are now defined in each derived subclass by allocating the "type query and cast" code in the original implementation to the appropriate overloaded visit() method.

Diagram


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
v.visit(this);
ElementTwo
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface» VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.
VisitorTwo