Evan Cummings
CSCI 441 – Advanced Programming
Fall '12 – Joel Henry
Mediator Maintenance and Testing

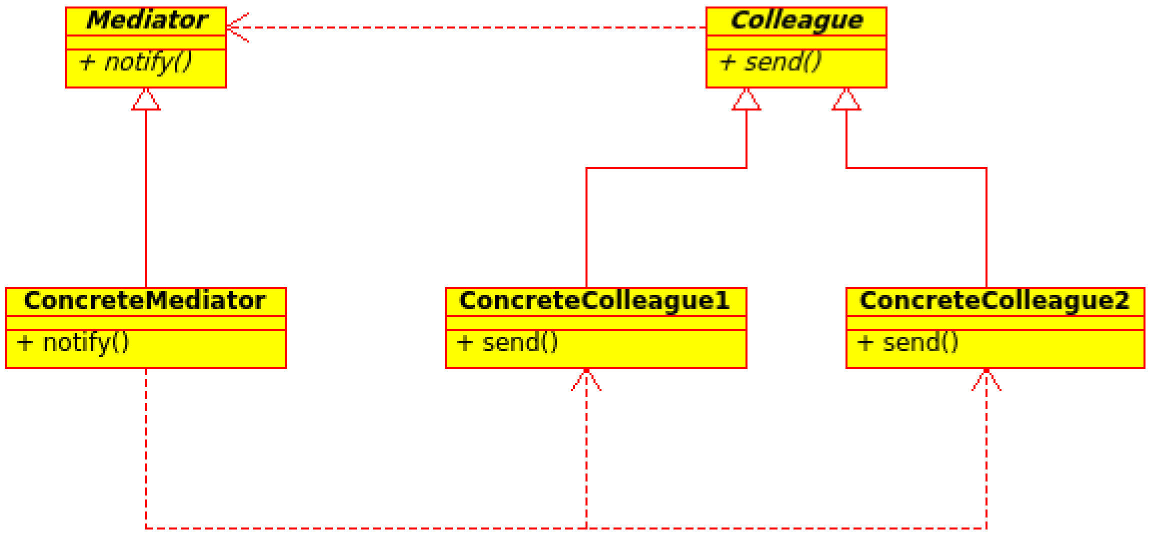
When using the Mediator pattern, and after you have decided the layout of the classes and interactions, it is necessary to test each interaction as they are built. In my example I created a Colleague *User* class which interacts with a Mediator *Chatroom*. When a *User* instance sends a message to the *Chatroom*, the *Chatroom* relays this message to all the other *Users* associated with the *Chatroom*. This is a rather simple process and it was not necessary for me to test this operation before proceeding to the next method – a strength of the pattern. The *notify* method of the *Chatroom* class contains a loop which cycles through and sends the *User's* message to all of the currently connected *Users*.

More complicated scenarios can be imagined than this, and may require a more rigorous testing procedure. For example, if the *User* class was designed to be abstract with child classes inheriting its methods, the *notify* method of the *Chatroom* instance may also need to perform a different task depending on the type of *User*. If this is so, these events must be individually tested. The *Chatroom* may also be similarly designed. These modifications preserve the pattern's readability and would be no more difficult to test than any other method.

As would be expected with any change in requirements, modification to the pattern can produce more complicated code. However, the mediator pattern is an elegant solution to a relatively straight-forward problem. When designed properly, the code is easier to maintain due to the fact that the *Mediator* handles the interactions between the different *Colleagues*. Adding a new rule for interactions between different types of *Colleagues* is easily accomplished, and the code can also be quickly located where you would logically expect it to be.

This pattern also blends well with other patterns. For example, The *Colleagues* may use an Adapter pattern to adapt some functionality of another class, or a Strategy object may be used to provide the *Mediator* with the appropriate *send* method to use for a given situation.

Any way you look at it, Mediator increases readability and maintainability. The downside of this pattern may be that it could be used in the wrong situation. For instance, if there are going to be a fixed number of *Colleagues* created with specific tasks and specific *inter-Colleague* communication rules, this pattern should not be used. The *Mediator* in this case becomes redundant due to the fact that the rules are individually specified.


Mediator
+ notify()
Colleague
+ send()
ConcreteMediator
+ notify()
ConcreteColleague1
+ send()
ConcreteColleague2
+ send()