CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 18 Command Processor, Memento, Mediator, Facade
Nov 5, 2020

Undo

Undo

Some examples

Counter

```
counter.increase();         //increase counter by 1
counter.decrease();         //decrease counter by 1
```

Undo

Some examples

- Text editing
  - Replace "Should" with "Could" at start of 3rd sentence in 5 paragraph

Undo - Some Issues

Redo

Multiple undo

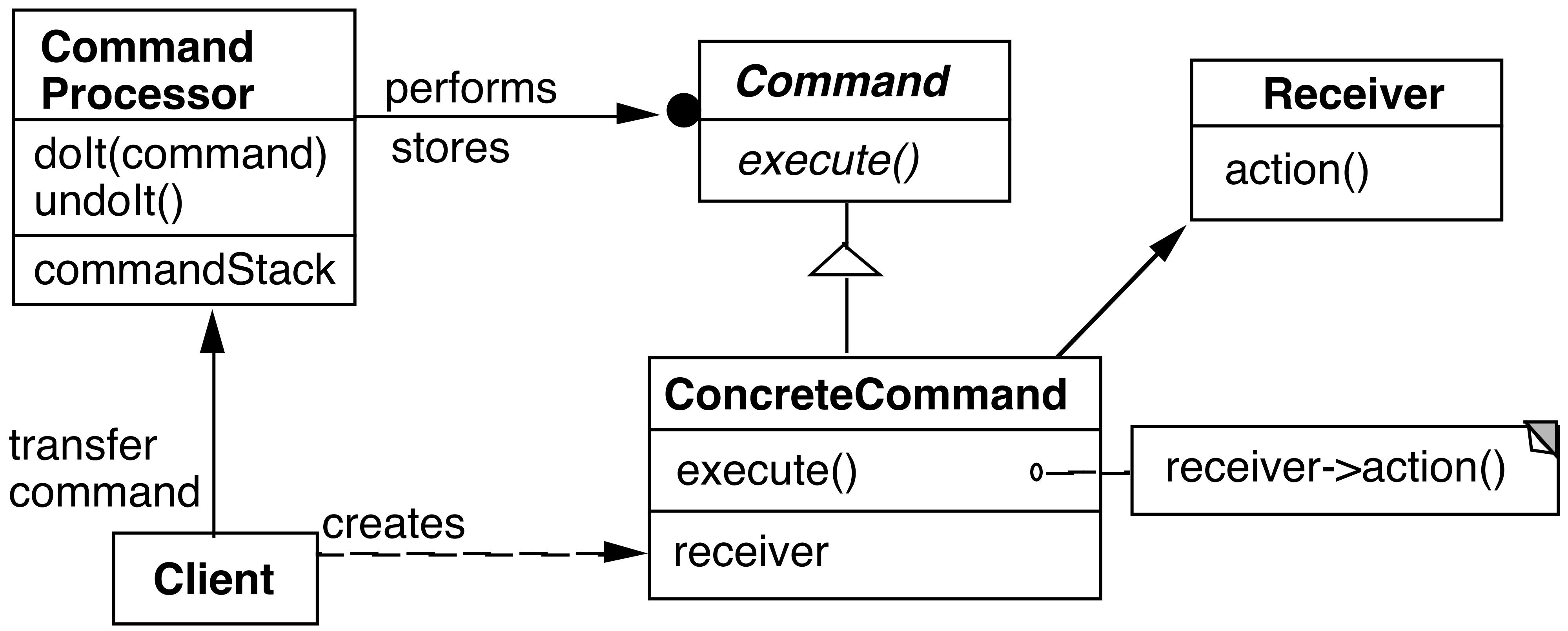
Command Processor Pattern

Command Processor manages the command objects

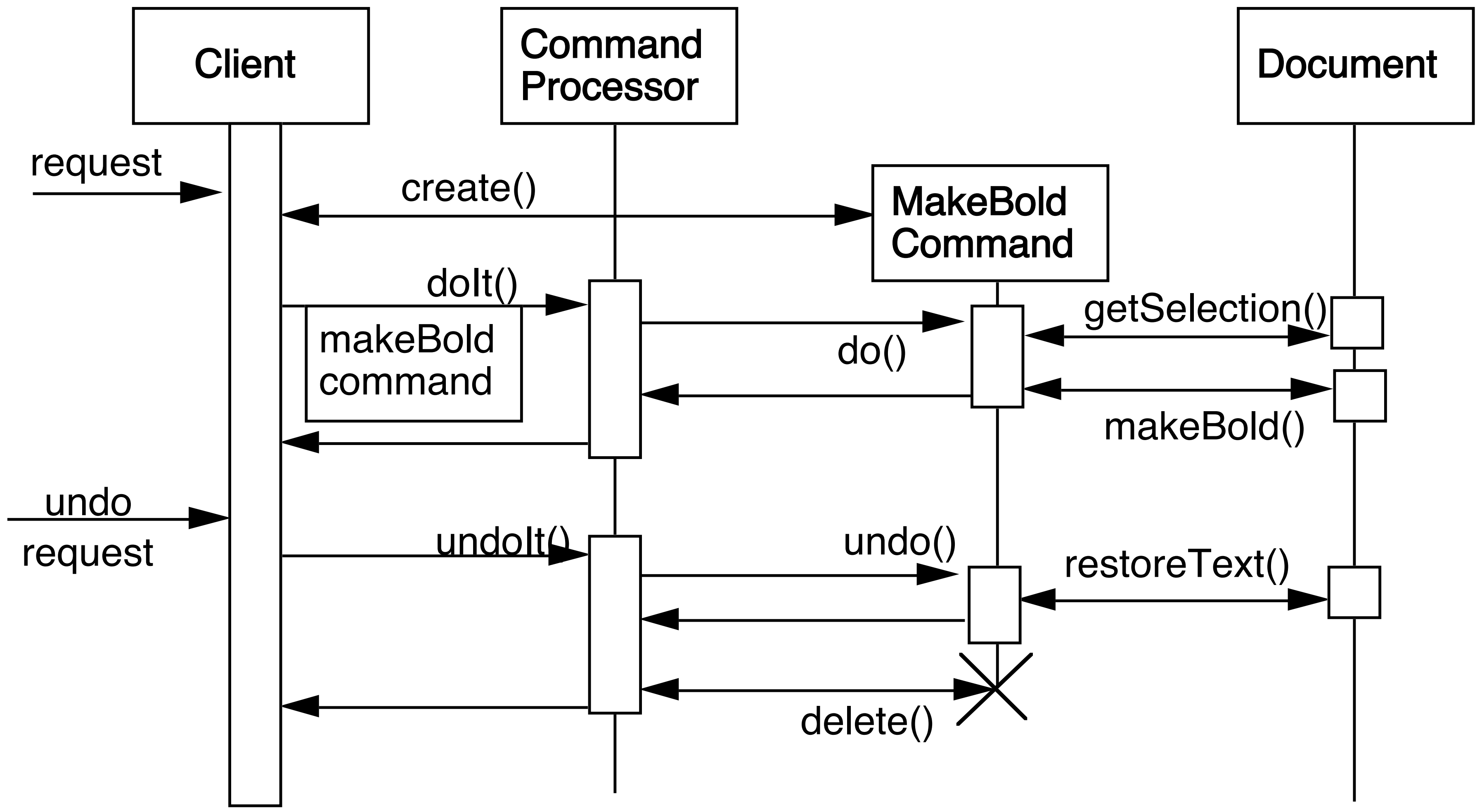
The command processor:

- Contains all command objects
- Schedules the execution of commands
- May store the commands for later undo
- May log the sequence of commands for testing purposes
- Uses singleton to insure only one instance

Structure


Command Processor
doIt(command)
undoIt()
commandStack
performs
stores
Command
execute()
Receiver
action()
ConcreteCommand
execute()
receiver
receiver->action()
transfer command
creates
Client

Dynamics



Client
Command Processor
Document
MakeBold Command
request
create()
doIt()
makeBold command
do()
getSelection()
makeBold()
undo request
undoIt()
undo()
restoreText()
delete()

Benefits

Flexibility in the way requests are activated

- Different user interface elements can generate the same kind of command object
- Allows the user to configure commands performed by a user interface element

Flexibility in the number and functionality of requests

- Adding new commands and providing for a macro language comes easy

Programming execution-related services

- Commands can be stored for later replay
- Commands can be logged
- Commands can be rolled back

Testability at application level

Concurrency

- Allows for the execution of commands in separate threads

Liabilities

Efficiency loss

Potential for an excessive number of command classes

- Try reducing the number of command classes by:
  - Grouping commands around abstractions
  - Unifying simple commands classes by passing the receiver object as a parameter

Complexity

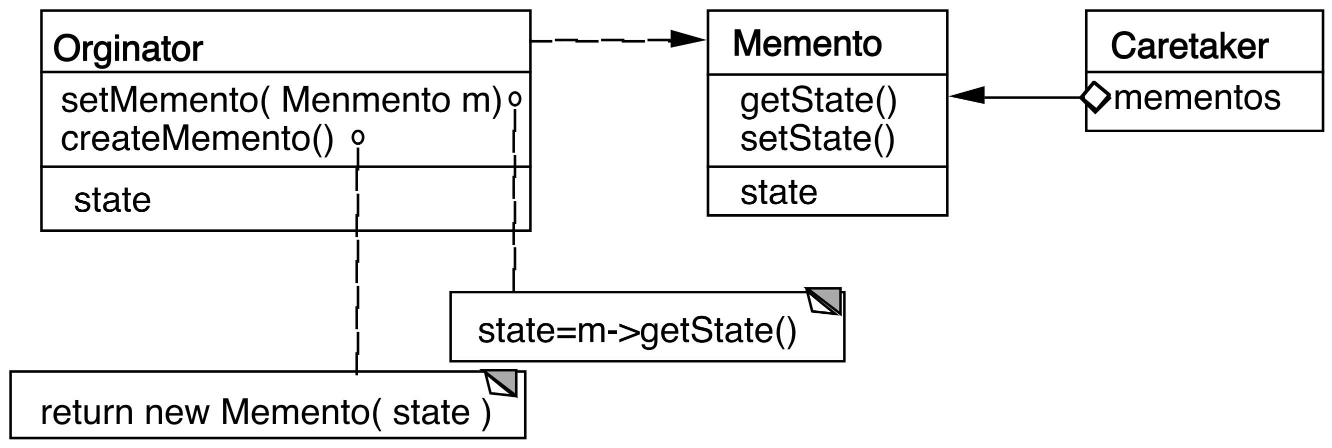
- How do commands get additional parameters they need?

Memento

Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

undo, rollbacks



Only originator:

Can access Memento’s get/set state methods

Create Memento

Example

```
package Examples;
class Memento{
    private Hashtable savedState = new Hashtable();

    protected Memento() {}; //Give some protection

    protected void setState( String stateName, Object stateValue ) {
        savedState.put( stateName, stateValue );
    }

    protected Object getState( String stateName) {
        return savedState.get( stateName);
    }

    protected Object getState(String stateName, Object defaultValue ) {
        if ( savedState.containsKey( stateName ) )
            return savedState.get( stateName);
        else
            return defaultValue;
    }
}
```

Sample Originator

```
package Examples;
class ComplexObject {
    private String name;
    private int someData;
    private Vector objectAsState = new Vector();

    public Memento createMemento() {
        Memento currentState = new Memento();
        currentState.setState( "name", name );
        currentState.setState( "someData", new Integer(someData) );
        currentState.setState( "objectAsState", objectAsState.clone() );
        return currentState;
    }

    public void restoreState( Memento oldState) {
        name = (String) oldState.getState( "name", name );
        objectAsState = (Vector) oldState.getState( "objectAsState" );
        Integer data = (Integer) oldState.getState( "someData");
        someData = data.intValue();
    }
```

Why not let the Originator save its old state?

```
class ComplexObject {
   private String name;
   private int someData;
   private Vector objectAsState = new Vector();
   private Stack history;

   public createMemento() {
      Memento currentState = new Memento();
      currentState.setState( "name", name );
      currentState.setState( "someData", new Integer(someData) );
      currentState.setState( "objectAsState", objectAsState.clone() );
      history.push(currentState);
   }

   public void restoreState() {
      Memento oldState = history.pop();
      name = (String) oldState.getState( "name", name );
      objectAsState = (Vector) oldState.getState( "objectAsState" );
      Integer data = (Integer) oldState.getState( "someData");
      someData = data.intValue();
   }
```

Some Consequences

Expensive

  Space

Narrow & Wide interfaces - Keep data hidden

```
Class Memento {
   public:
      virtual ~Memento();
   private:
      friend class Originator;
      Memento();
      void setState(State*);
      State* GetState();
```

```
class Originator {
   private String state;

   private class Memento {
      private String state;
      public Memento(String stateToSave)
         { state = stateToSave; }
     public String getState() { return state; }
      }

      public Object memento()
         { return new Memento(state);}
```

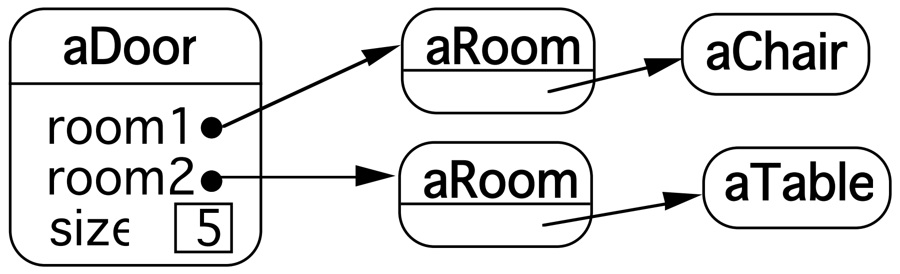
Using Clone to Save State

```
interface Memento extends Cloneable { }

class ComplexObject implements Memento {
    private String name;
    private int someData;

    public Memento createMemento() {
        Memento myState = null;
        try {
            myState =  (Memento) this.clone();
        }
        catch (CloneNotSupportedException notReachable) {
        }
        return myState;
    }

    public void  restoreState( Memento savedState) {
        ComplexObject myNewState = (ComplexObject)savedState;
        name = myNewState.name;
        someData = myNewState.someData;
    }
}
```

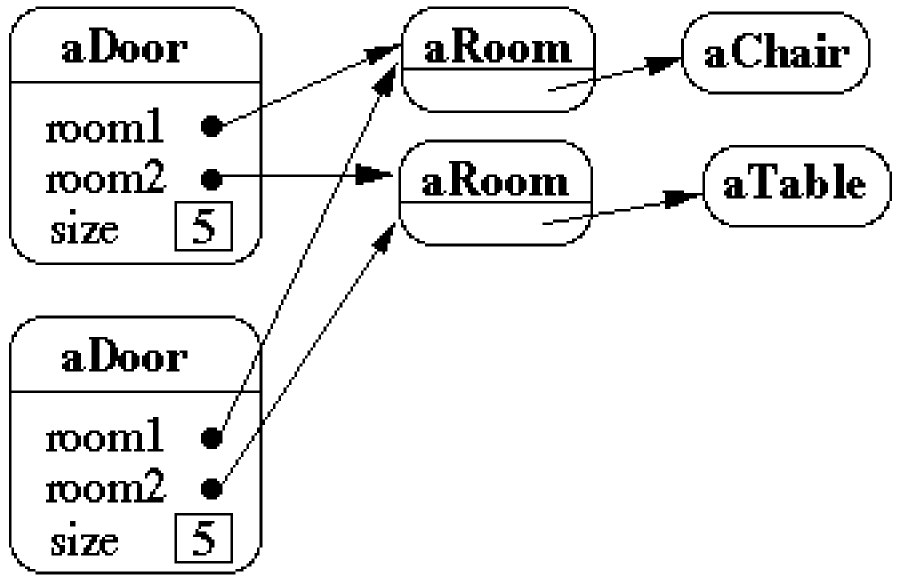
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

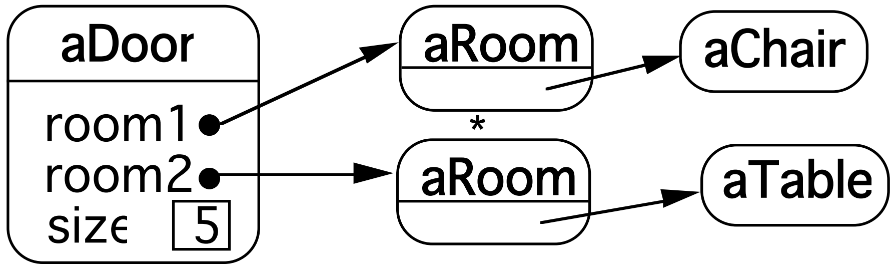


Shallow Copy

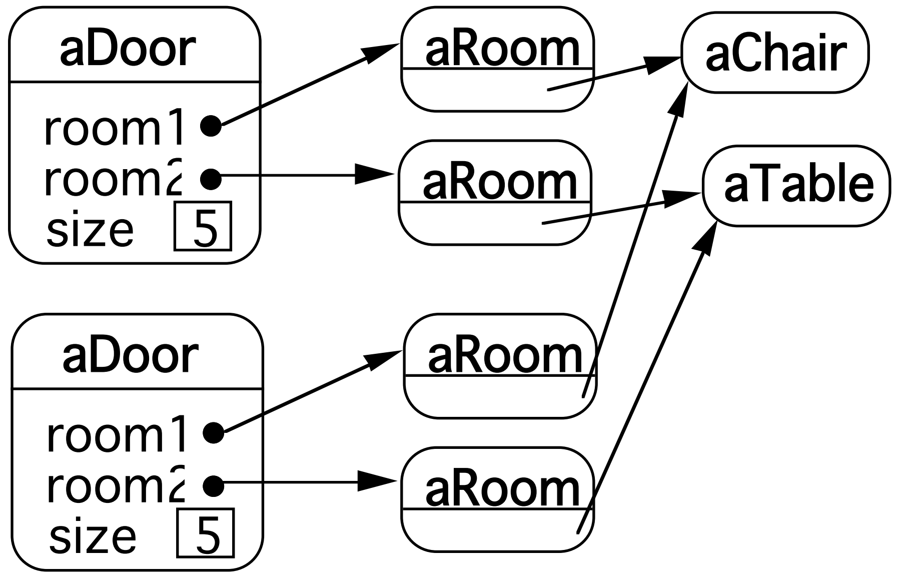


Shallow Copy Verse Deep Copy

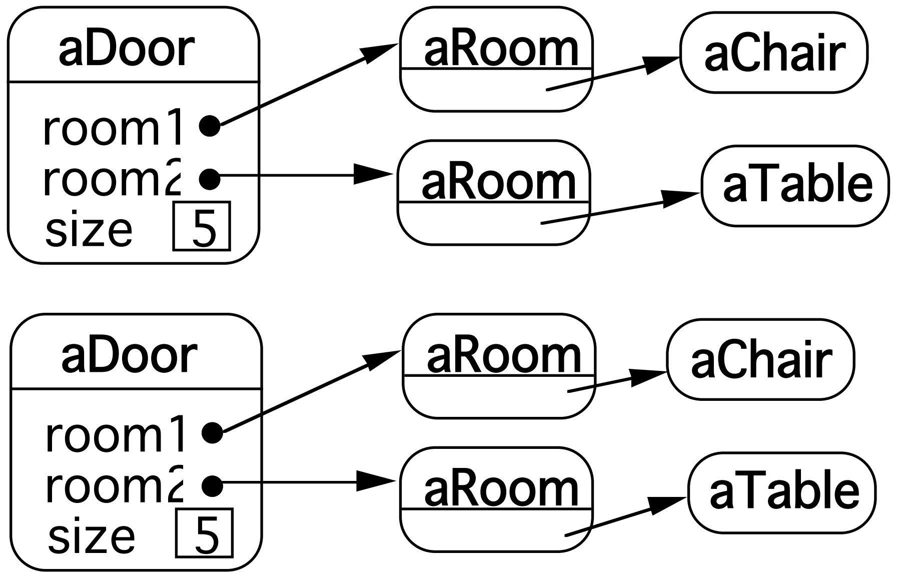
Original Objects



Deep Copy



Deeper Copy



Cloning Issues - C++ Copy Constructors

```
class Door {
    public:
        Door();
        Door( const Door&);
        virtual Door* clone() const;

        virtual void Initialize( Room*, Room* );
        // stuff not shown
    private:
        Room* room1;
        Room* room2;
    }

Door::Door ( const Door& other ) //Copy constructor {
    room1 = other.room1;
    room2 = other.room2;
    }

Door* Door::clone()  const {
    return new Door( *this );
    }
```

Cloning Issues - Java Clone

Shallow Copy

```
class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws  CloneNotSupportedException {
        return super.clone();
    }
}
```

Deep Copy

```
public class Door implements Cloneable {
    private Room room1;
    private Room room2;

    public Object clone() throws CloneNotSupportedException {
        Door thisCloned =(Door) super.clone();
        thisCloned.room1 = (Room)room1.clone();
        thisCloned.room2 = (Room)room2.clone();
        return thisCloned;
    }
}
```

What if Protocol

When there are complex validations or
performing operations that make it difficult to restore later

Make a copy of the Originator

Perform operations on the copy

Check if operations invalidate the internal state of copy

If so discard the copy & raise an exception

Else perform the operations on the Originator

Memento & Functional Programming

- Immutable data
  - Data that can not change
  - Functional languages have primarily immutable data

- If data can not change
  - Don’t need memento pattern

Datomic

Database system where all data is immutable

Transactions become easy

Read and writes become independent

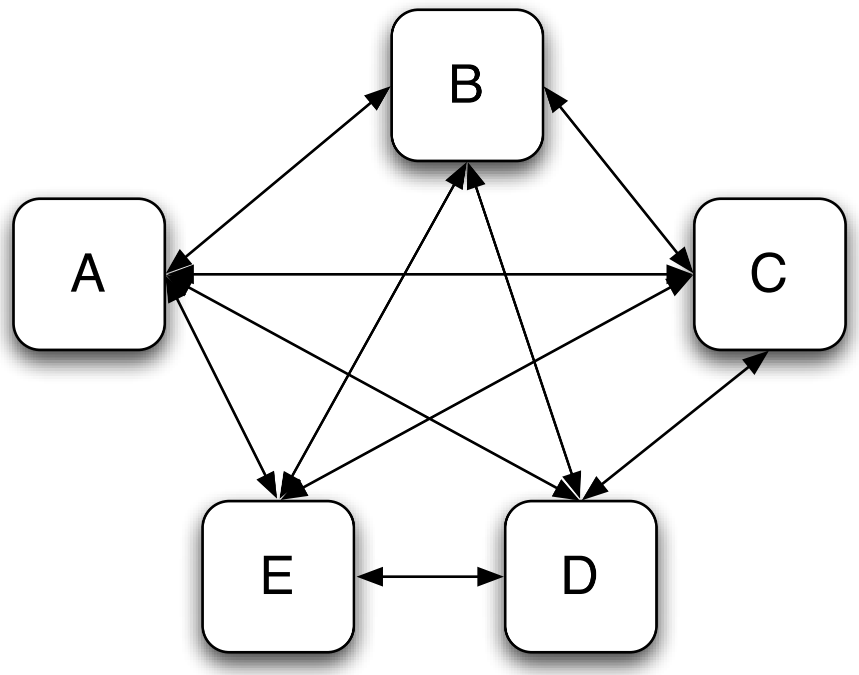
Historical data,
    role backs are easy

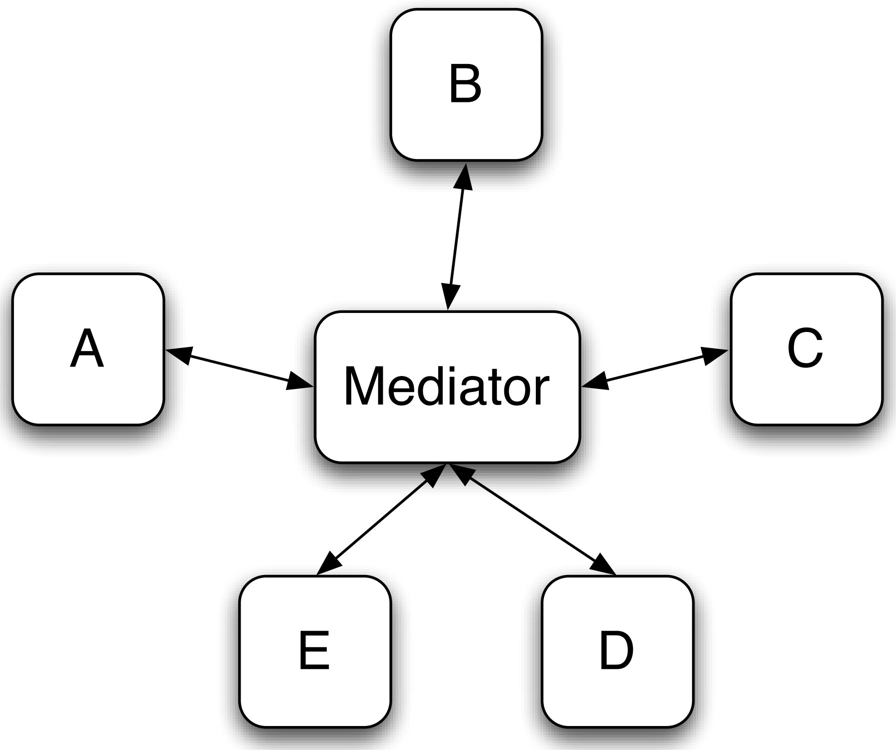
Auditability

Mediator

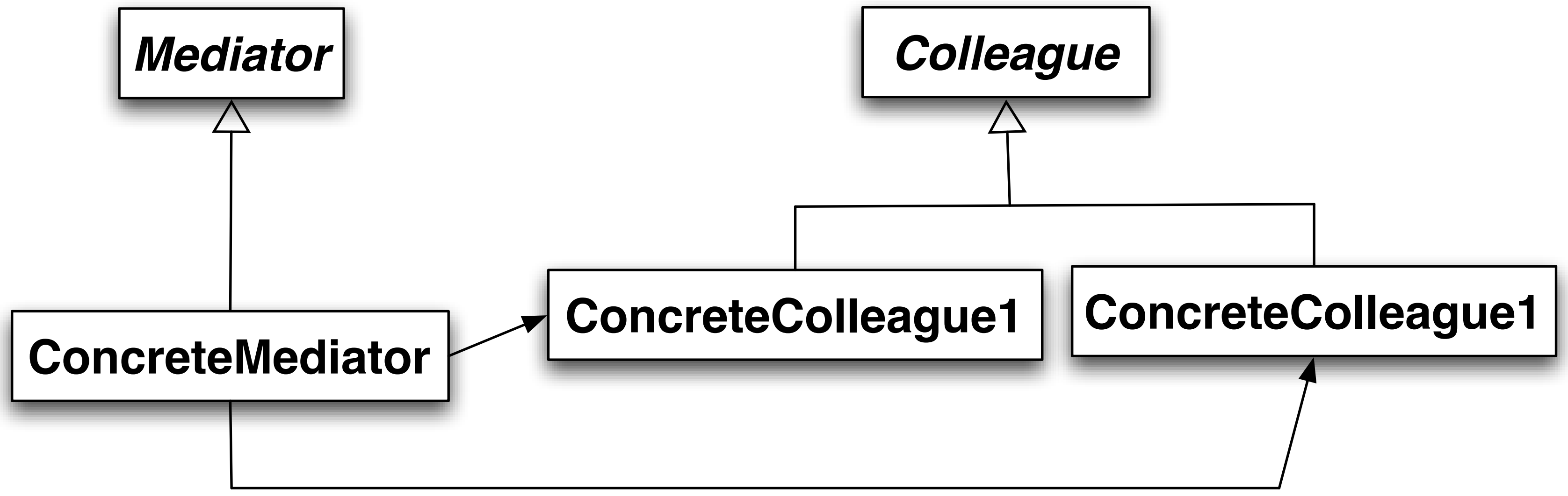
Mediator

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

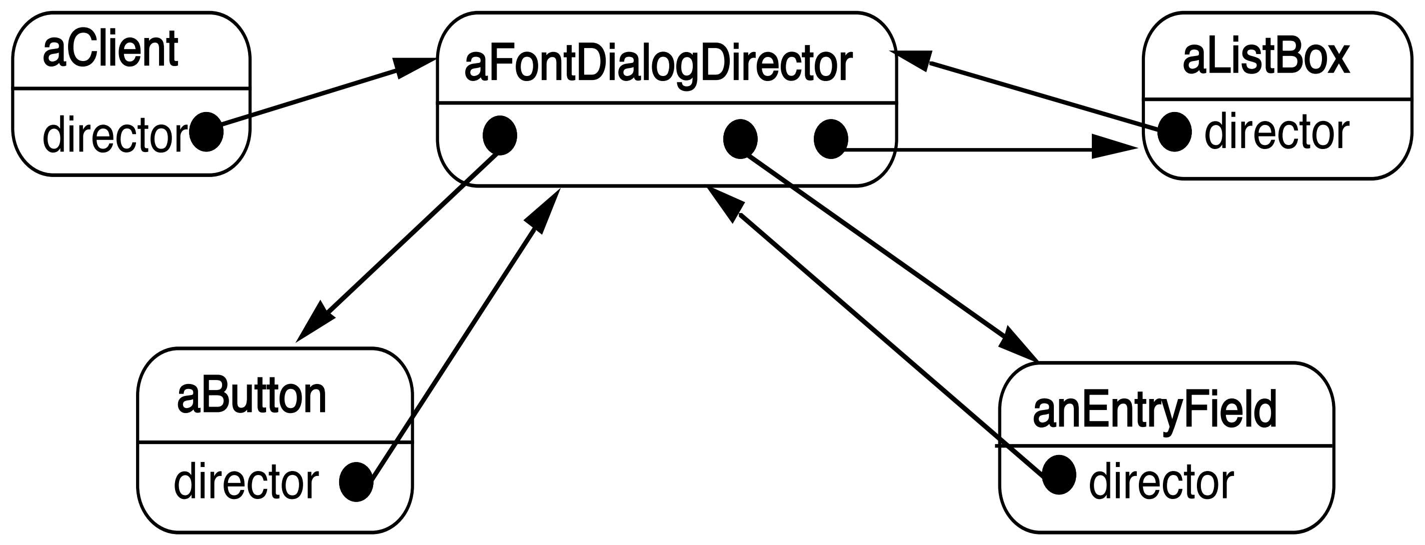
Knows and maintains its colleagues

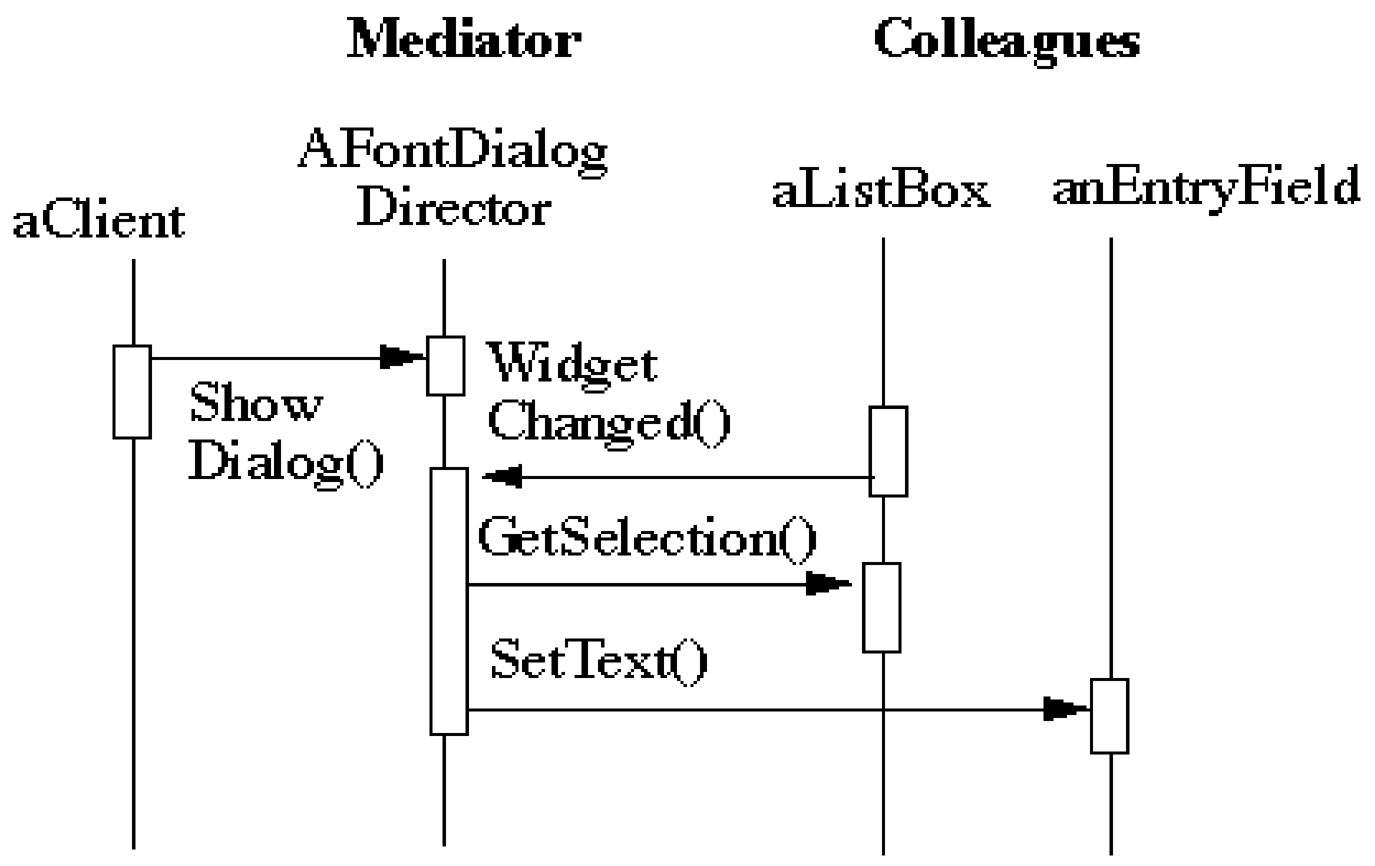
Colleague classes

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?

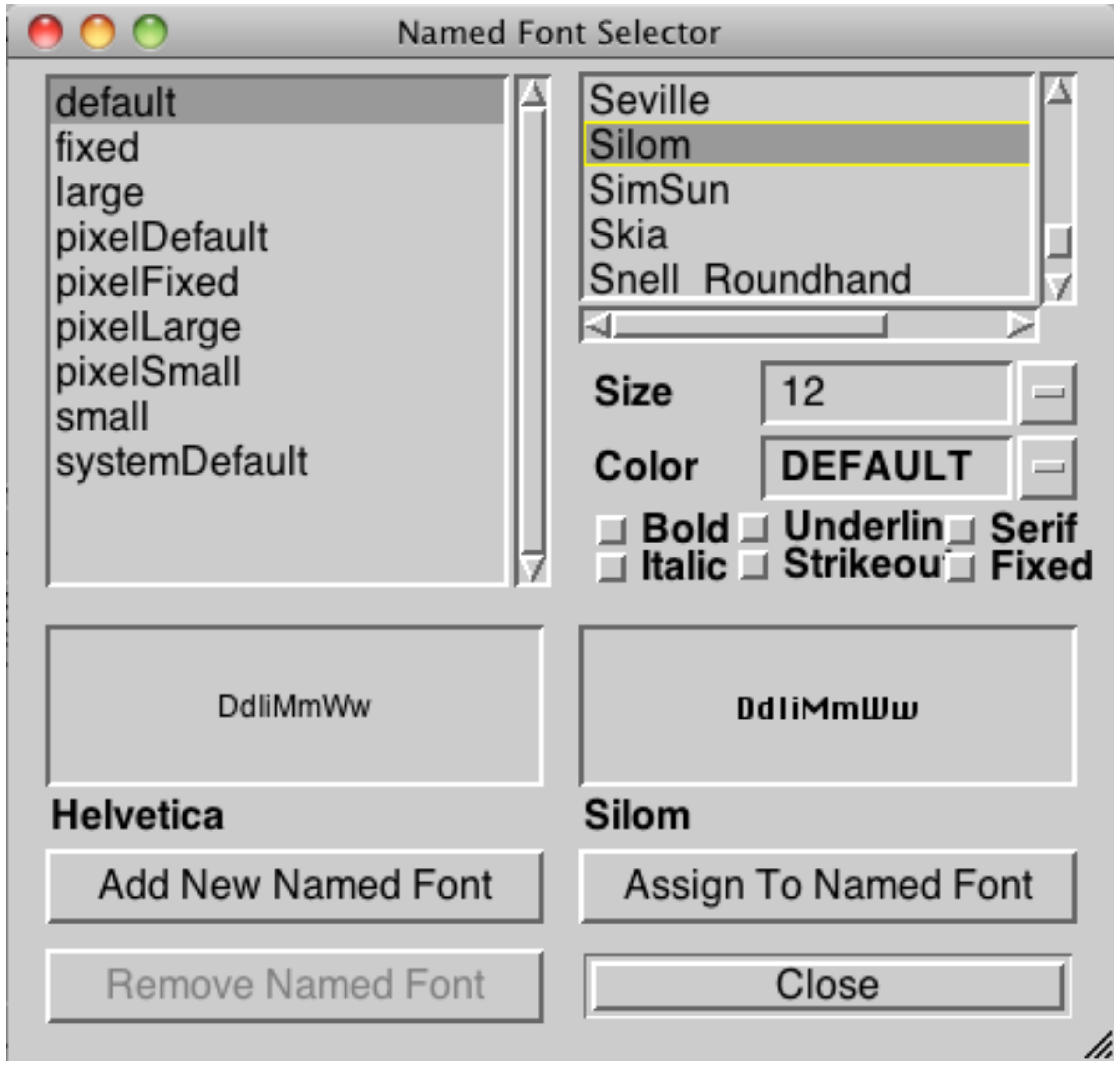
When to use the Mediator Pattern

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

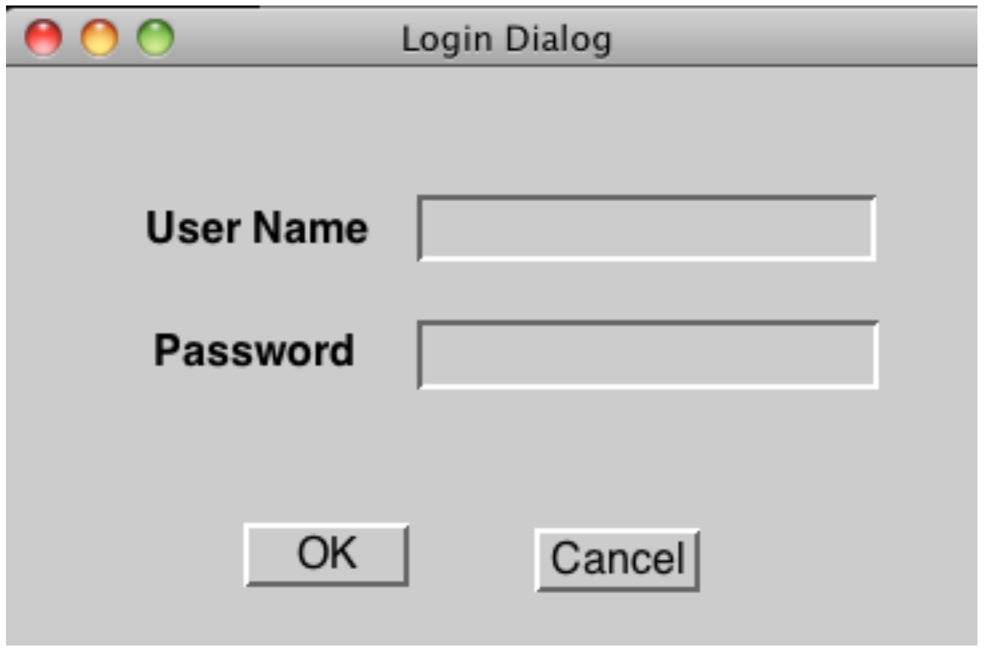
Classic Mediator Example



Size 12

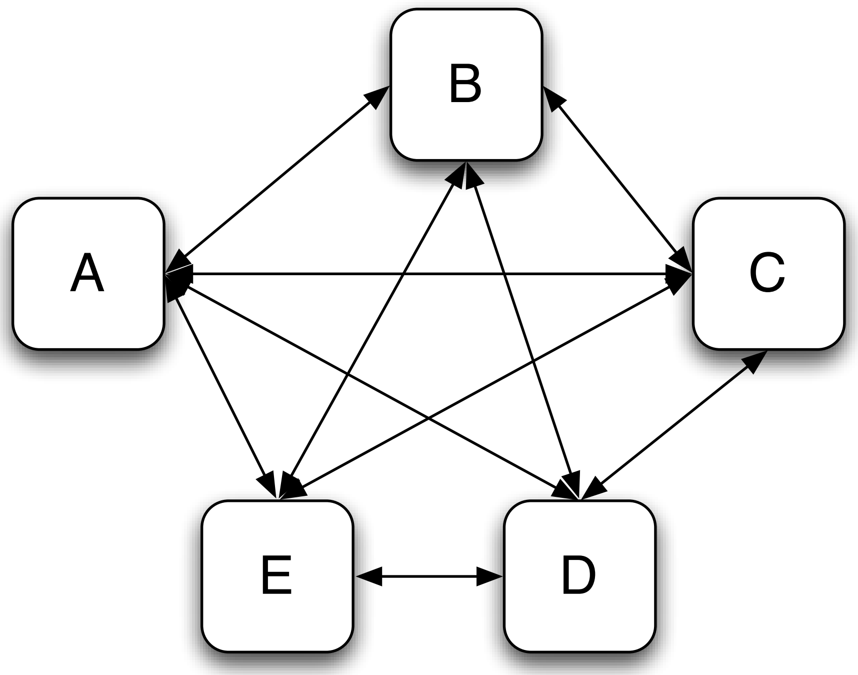
Color DEFAULT

Simpler Example

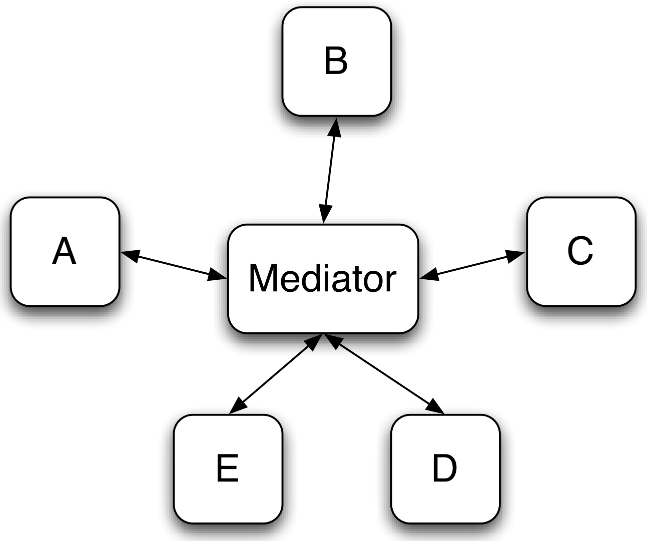


Non Mediator Solution

```
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	Mediator Example
Special Button class	No specialButton class
OK button coupled to text fields	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

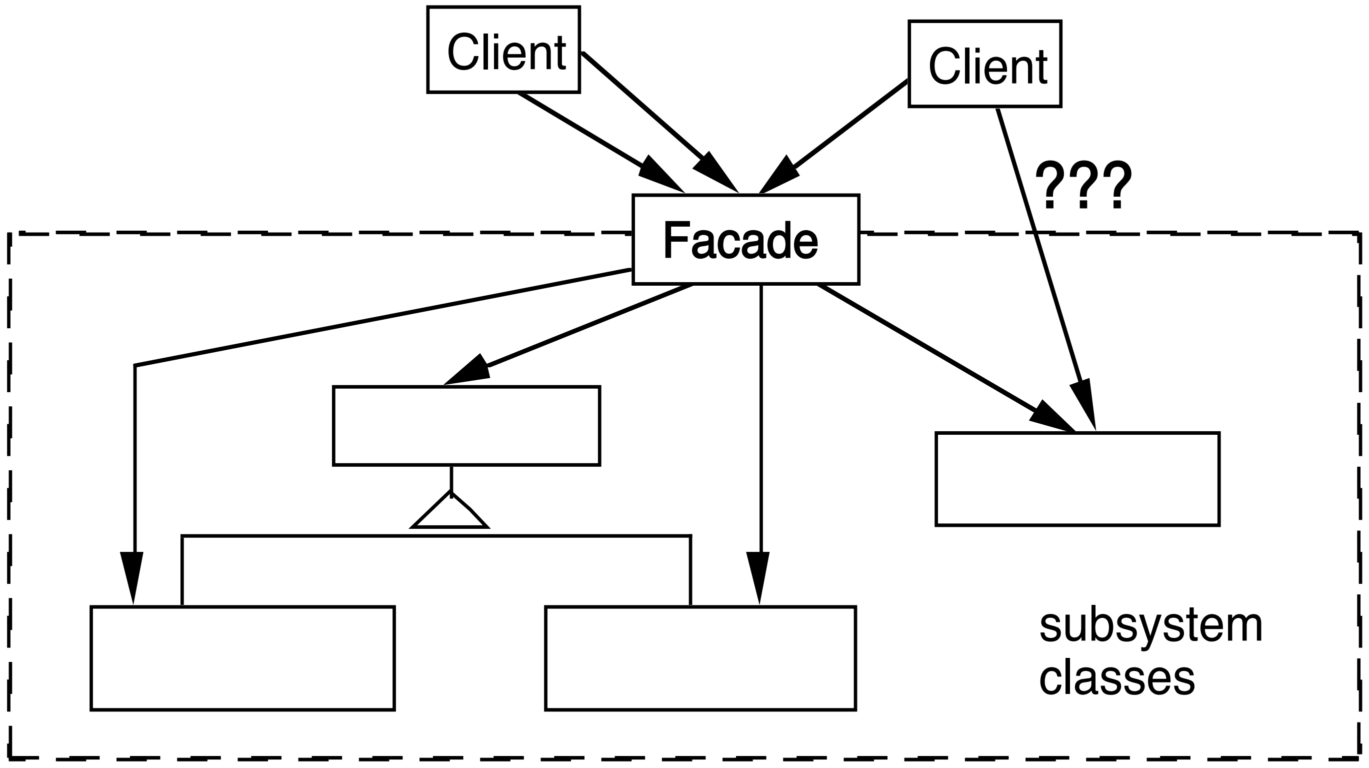
Item	Source Lines of Code (Millions)
F-22 Raptor US jet fighter	1.7
Boeing 787	6.5
Chevy Volt - Embedded Code	10
S-class Mercedes-Benz radio & navigation system	20
Mac OS 10.4	86
New automobile	~100
Debian 5.0	342
Tesla	Linux + ?

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

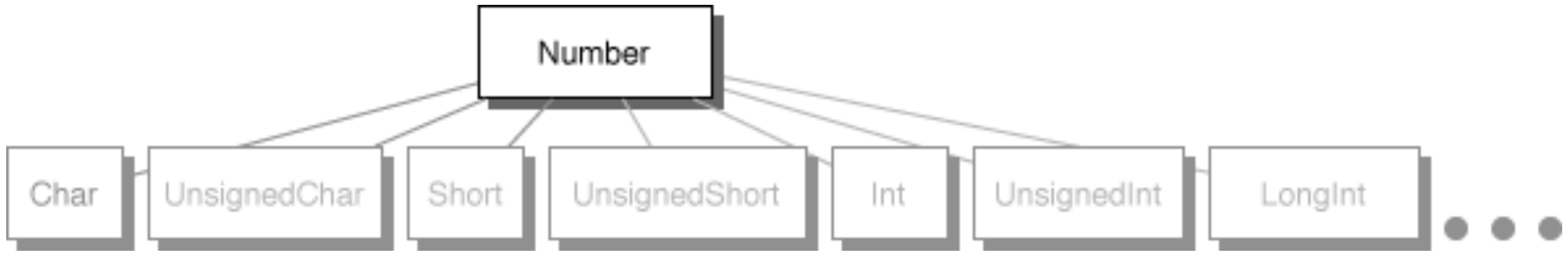
The VisualWorks Smalltalk compiler system has 75 classes

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


Number
Char
UnsignedChar
Short
UnsignedShort
Int
UnsignedInt
LongInt
• • •