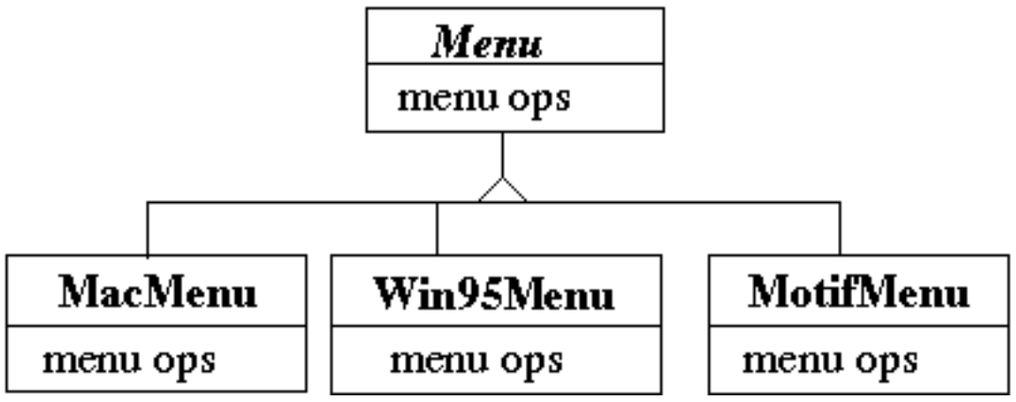
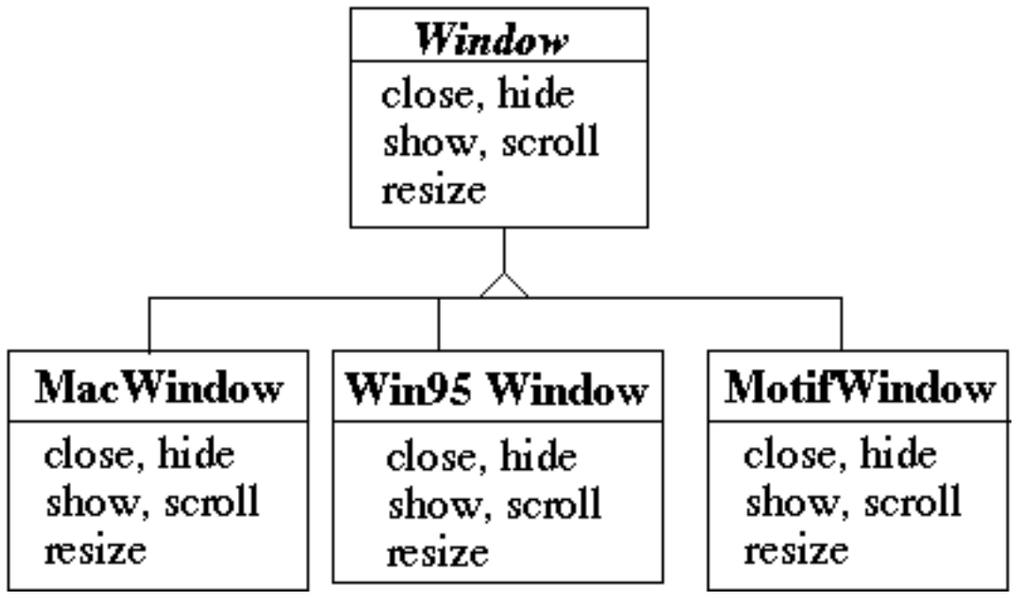
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 19 Abstract Factory, Flyweight, Mediator, Facade
Nov 17, 2022

Abstract Factory

Abstract Factory

Write a cross platform window toolkit





Bad Code Dependencies

```
public void installDisneyMenu()
    {
    Menu disney = new MacMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```

Use Abstract Factory

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
    }
```

```
class MacWidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a mac window }

    public Menu createMenu()
        { code to create a mac Menu }

    public Button createButton()
        { code to create a mac button }
}

class Win95WidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a Win95 window }

    public Menu createMenu()
        { code to create a Win95 Menu }

    public Button createButton()
        { code to create a Win95 button }
}
```

Use one Factory per Application

```
public void installDisneyMenu(WidgetFactory myFactory)
   {
   Menu disney = myFactory.createMenu();
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

Abstract Factory

Encapsulate a group of individual factories that have a common theme

Separates the details of implementation of a set of objects
from its general usage

How Do Abstract Factories create Things?

Use Subclass Factory Method

```
abstract class WidgetFactory
    {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
    }

class MacWidgetFactory extends WidgetFactory
    {
    public Window createWindow()
      { return new MacWidow() }

    public Menu createMenu()
      { return new MacMenu() }

    public Button createButton()
      { return new MacButton()  }
    }
```

Use Widget Factory Method

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
       { return windowFactory.createWindow() }

    public Menu createMenu();
       { return menuFactory.createMenu() }

    public Button createButton()
       { return buttonFactory.createMenu() }
}
```

```
class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
       windowFactory = new MacWindow();
       menuFactory = new MacMenu();
       buttonFactory = new MacButton();
    }
}
```

```
class MacWindow extends Window {
    public Window createWindow() { blah }
    etc.
```

Why Widget Factory Method?

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow()
       { return windowFactory.createWindow() }

    public Window createWindow( Rectangle size)
       { return windowFactory.createWindow( size ) }

public Window createWindow( Rectangle size, String title)
    { return windowFactory.createWindow( size, title) }

public Window createFancyWindow()
    { return windowFactory.createFancyWindow() }

public Window createPlainWindow()
    { return windowFactory.createPlainWindow() }
```

Multiple ways to create Widget

Use Prototype

```
class WidgetFactory{
    private Window windowPrototype;
    private Menu menuPrototype;
    private Button buttonPrototype;

    public WidgetFactory( Window windowPrototype,
                          Menu menuPrototype,
                          Button buttonPrototype)
       {
       this.windowPrototype = windowPrototype;
       this.menuPrototype = menuPrototype;
       this.buttonPrototype = buttonPrototype;
       }

  public Window createWindow()
     { return windowPrototype.createWindow() }

  public Window createWindow( Rectangle size)
     { return windowPrototype.createWindow( size ) }

  public Window                ()
     { return menuPrototype.createMenu() }
  etc.
```

How to prevent Cheating?

```
public void installDisneyMenu(WidgetFactory myFactory)
    {
    // We ship next week, I can't get the stupid generic Menu
    // to do the fancy Mac menu stuff
    // Windows version won't ship for 6 months
    // Will fix this later

    MacMenu disney = (MacMenu) myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addMacGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```

Flyweight

Flyweight

Use sharing to support large number of fine-grained objects efficiently

Text Example

A document has many instances of the character 'a'

Character has

- Font
- width
- Height
- Ascenders
- Descenders
- Where it is in the document

Most of these are the same for all instances of 'a'

Use one object to represent all instances of 'a'

Java String Example

```
public void testInterned() {
    String a1 = "catrat";
    String a2 = "cat";
    assertFalse(a1 == (a2+ "rat"));

    String a3 = (a2 + "rat").intern();
    assertTrue(a1 == a3);
    String a4 = "cat" + "rat";
    assertTrue(a1 == a4);
    assertTrue(a3 == a4);
}
```

public String intern()

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

Intrinsic State

Information that is independent from the object's context

The information that can be shared among many objects

So can be stored inside of the flyweight

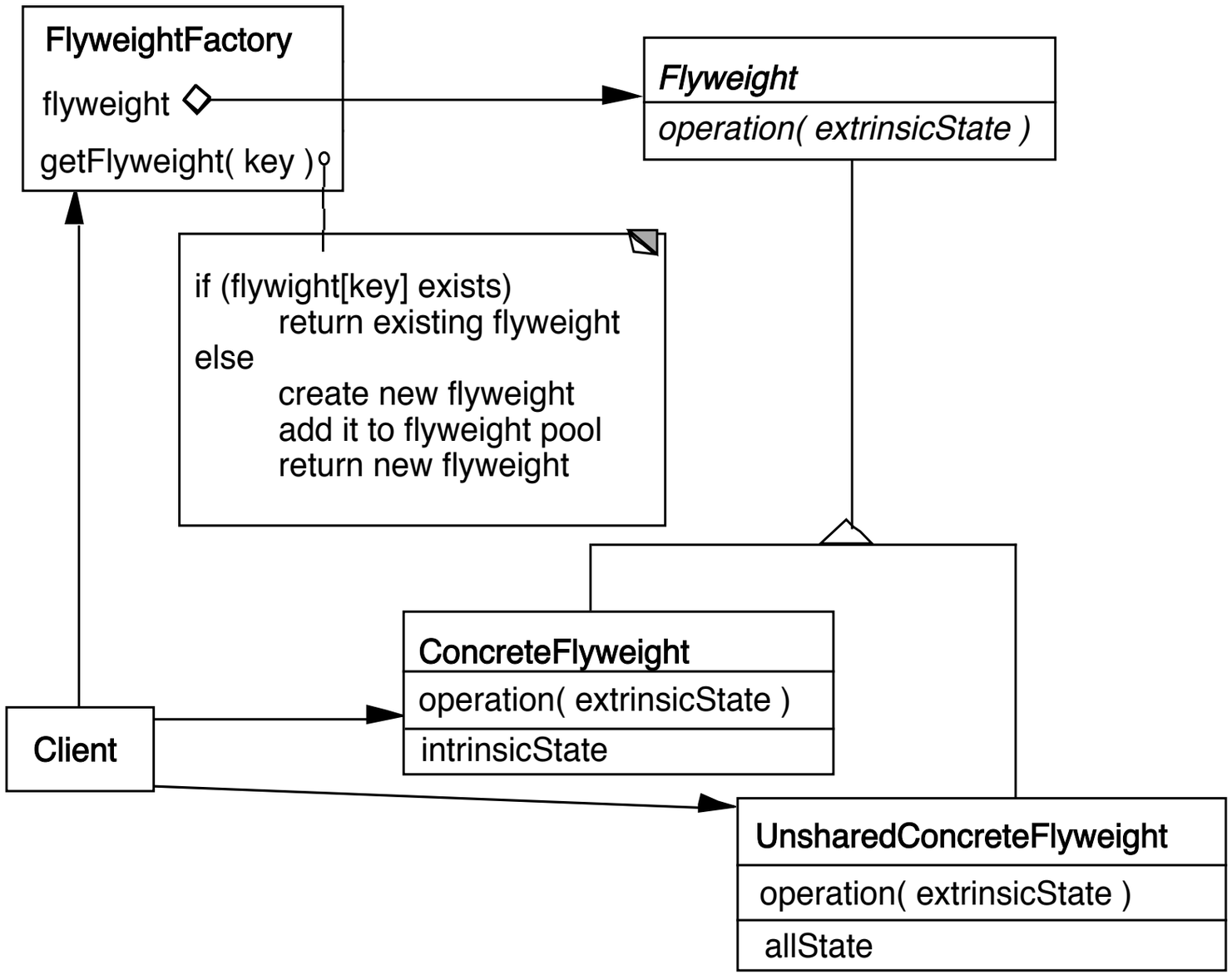
Extrinsic State

Information that is dependent on the object's context

The information that can not be shared among objects

So has to be stored outside of the flyweight

Structure


FlyweightFactory
flyweight
getFlyweight(key)
Flyweight
operation(extrinsicState)
if (flywight[key] exists)
return existing flyweight
else
create new flyweight
add it to flyweight pool
return new flyweight
ConcreteFlyweight
operation(extrinsicState)
intrinsicState
Client
UnsharedConcreteFlyweight
operation(extrinsicState)
allState

The Hard Part

Separating state from the flyweight

How easy is it to identify and remove extrinsic state

Will it save space to remove extrinsic state

Example Text

Run Arrays

aaaaabaaaaaaaaaaaaaaaaaa

a b a

5 1 20

Text Example

Lexi Document Editor

Uses character objects with font information
(To support graphic elements)

"A Cat in the hat came ***back*** the very next day"

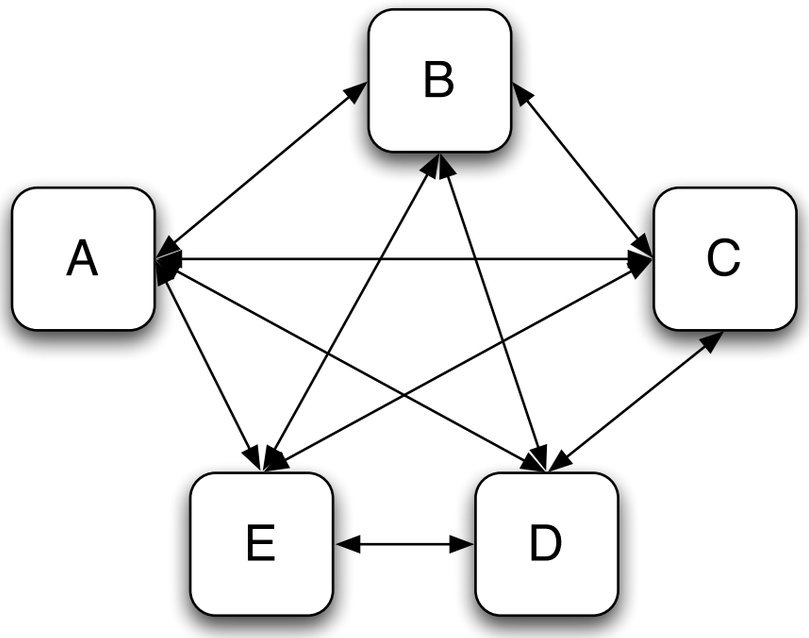
Use run array to store font information (extrinsic state)

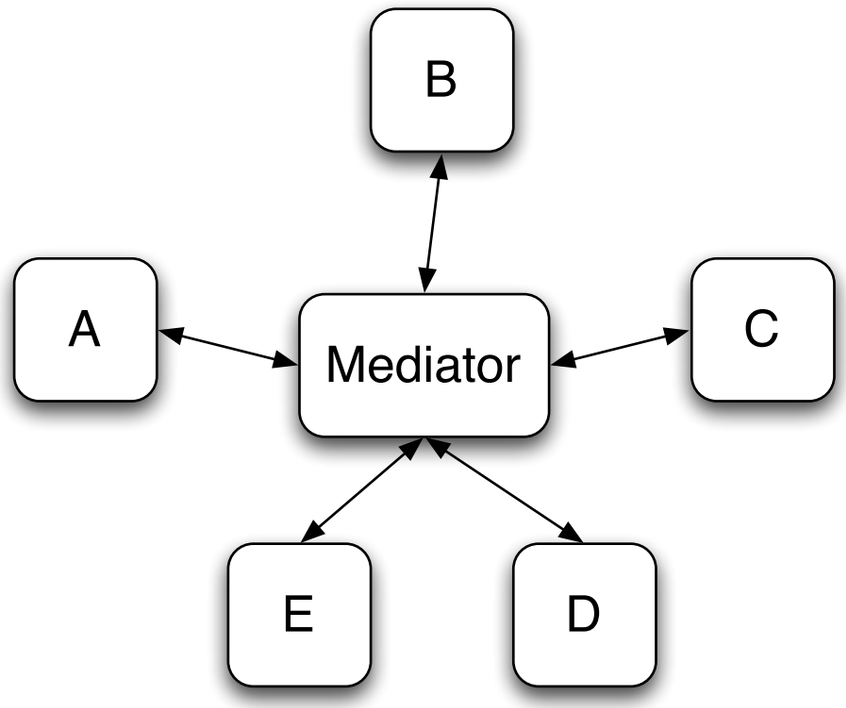
Normal	Bold	Normal
22	4	18

Mediator

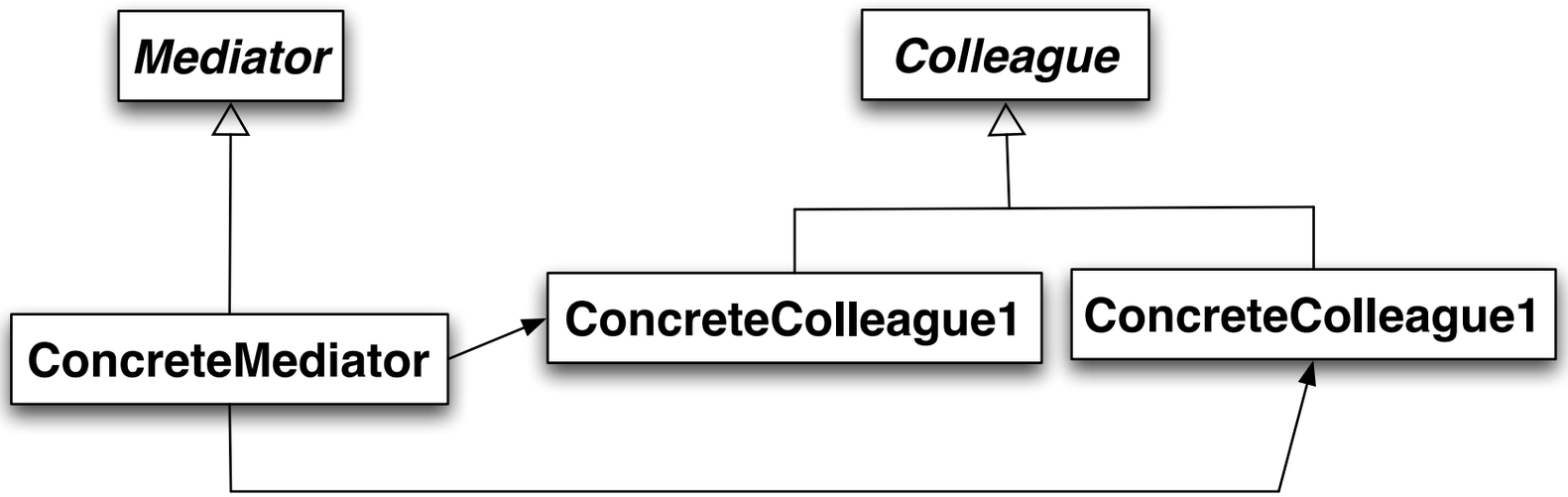
Mediator

A mediator controls and coordinates the interactions of a group of objects





Structure


Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague1

Participants

Mediator

Defines an interface for communicating with Colleague objects

ConcreteMediator

Implements cooperative behavior by coordinating Colleague objects

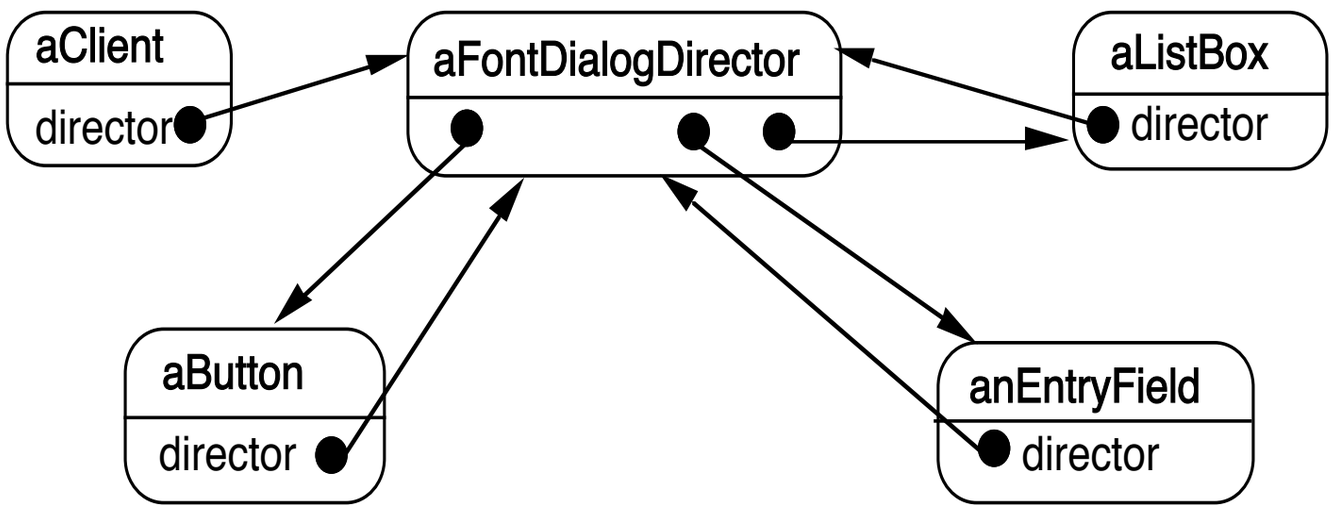
Knows and maintains its colleagues

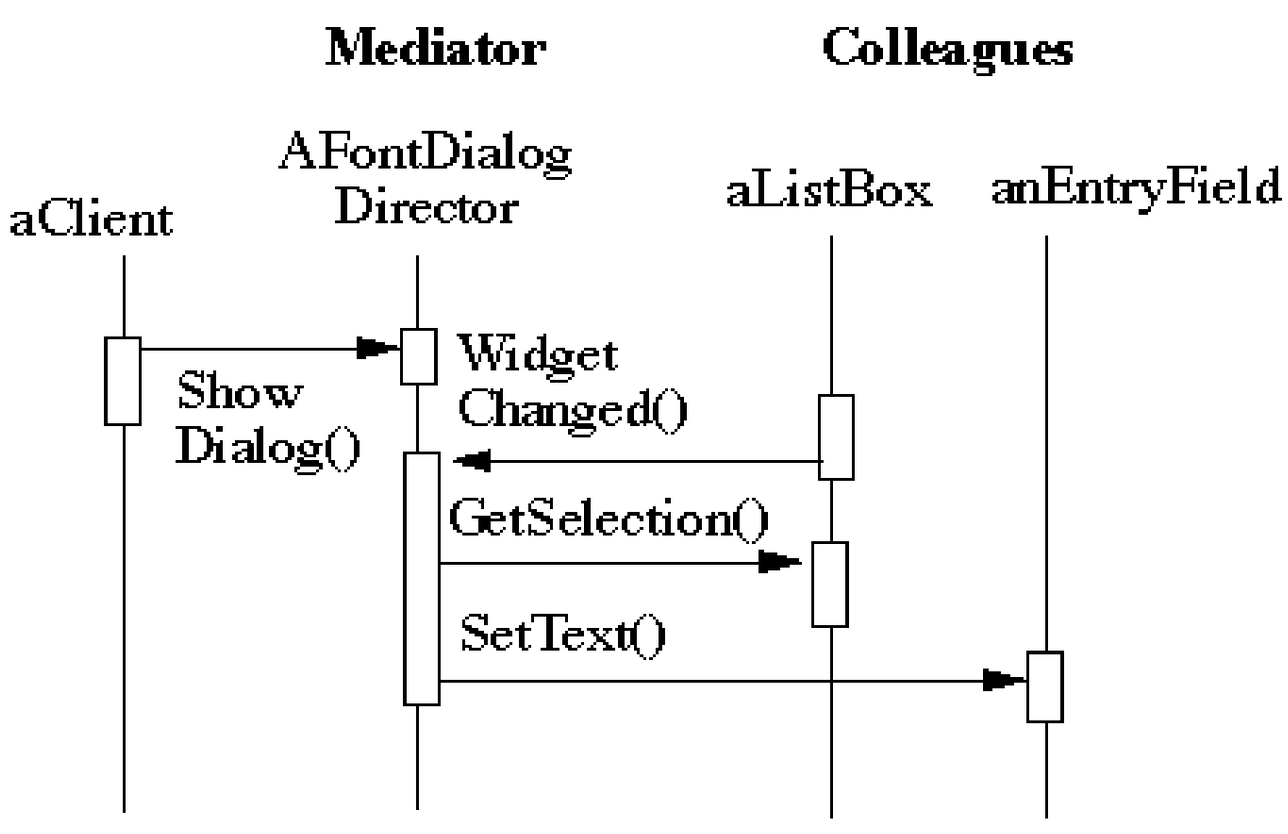
Colleague classes

Each Colleague class knows its Mediator object

Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Motivating Example - Dialog Boxes





How does this differ from a God Class?

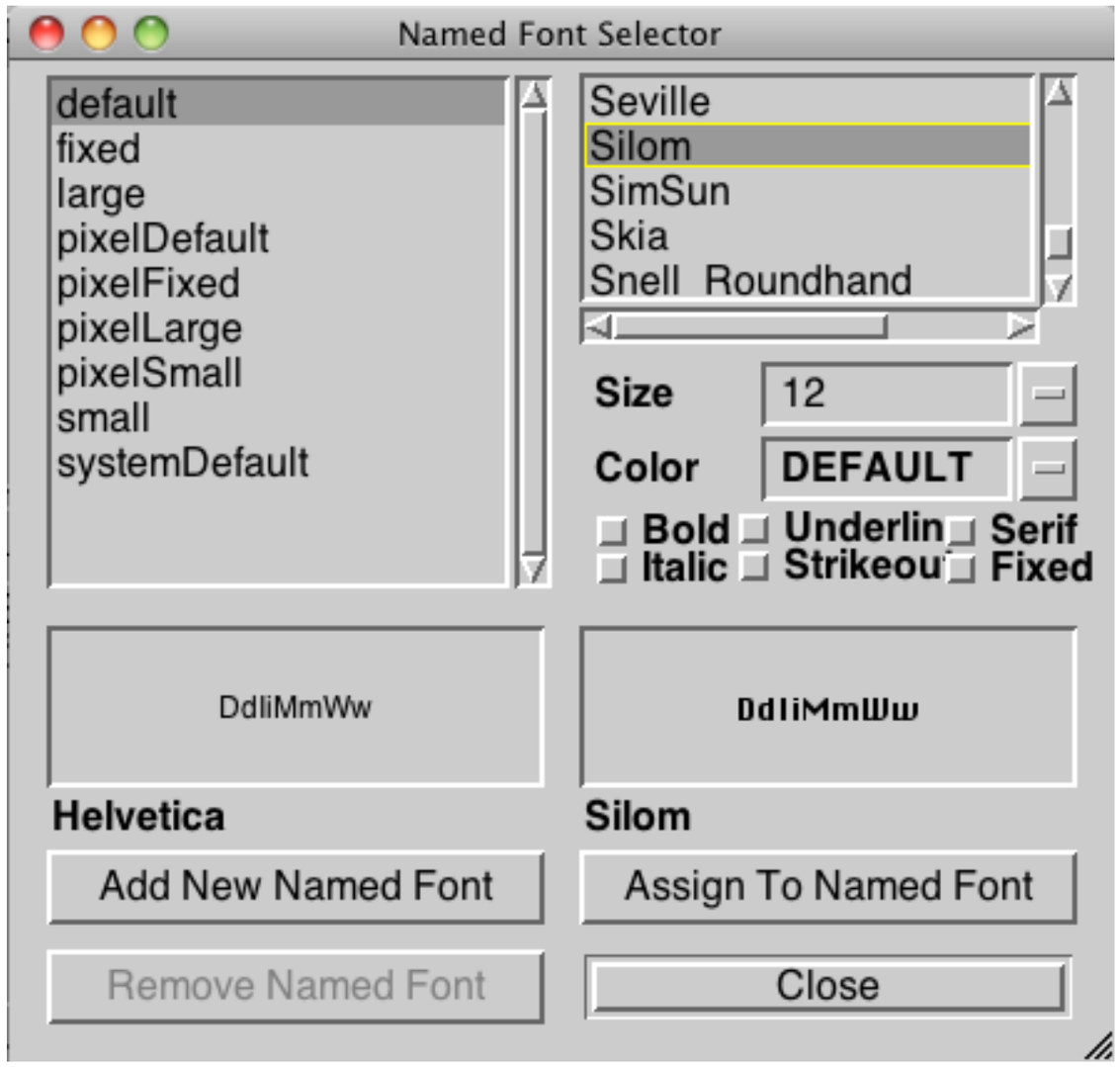
When to use the Mediator Pattern

When a set of objects communicate in a well-defined but complex ways

When reusing an object is difficult because it refers to and communicates with many other objects

When a behavior that's distributed between several classes should be customizable without a lot of subclassing

Classic Mediator Example

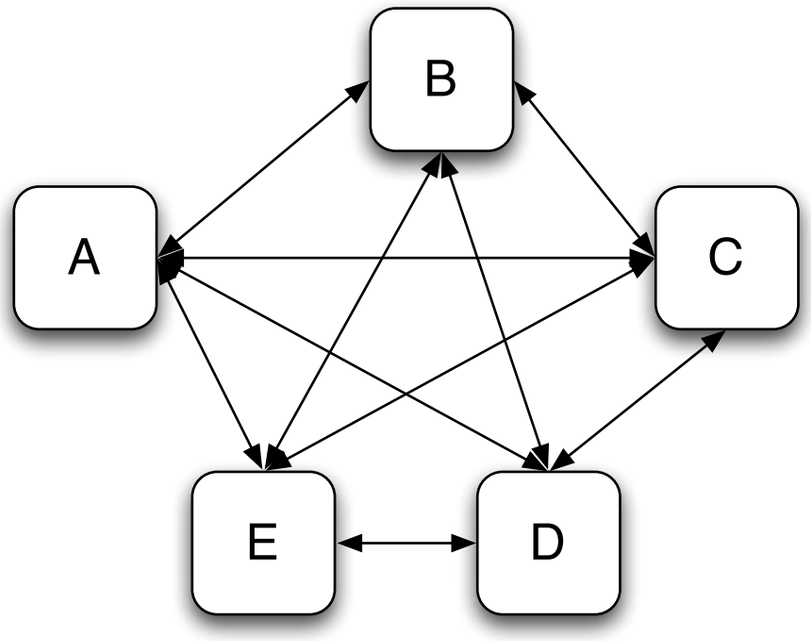


Simpler Example

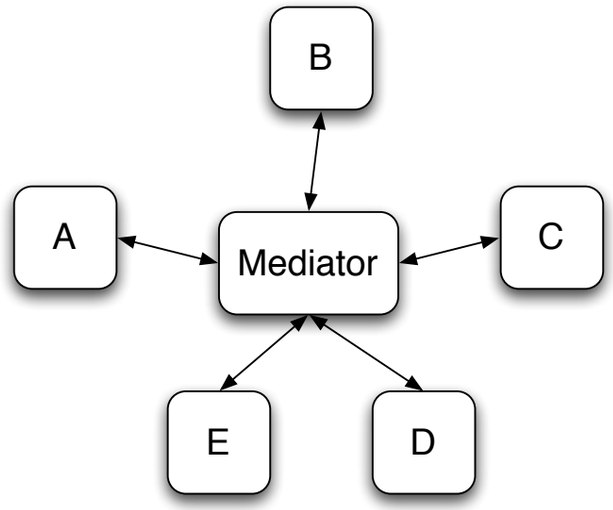
Non Mediator Solution

```
class OKButton extends Button {
   TextField password;
   TextField username;
   Database userData;
   Model application;

   protected void processEvent(AWTEvent e) {
      if (!e.isButtonPressed()) return;
      e.consume();
      if (password.getText() = "") {
   notifyUser("Must enter password");
   return;
}
if (username.getText() = "") {
   notifyUser("Must enter user name");
   return;
}
if (!userData.validUser(password.getText(), username.getTest()))
   notifyUser("Invalid username & password");
   return;
}
```



Mediator Solution




```
class LoginDialog extends Panel {
   TextField password;
   TextField username;
   Database userData;
   Button ok, cancel;

   protected void actionPerformed(ActionEvent e)  {
      if (!e.isButtonPressed() or e.getSource() != ok) return;
      if (password.getText() = "") {
         notifyUser("Must enter password");
         return;
      }
      if (username.getText() = "") {
         notifyUser("Must enter user name");
         return;
      }
      if (!userData.validUser(password.getText(), username.getTest()))
         notifyUser("Invalid username & password");
         return;
      }
```

What is Different?

Non Mediator Example

- Special Button class
- OK button coupled to text fields

Mediator Example

- No specialButton class
- LoginDialog coupled to text fields

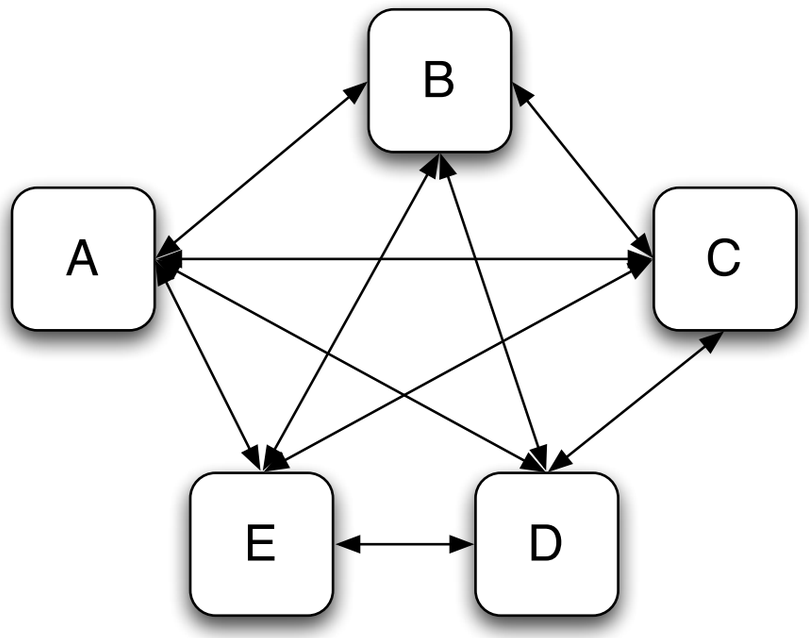
Logic moved from button class to LoginDialog

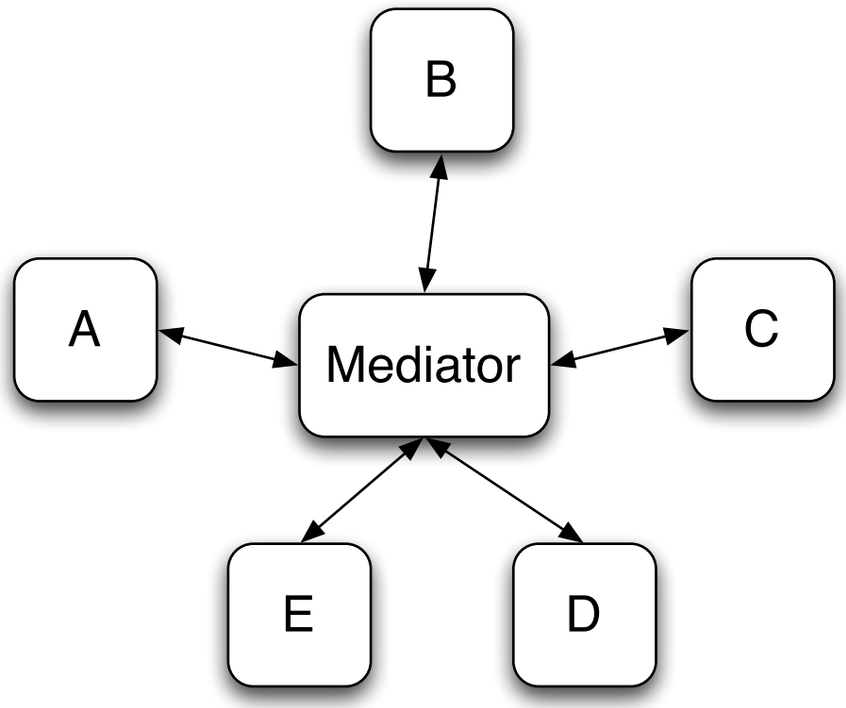
ReactiveX

In some cases ReactiveX reduces mediator to setting up streams

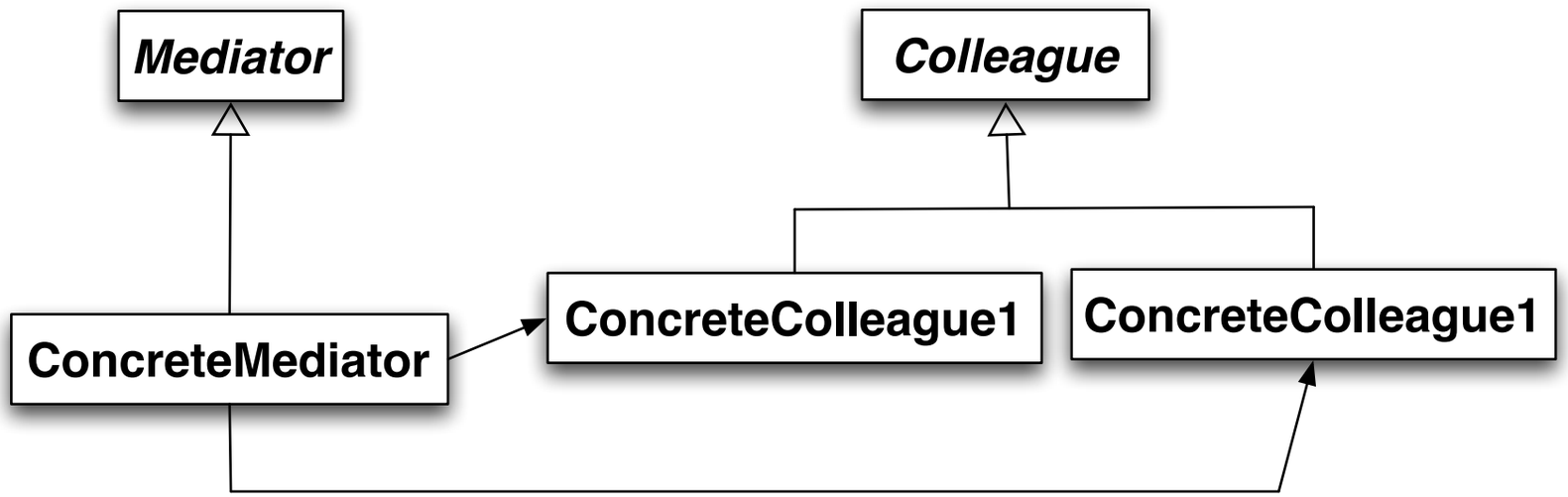
Mediator

A mediator controls and coordinates the interactions of a group of objects





Structure


Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague1

Participants

Mediator

Defines an interface for communicating with Colleague objects

ConcreteMediator

Implements cooperative behavior by coordinating Colleague objects

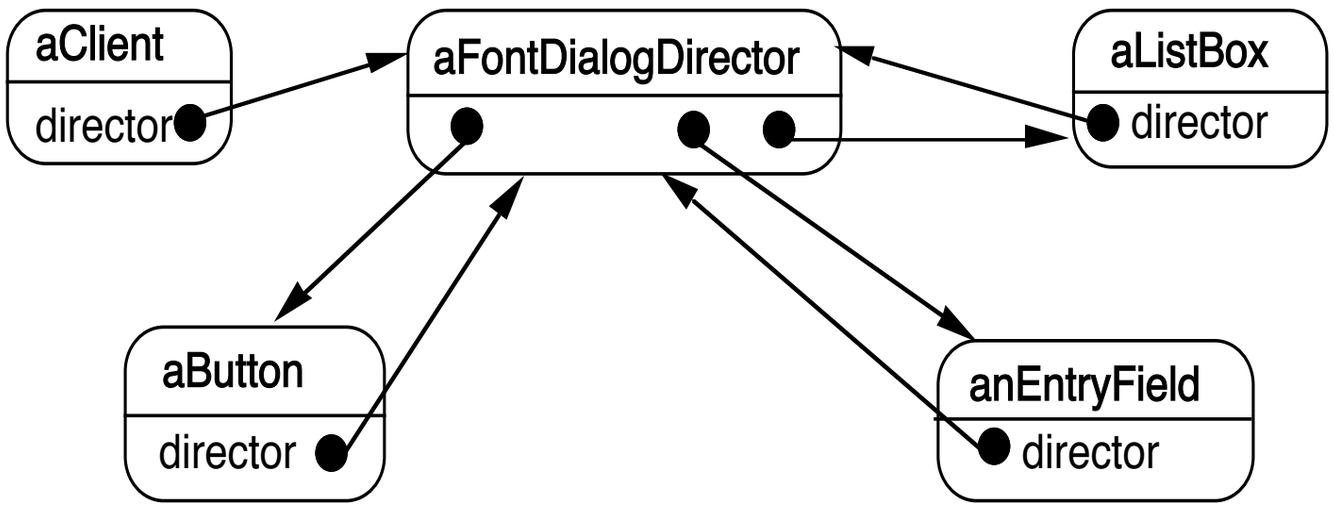
Knows and maintains its colleagues

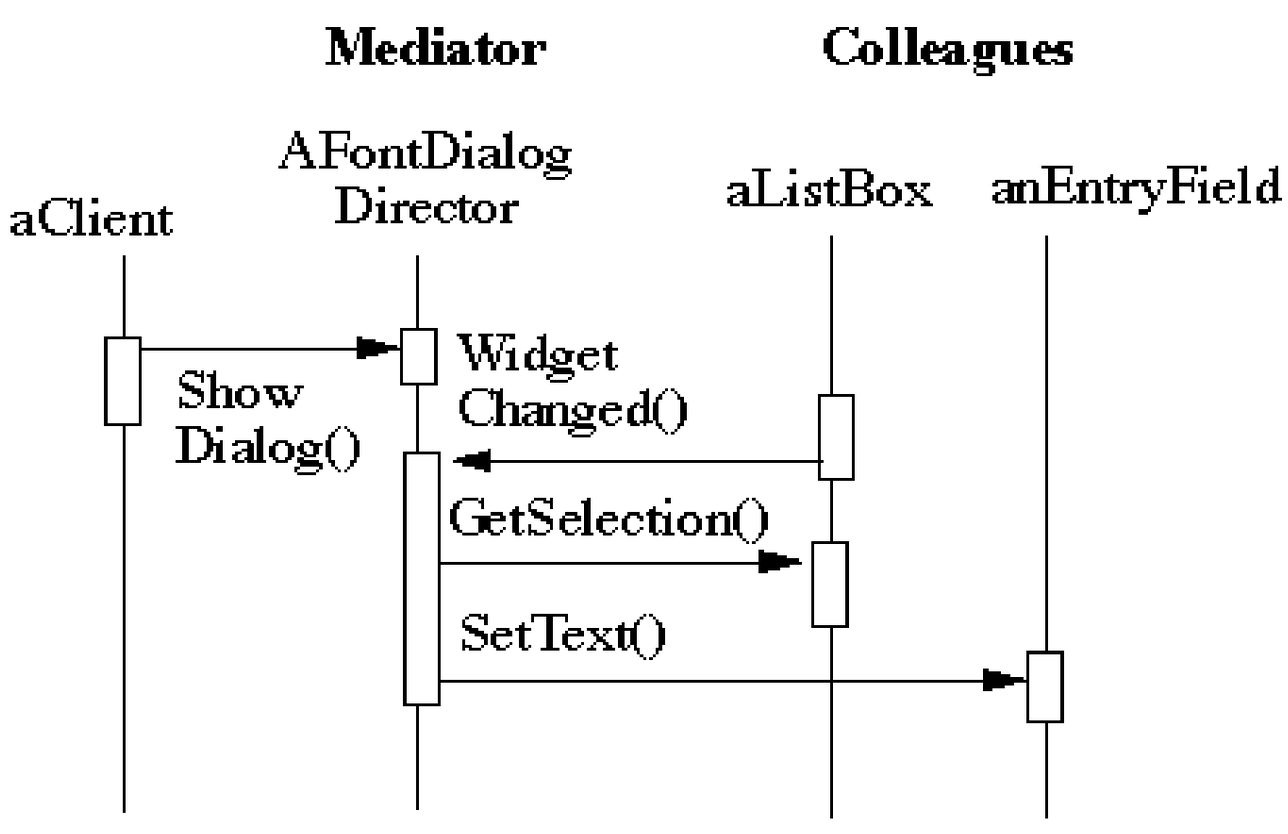
Colleague classes

Each Colleague class knows its Mediator object

Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Motivating Example - Dialog Boxes





How does this differ from a God Class?

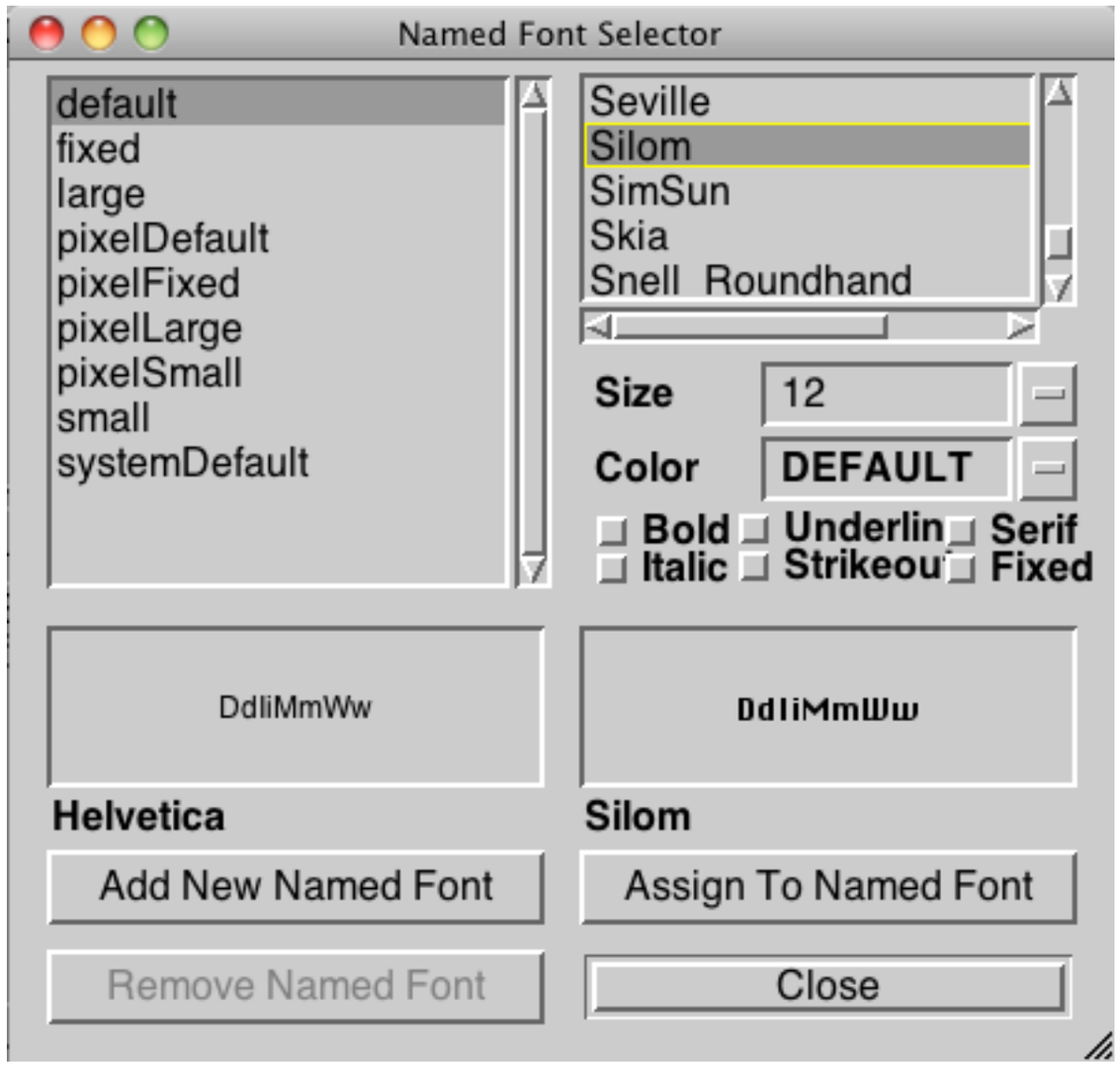
When to use the Mediator Pattern

When a set of objects communicate in a well-defined but complex ways

When reusing an object is difficult because it refers to and communicates with many other objects

When a behavior that's distributed between several classes should be customizable without a lot of subclassing

Classic Mediator Example

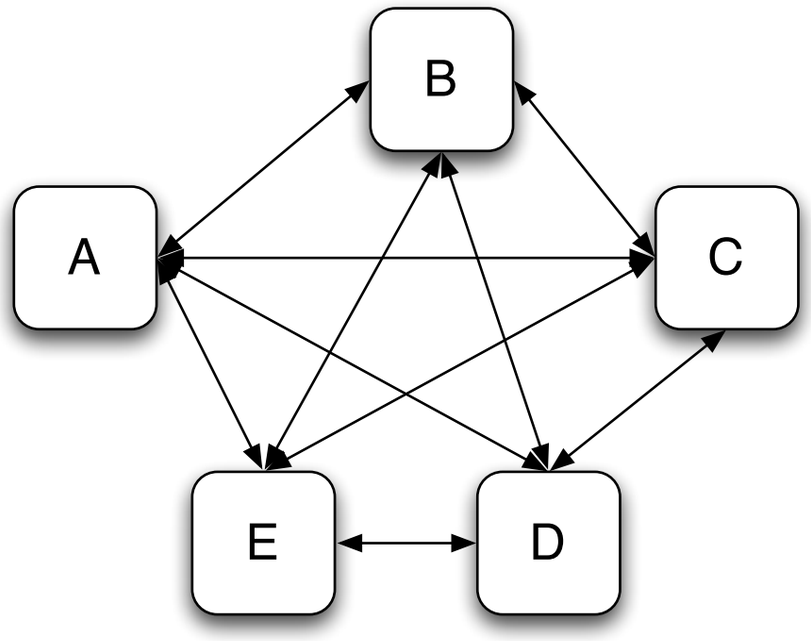


Simpler Example

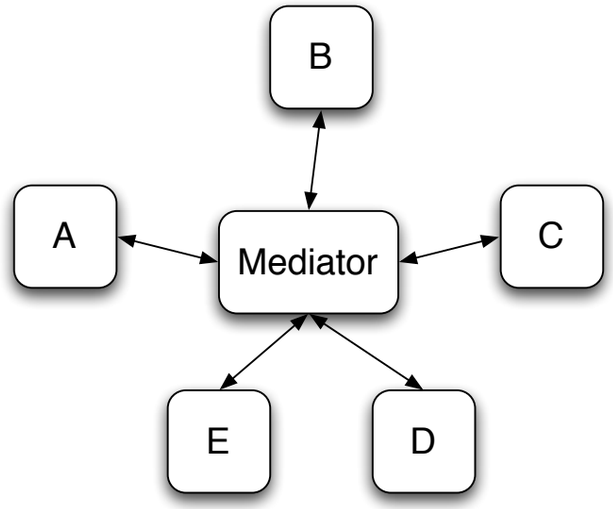
Non Mediator Solution

```
class OKButton extends Button {
   TextField password;
   TextField username;
   Database userData;
   Model application;

   protected void processEvent(AWTEvent e) {
      if (!e.isButtonPressed()) return;
      e.consume();
      if (password.getText() = "") {
   notifyUser("Must enter password");
   return;
}
if (username.getText() = "") {
   notifyUser("Must enter user name");
   return;
}
if (!userData.validUser(password.getText(), username.getTest()))
   notifyUser("Invalid username & password");
   return;
}
```



Mediator Solution




```
class LoginDialog extends Panel {
   TextField password;
   TextField username;
   Database userData;
   Button ok, cancel;

   protected void actionPerformed(ActionEvent e)  {
      if (!e.isButtonPressed() or e.getSource() != ok) return;
      if (password.getText() = "") {
         notifyUser("Must enter password");
         return;
      }
      if (username.getText() = "") {
         notifyUser("Must enter user name");
         return;
      }
      if (!userData.validUser(password.getText(), username.getTest()))
         notifyUser("Invalid username & password");
         return;
      }
```

What is Different?

Non Mediator Example

- Special Button class
- OK button coupled to text fields

Mediator Example

- No specialButton class
- LoginDialog coupled to text fields

Logic moved from button class to LoginDialog

ReactiveX

In some cases ReactiveX reduces mediator to setting up streams

Facade

Size

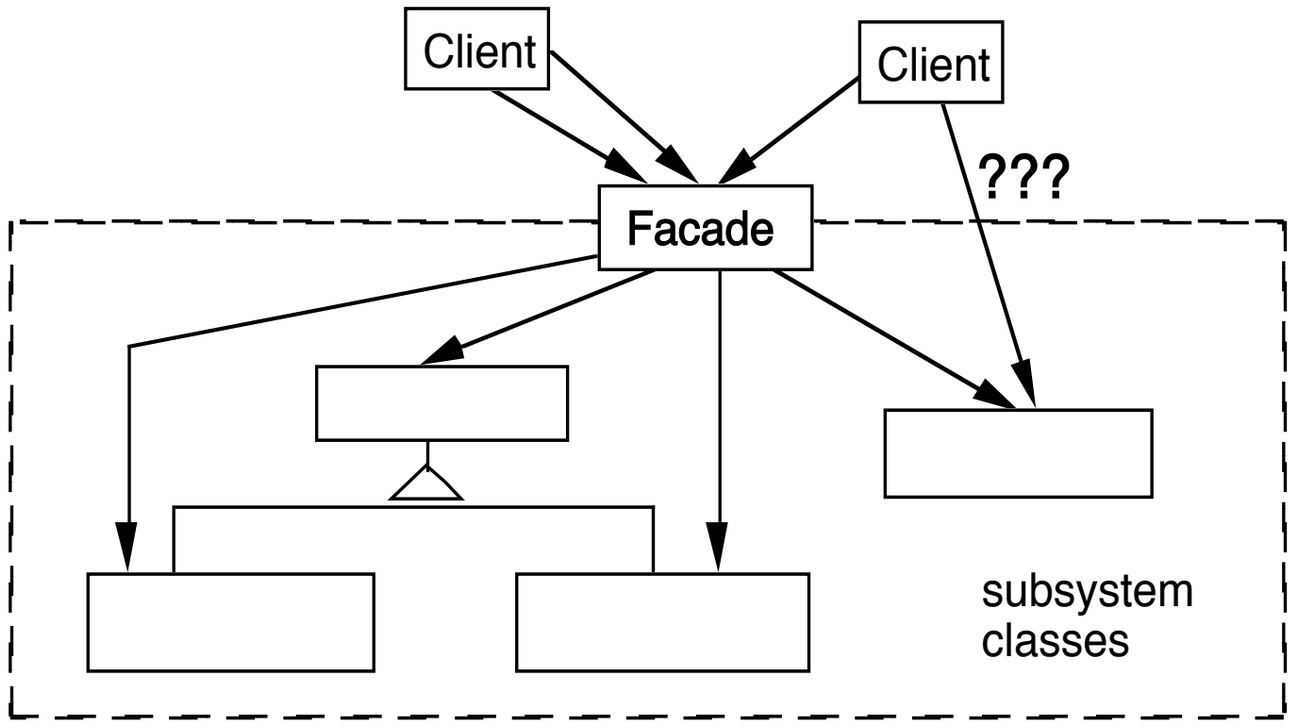
Item	Source Lines of Code (Millions)
F-22 Raptor US jet fighter	1.7
Boeing 787	6.5
Chevy Volt - Embedded Code	10
S-class Mercedes-Benz radio & navigation system	20
Mac OS 10.4	86
New automobile	~100
Debian 5.0	342
Tesla	Linux + ?

Design Patterns text contains under 8,000 lines

The Facade Pattern

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem

Public versus Private Subsystem classes

Some classes of a subsystem are

- public
- facade
- private

Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

Programmers only use Compiler, which uses the other classes

Compiler evaluate: '100 factorial'

```
| method compiler |
method := 'reset
    "Resets the counter to zero"
    count := 0.'.

compiler := Compiler new.
compiler
    parse:method
    in: Counter
    notifying: nil
```

Objective-C Class Clusters & Facade