Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern University

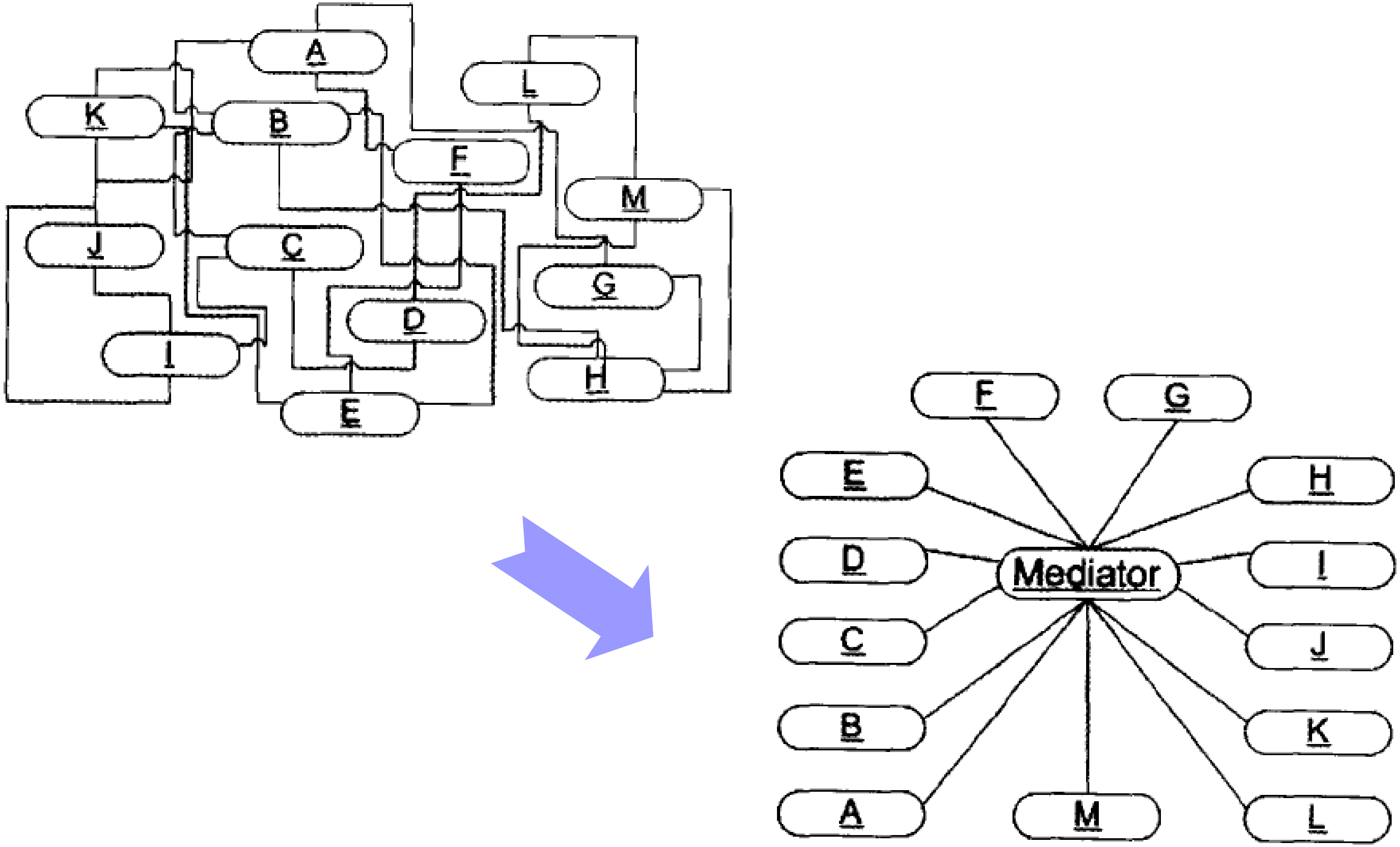
20. Mediator Pattern

Intent

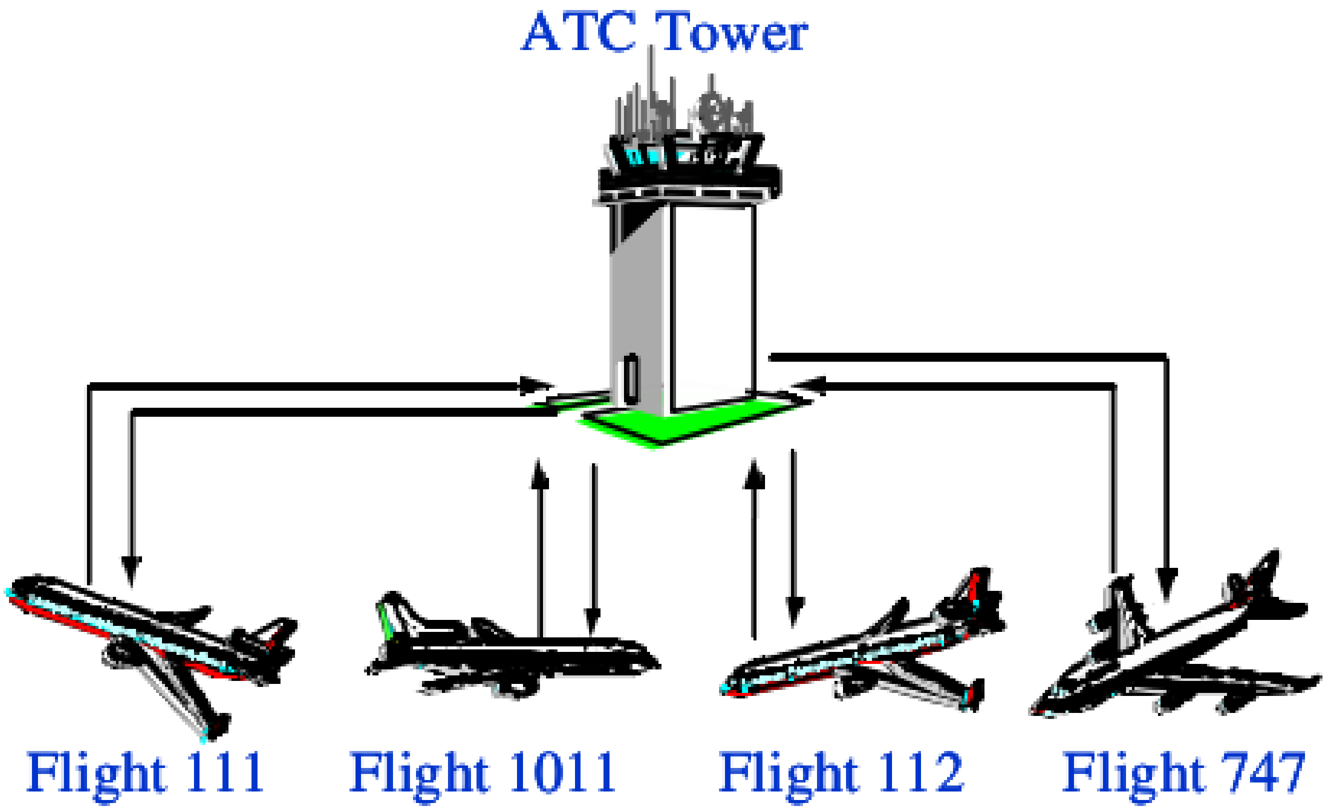
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- 调停者模式包装了一系列对象相互作用的方式，使得这些对象不必互相明显引用。从而使它们可以较松散地耦合。当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响到其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立地变化。

Problems of Object-oriented design

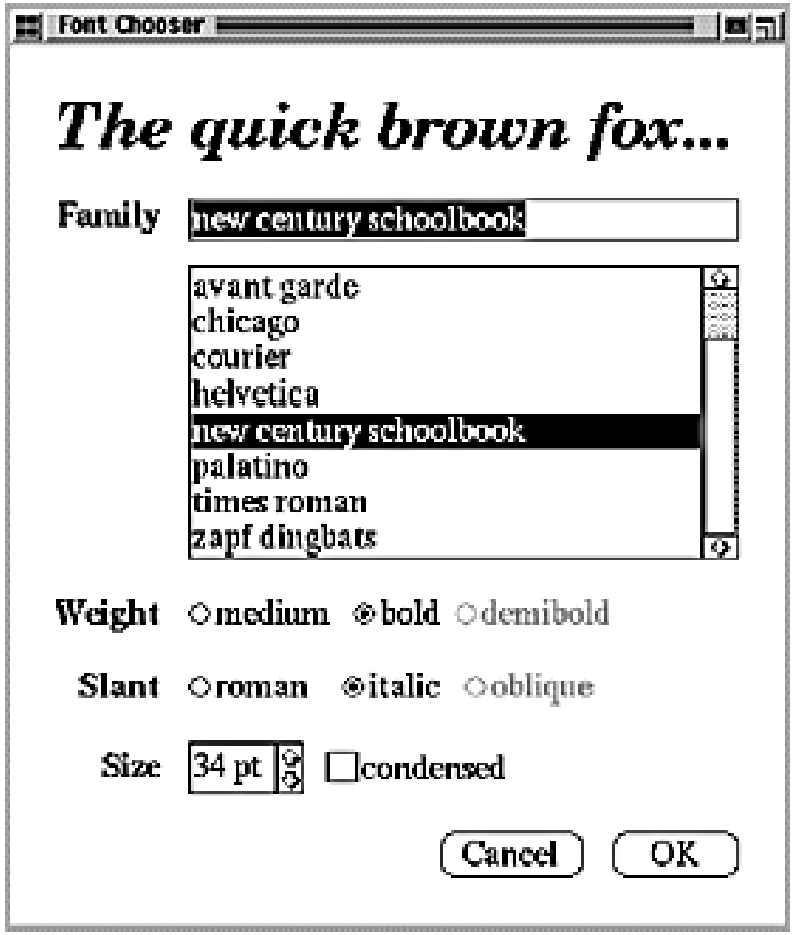
- Object-oriented design encourages the distribution of behavior among objects.
  - □ Such distribution can result in an object structure with many connections between objects;
  - □ In the worst case, every object ends up knowing about every other.
- Though partitioning a system into many objects generally enhances reusability, proliferating interconnections tend to reduce it again.


A
L
K
B
F
M
J
C
G
D
I
H
E
F
G
E
H
D
Mediator
I
C
J
B
K
A
M
L

Example:



Example: Font box



Font Chooser

The quick brown fox...

Family: new century schoolbook

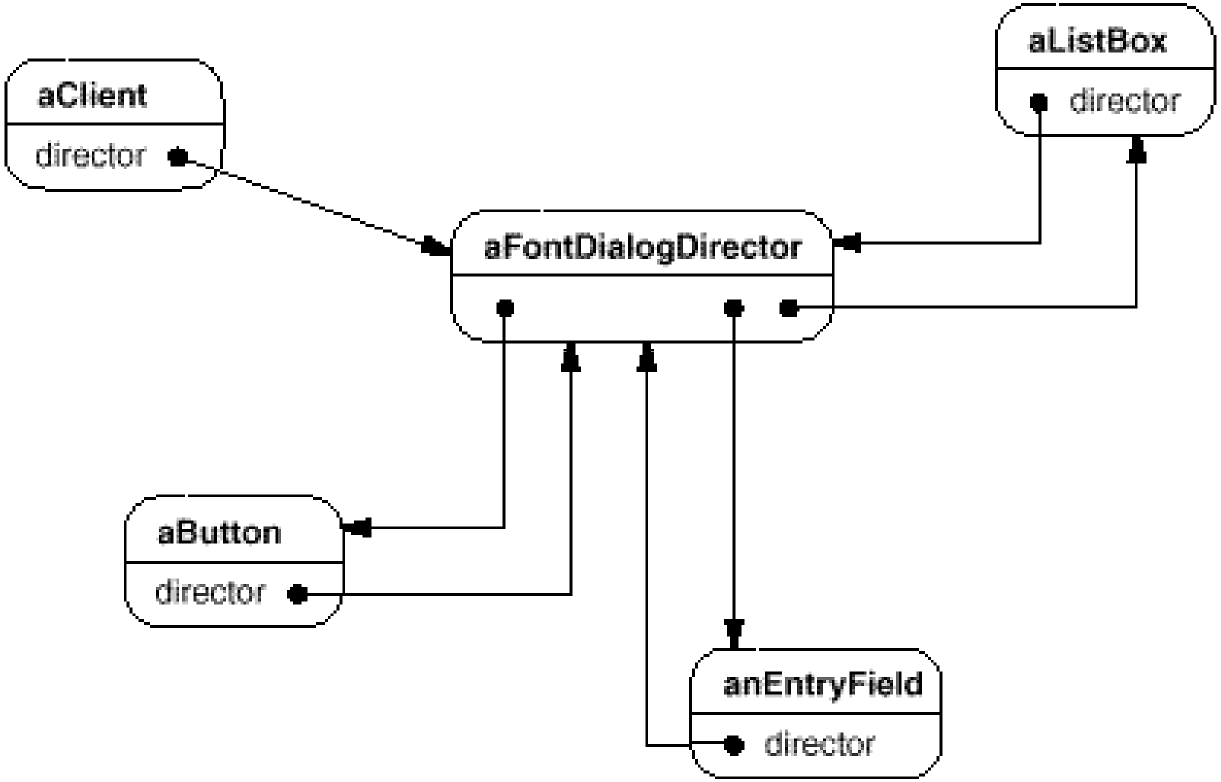
- avant garde
- chicago
- courier
- helvetica
- new century schoolbook
- palatino
- times roman
- zapf dingbats

Weight ○medium ●bold ○demibold

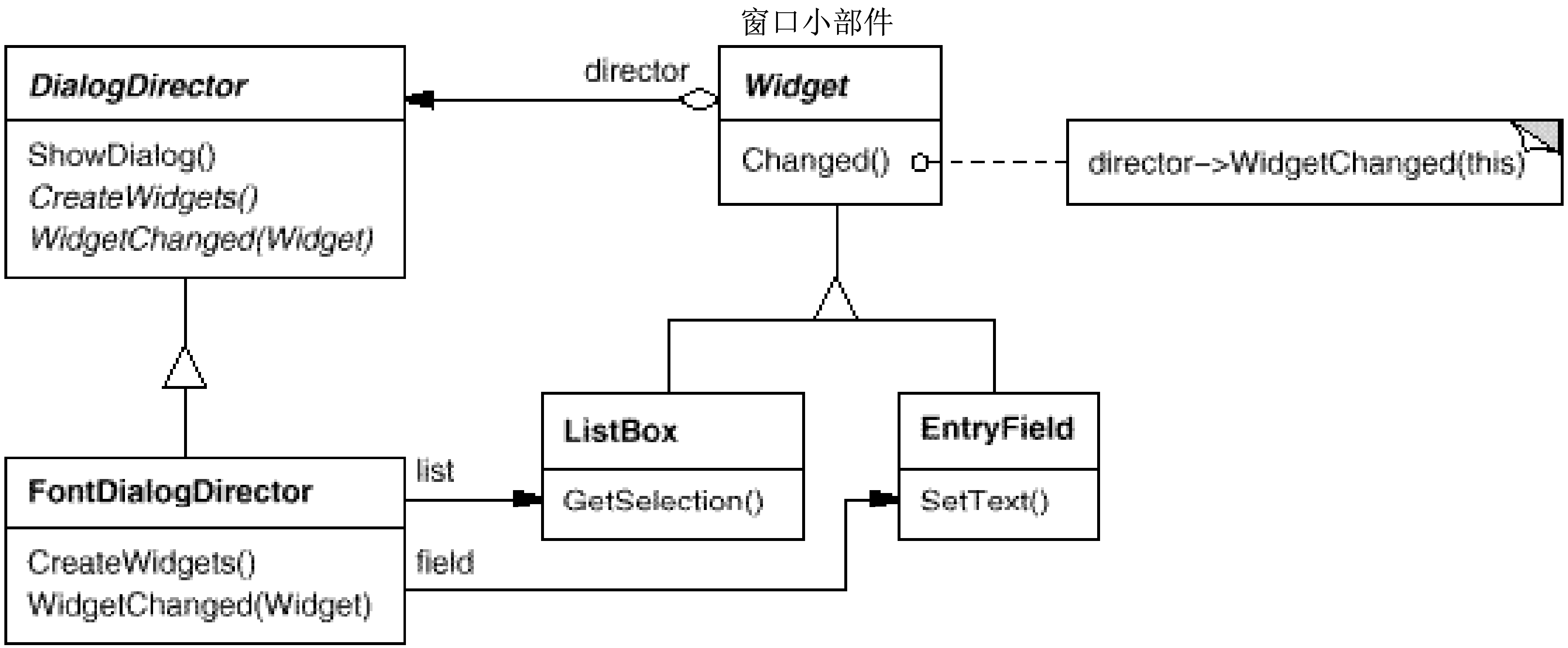
Slant ○roman ●italic ○oblique

Size 34 pt □condensed

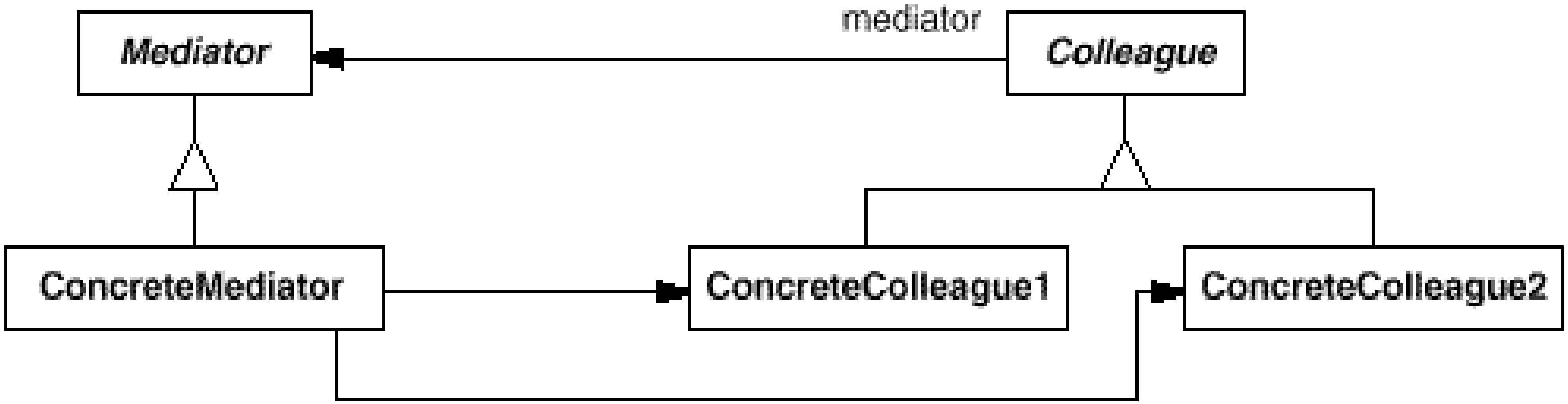
Example: Font box



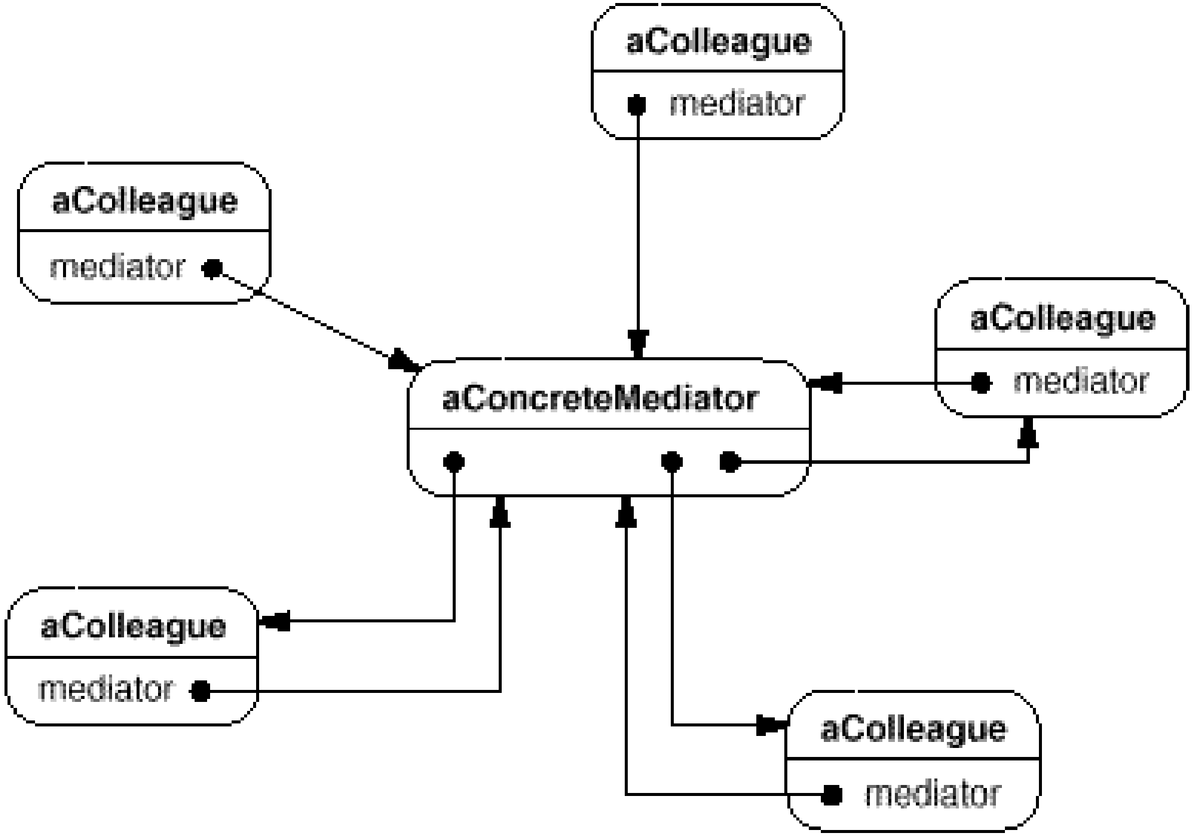
Example: Font box



Structure


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Structure


aColleague
mediator
aColleague
mediator
aColleague
mediator
aConcreteMediator
aColleague
mediator
aColleague
mediator

Participants

- Mediator: Defines an interface for communicating with Colleague objects.
- ConcreteMediator: Implements cooperative behavior by coordinating Colleague objects. knows and maintains its colleagues.
- Colleague classes:
  - Each Colleague class knows its Mediator object.
  - Each Colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Consequences – advantages

- It decouples colleagues.
- It simplifies object protocols.
- It abstracts how objects cooperate.
- It centralizes control.
    - The Mediator pattern trades complexity of interaction for complexity in the mediator. Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

Consequences – drawbacks

- Mediator pattern decrease the complexity between colleagues but increase the complexity of mediator.
  - Sometime, “with a mediator” may worse than “without a mediator”
- Reusing colleagues is possible but reusing the code in Mediator is impractical.
- Mediator providers the extensionality to the colleagues but not itself.
  - Extensionality of Mediator pattern is lean to the colleagues

Applicability

- A set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- Reusing an object is difficult because it refers to and communicates with many other objects.
- A behavior that's distributed between several classes should be customizable without a lot of subclassing.

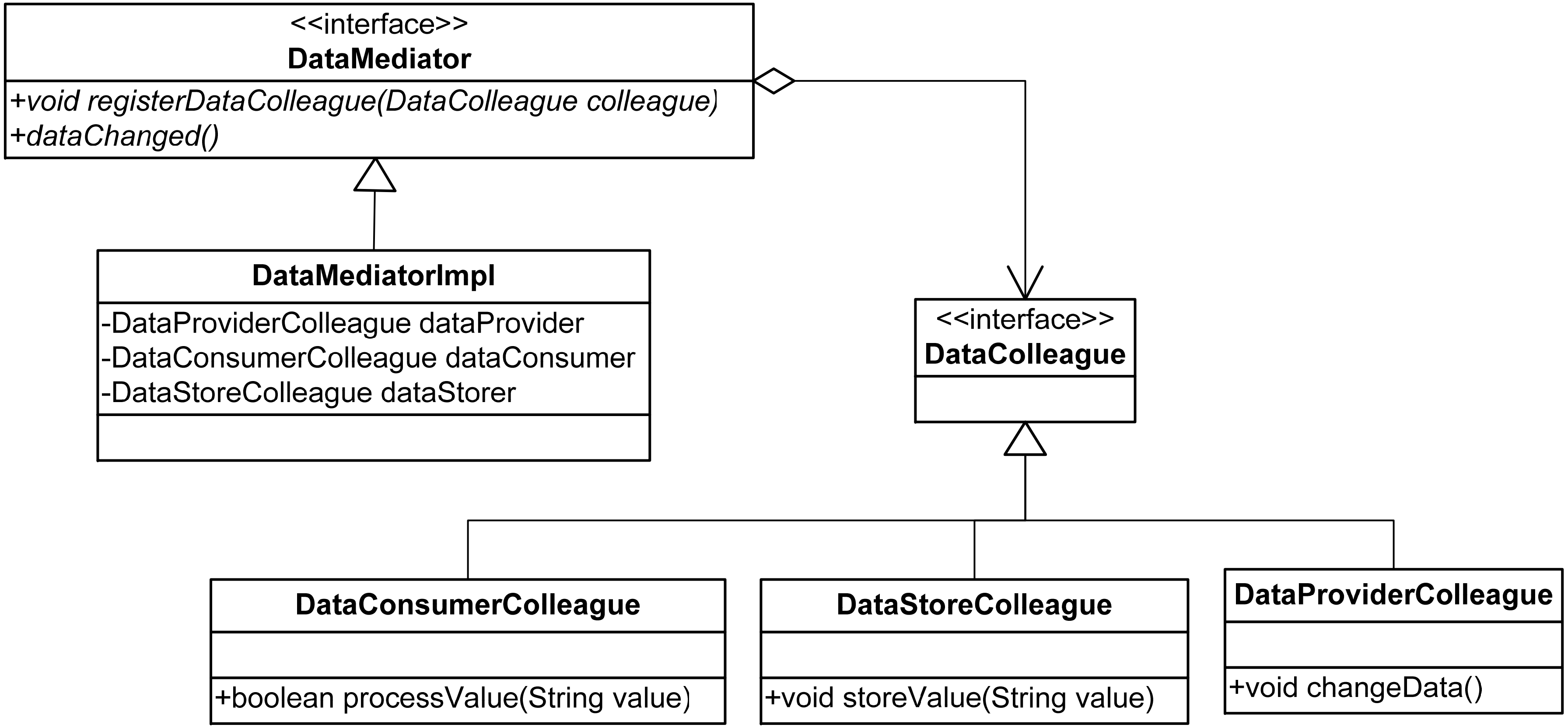
Implementation 1: Omitting the abstract Mediator class.

- The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.
- There's no need to define an abstract Mediator class when colleagues work with only one mediator.

Implementation 2: Colleague-Mediator communication.

- Colleagues have to communicate with their mediator when an event of interest occurs, using the
  - Observer pattern: Colleague classes act as Subjects, sending notifications to the mediator whenever they change state. mediator will notify all college including the sender.
  - Notification interface: Defines a specialized notification interface in Mediator, colleagues communicates each other by this interface. a colleague passes itself as an argument, allowing the mediator to identify the sender.

Example



<<interface>>
DataMediator

+void registerDataColleague(DataColleague colleague)
+dataChanged()

DataMediatorImpl

-DataProviderColleague dataProvider
-DataConsumerColleague dataConsumer
-DataStoreColleague dataStorer

<<interface>>
DataColleague

DataConsumerColleague

+boolean processValue(String value)

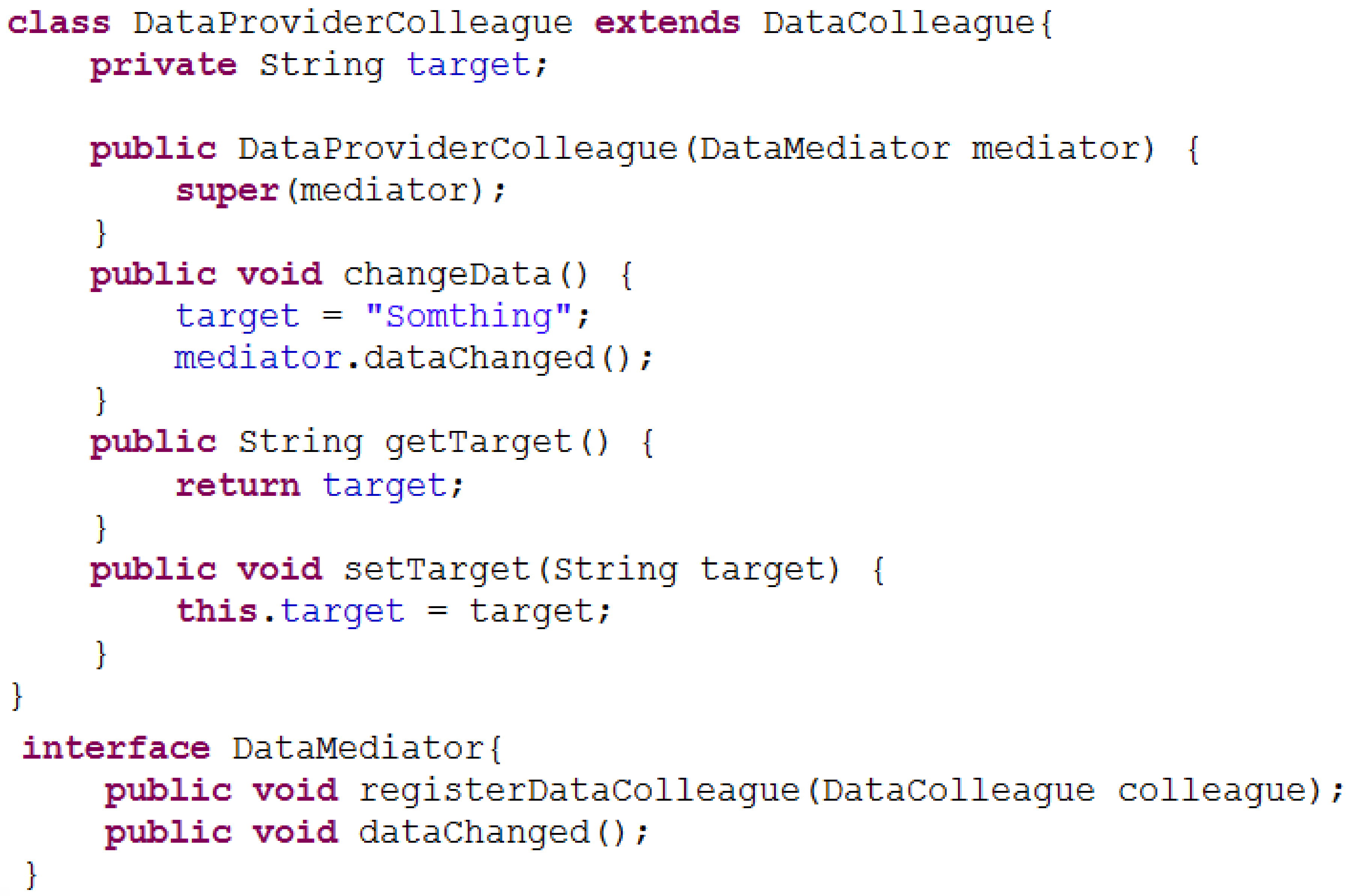
DataStoreColleague

+void storeValue(String value)

DataProviderColleague

+void changeData()

```java
abstract class DataColleague{
    protected DataMediator mediator;
    public DataColleague(DataMediator mediator) {
        this.mediator = mediator;
        mediator.registerDataColleague(this);
    }
}
class DataConsumerColleague extends DataColleague{
    public DataConsumerColleague(DataMediator mediator) {
        super(mediator);
    }
    public boolean processValue(String value) {
        // TODO process the target value
        // if (condition){ return false; }
        return true;
    }
}
class DataStoreColleague extends DataColleague{
    public DataStoreColleague(DataMediator mediator) {
        super(mediator);
    }
    public void storeValue(String value) {
        // TODO store the target value
    }
}
```



```
class DataProviderColleague extends DataColleague{
    private String target;

    public DataProviderColleague(DataMediator mediator) {
        super(mediator);
    }
    public void changeData() {
        target = "Somthing";
        mediator.dataChanged();
    }
    public String getTarget() {
        return target;
    }
    public void setTarget(String target) {
        this.target = target;
    }
}
 interface DataMediator{
     public void registerDataColleague(DataColleague colleague);
     public void dataChanged();
 }
```

```java
class DataMediatorImpl implements DataMediator{
    private DataProviderColleague dataProvider;
    private DataConsumerColleague dataConsumer;
    private DataStoreColleague dataStorer;

    public void registerDataColleague(DataColleague colleague) {
        Class<?> clazz = colleague.getClass();
        if (clazz.equals(DataProviderColleague.class)) {
            dataProvider = (DataProviderColleague)colleague;
        } else if (clazz.equals(DataConsumerColleague.class)) {
            dataConsumer = (DataConsumerColleague)colleague;
        }else if (clazz.equals(DataStoreColleague.class)) {
            dataStorer = (DataStoreColleague)colleague;
        }else {
            throw new RuntimeException("Unknown DataColleague " + colleague);
        }
    }
    public void dataChanged() {
        String value = dataProvider.getTarget();
        if (dataConsumer != null) {
            if (dataConsumer.processValue(value)) {
                if (dataStorer != null) {
                    dataStorer.storeValue(value);
                }
            }
        }
    }
}
```

```java
public class Client {
    public void test() {
        DataMediator mediator = new DataMediatorImpl();
        DataProviderColleague dataProvider= new DataProviderColleague(mediator);
        //DataConsumerColleague dataConsumer= new DataConsumerColleague(mediator);
        //DataStoreColleague dataStorer= new DataStoreColleague(mediator);
        new DataConsumerColleague(mediator);
        new DataStoreColleague(mediator);
        dataProvider.changeData();
    }
}
```

Extension 1: Law of Demeter (LoD)
A Principle of Object Oriented Design

- Only talk to your immediate friends.
- One never calls a method on an object you got from another call nor on a global object.
  - □ You can play with yourself.
  - □ You can play with your own toys (but you can't take them apart),
  - □ You can play with toys that were given to you.
  - □ And you can play with toys you've made yourself.

Extension 1: Law of Demeter (LoD)

- Explanation in plain English:
  - Your method can call other methods in its class directly;
  - Your method can call methods on its own fields directly (but not on the fields' fields);
  - When your method takes parameters, your method can call methods on those parameters directly.
  - When your method creates local objects, that method can call methods on the local objects.

  But

  - One should not call methods on a global object
  - One should not have a chain of messages a.getB().getC().doSomething() in some class other than a's class.

Extension 2: Misusing Mediator

- Mediator pattern is applied to a system for avoiding mess and ugly.
- Mediator pattern should not be applied to a system which has been mess and ugly.
  - Such system should be re-designed;
  - Responsibilites of classes should be repartitioned;
  - When the system is going to be mess, firstly, try to clarify the functional dependency;
  - Mediator pattern should be used to avoid the mess system, but not fix it.
  - Mediator pattern is a pattern but not a sliver bullet.

Extension 2: Mediator is not for fixing mess

- 一个初级设计师在对面向对象的技术不熟悉时，会使一个系统在责任的分割上发生混乱。
- 责任分割的混乱会使得系统中的对象与对象之间产生不适当的复杂关系。
- 这时候，一个很糟的想法就是继续这个错误，并使用调停者模式“化解”这一团乱麻。实际上，这样一来，责任错误划分的混乱不但不会得到改正，而且还会制造出一个莫名其妙的怪物：一个处于一团乱麻之中的混乱之首。

Let’s go to next…