Implementing Design Patterns Using Java

St. Louis Java Special Interest Group

Eric M. Burke

Object Computing, Inc.

Presented on July 9, 1998

(updated July 14, 1998)

What are Design Patterns?

- A **Design Pattern** systematically names, explains, and evaluates an important and recurring design in object-oriented systems.
  - **Patterns** are also used outside of the software profession
- Software Design Patterns…
  - are typically targeted toward object-oriented development
  - describe time-proven ways in which good OO concepts can be used to solve common problems
    - encapsulation, inheritance, polymorphism
  - satisfy specific application needs
  - patterns are not domain specific

Four essential elements

- Pattern Name
  - a word or two to describe the pattern
  - allows designers to communicate using a common vocabulary
- Problem
  - describes where to apply the pattern
- Solution
  - describes the elements that make up a design
  - does not describe a particular, concrete solution
- Consequences
  - results and trade-offs of applying the pattern

3 Basic Types of Patterns

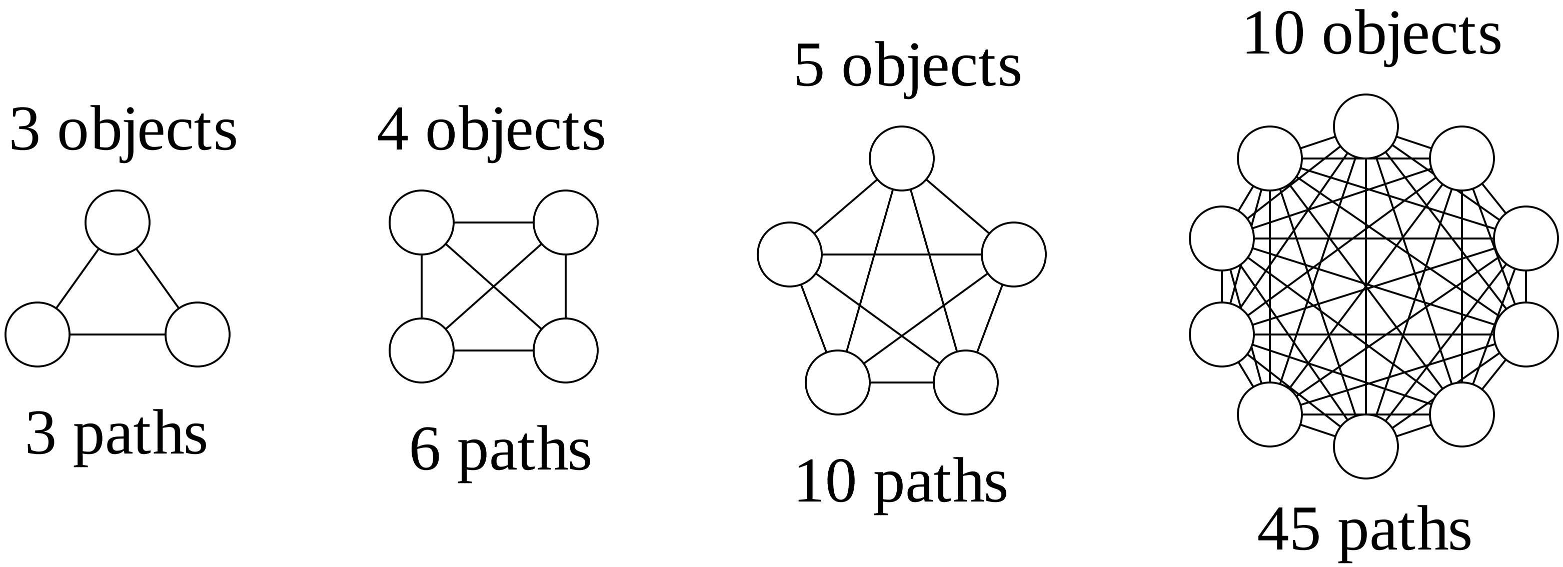
- Creational
  - a family of patterns which abstract the creation of new objects
- Structural
  - describe how to compose groups of cooperating objects into larger systems
- Behavioral
  - characterize patterns of communication between a system of objects

How are Patterns Used?

- Salvaging legacy systems
  - the **Facade** pattern can be used to simplify the interface to a complex legacy system
  - the **Adapter** pattern allows designers to adapt the interface of an existing class or application to work with newer code
- Distributed applications
  - the **Observable** pattern is described in detail later in this presentation
- Object-Oriented class libraries
  - **Iterator** provides a way to access the elements of a collection without violating encapsulation

The Spaghetti Problem

- As the number of objects increases, the number of potential communication paths increases **geometrically**
- If structural and behavioral patterns are not used, code quickly becomes too complex to understand



3 objects

3 paths

4 objects

6 paths

5 objects

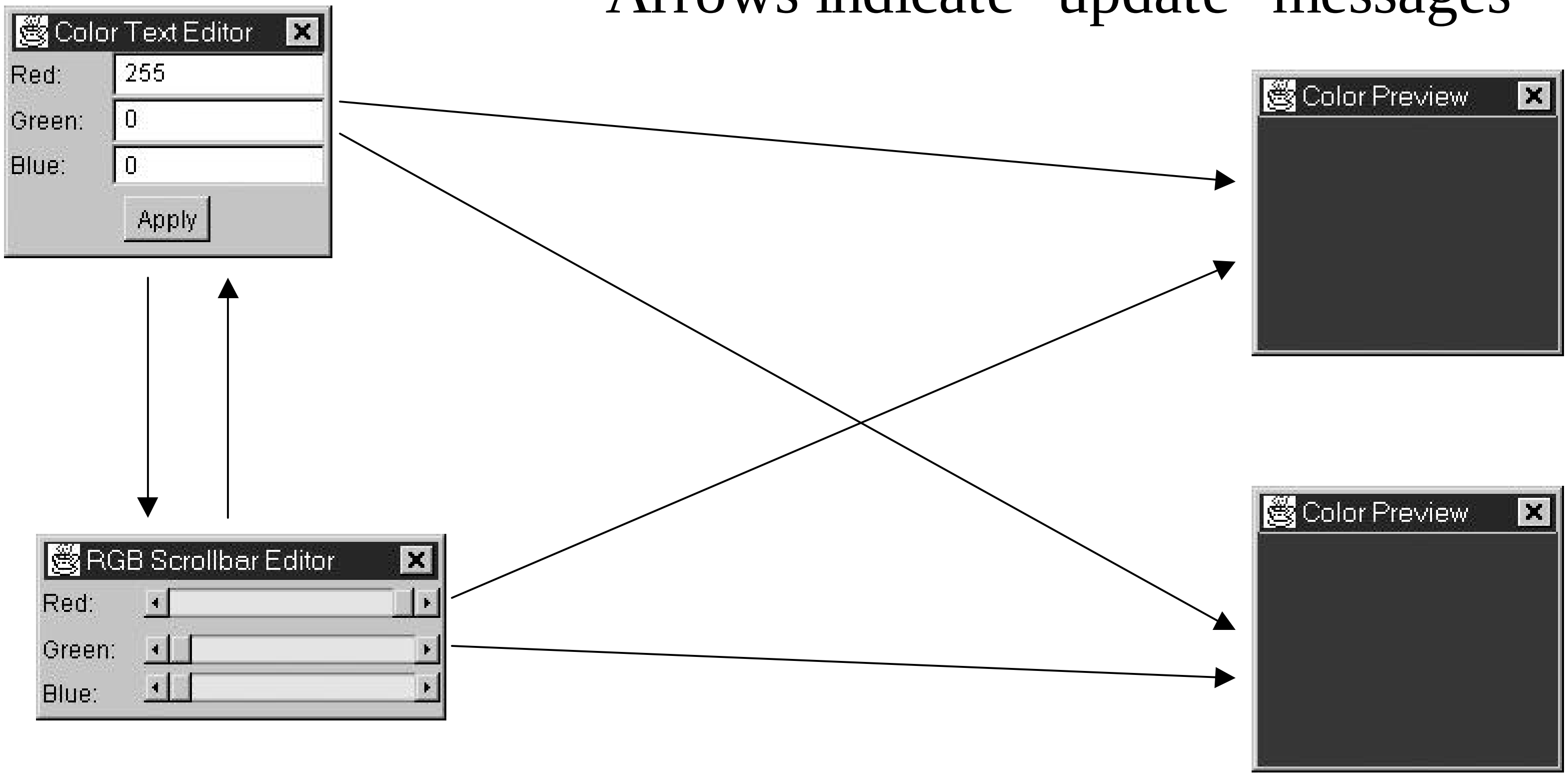
10 paths

10 objects

45 paths

How Not to Do It

- Arrows indicate "update" messages



Red: 255

Green: 0

Blue: 0

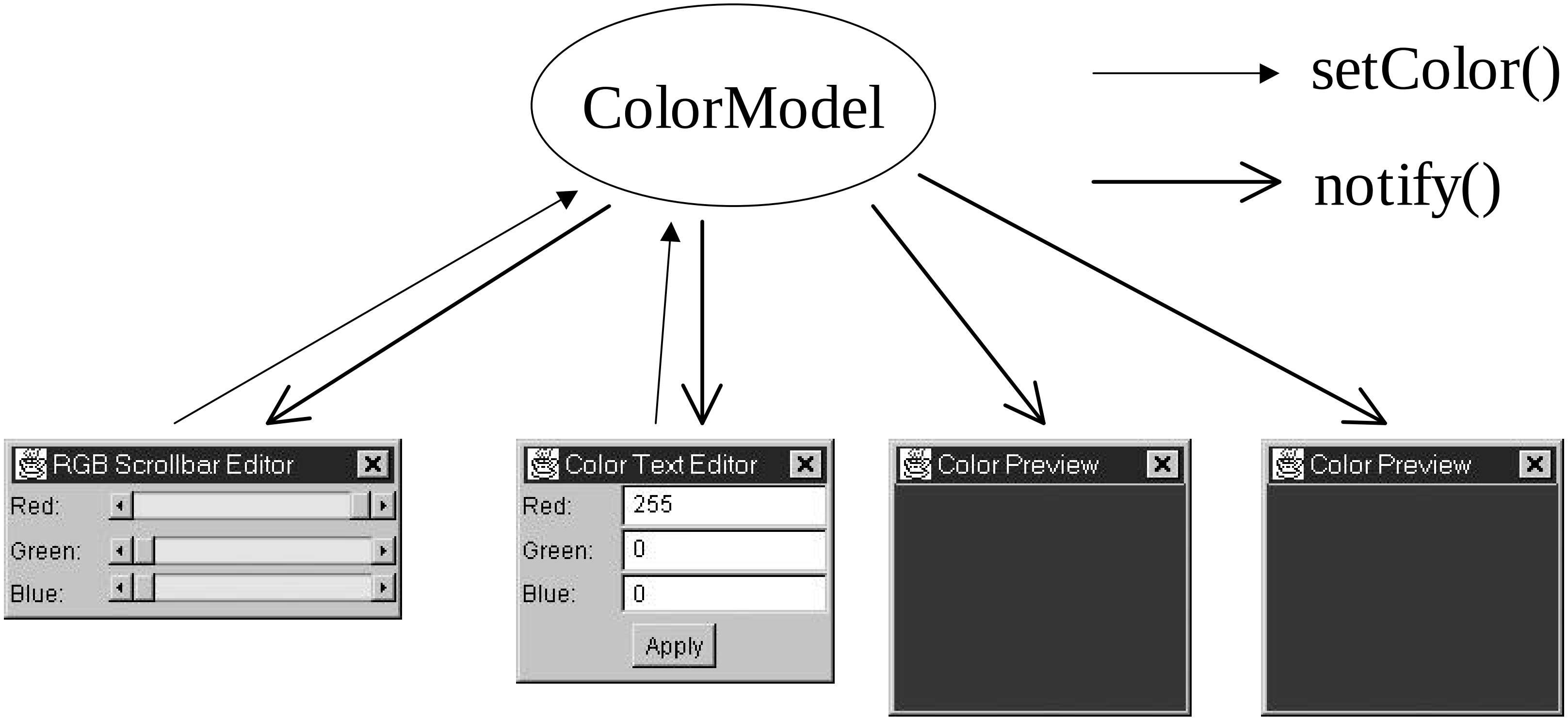
The Observable Pattern

- A notification mechanism which allows one object to notify many **observers** when a state change has occurred
- Known by many names
  - Doc-View, Model-View-Controller, Subject-Observer, Publish-Subscribe
  - the example implementation uses Model and View classes
    - the **Model** contains data (Subject)
    - one or more **Views** (Observers) display the data contained in the **Model**

Typical Uses

- Distributed computing
  - a server may notify many client machines when data changes
  - CORBA, RMI, Sockets, or other forms of communication may be utilized - **patterns are independent of implementation**
- Graphical User Interfaces
  - a spreadsheet notifies several different graphical charts whenever a cell is edited
  - Emacs and other text editors provide paned windows which display different views of the same document
  - CAD programs allow multiple 3D views of the same data

The "Observable" Approach



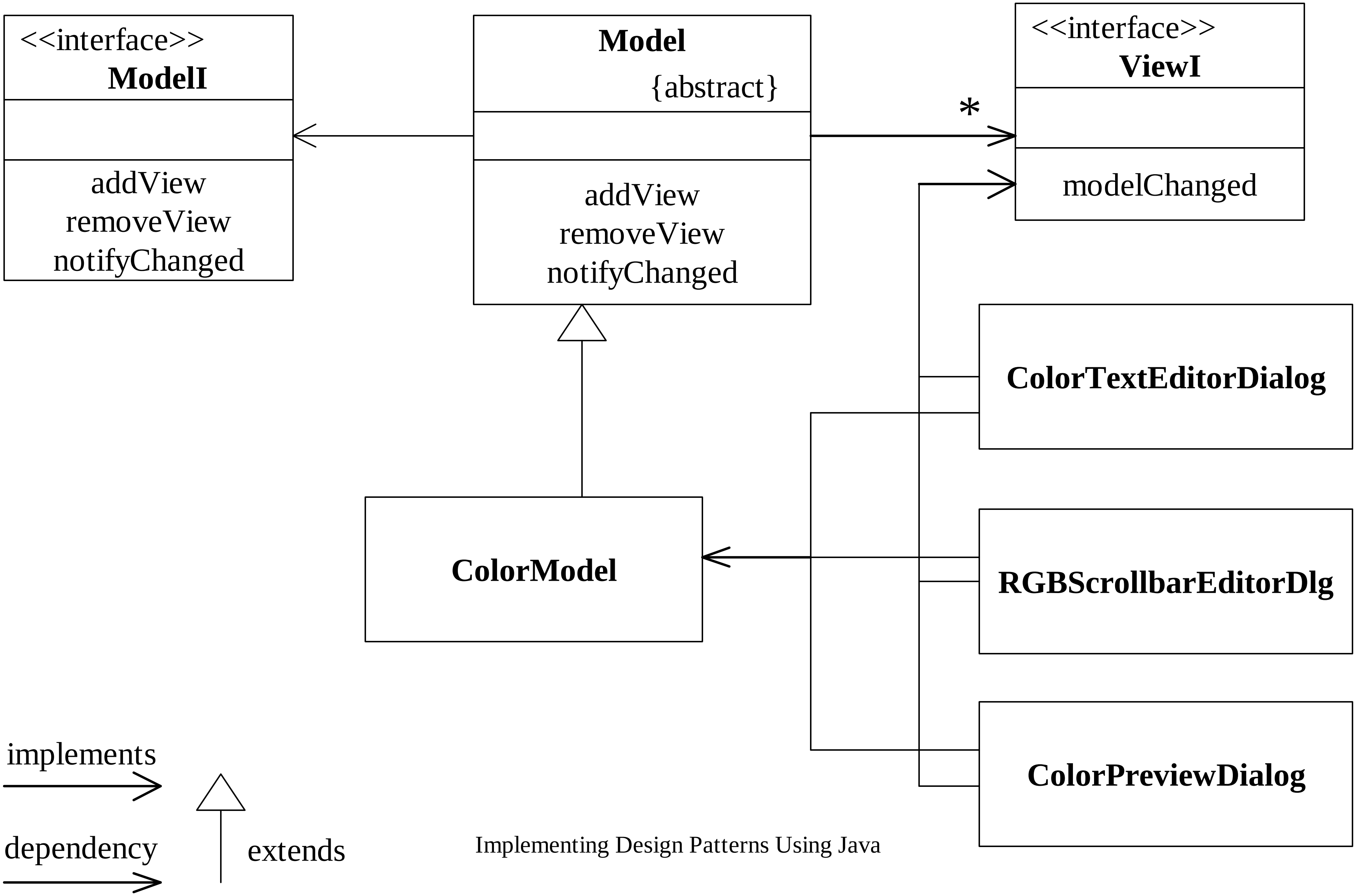
Red: 255
Green: 0
Blue: 0

- Views are not dependent upon one another
  - new views can be coded without impacting existing code
- Views are always displaying the correct data
- Ownership of the "Color" object is clearly defined in the ColorModel class

Implementing the Pattern

- Extend the Model class or implement the ModelI interface
  - business data is stored in your subclass
  - no GUI code is written here
  - no explicit connections to view subclasses
- Implement the ViewI interface
  - GUI panels, Frames, and Dialogs will do this
  - may hold a reference to one or more Model subclasses
  - provide a visual representation of business data
    - may be read-only or editable
    - must update display when notified by the Model

UML Object Model



<<interface>>
ModelI

addView
removeView
notifyChanged

Model
{abstract}

addView
removeView
notifyChanged

*

<<interface>>
ViewI

modelChanged

ColorTextEditorDialog

ColorModel

RGBScrollbarEditorDlg

ColorPreviewDialog

implements

dependency

extends

ModelI.java

```java
/**
 * Define a protocol for notifying dependent views whenever a data model
 * changes.  This is an implementation of the Observable pattern.
 * @author Eric M. Burke, Object Computing, Inc.
 * @version $Name$ $Revision$
 */
public interface ModelI {
    /**
     * @param v the view to register.
     */
    void addView(ViewI v);

    /**
     * @param v the view to un-register.
     */
    void removeView(ViewI v);

    /**
     * Notify all registered views that this model has changed.
     * @param obj optional data to pass to each view.  May be null.
     */
    void notifyChanged(Object obj);
}
```

ViewI.java

```
package com.ociweb.patterns.observer;

/**
 * A notification mechanism when a data model changes.
 * @author Eric M. Burke, Object Computing, Inc.
 * @version $Name$ $Revision$
 */
public interface ViewI {
    /**
     * The data model has changed.  Views implementing this interface
     * should update their displays.
     * @param model the data model which has changed.
     * @param obj optional data from the model, may be null.
     */
    void modelChanged(ModelI model, Object obj);
}
```

ColorModel.java

```
/**
 * A concrete type of Model.  Represents an AWT Color.
 */
public class ColorModel extends Model {
    private Color color;

    /**
     * @return a reference to a Color object.
     */
    public Color getColor() {
        return color;
    }

    /**
     * Set a new color and notify dependent views.
     * @param color the new Color.
     */
    public void setColor(Color color) {
        this.color = color;
        notifyChanged(null);
    }
}
```

ColorPreviewDialog.java

```java
// some details omitted to fit on slide
public class ColorPreviewDialog extends Dialog implements ViewI {
    ColorModel colorModel;

    public ColorPreviewDialog(Frame parent, ColorModel model) {
        super(parent, "Color Preview", false);
        this.colorModel = model;
        model.addView(this);
        setBackground(model.getColor());

        setSize(150,150);
    }

    // implement the ViewI interface
    public void modelChanged(ModelI model, Object obj) {
        setBackground(colorModel.getColor());
    }
}
```

JDK Observable

- The JDK includes a class called java.util.Observable
  - important methods include:
    - addObserver(Observer o)
    - deleteObserver(Observer o)
    - notifyObservers(Object arg)
    - setChanged(), clearChanged()
- Why use Model.java and ModelI.java instead of just using java.util.Observable?
  - Observable requires the extra step of setting the "changed" flag
  - Observable is a class, which **requires** inheritance
    - ModelI is an interface, which offers more flexibility

Advantages of the Observable Pattern

- Business data can be coded independently of specific GUI views
  - developers can work independently
  - text-only unit tests can be written to test data model before GUI is finished
- GUI views have no knowledge of each other
  - allows developers to work independently
  - allows new views to be added later without breaking existing views
  - reduces web of interconnected views

JDK Patterns

- Java is full of patterns - look at the source code

– Factory Method	java.util.Calendar
– Composite	java.awt.Container
– Iterator	java.util.Enumeration
– Observer	java.util.Observer
– Strategy	java.awt.LayoutManager

- The entire JavaBeans and AWT 1.1 event model is based upon a variation of the Observer pattern

Learning More

- Read the book "Design Patterns: Elements of Reusable Object-Oriented Software"

  Addison-Wesley, 1994 - Gamma, Helm, Johnson, Vlissides
- Study the JDK source code
- Practice
  - learning to recognize recurrent patterns and apply them to concrete designs takes experience

Time for Demo and Questions...

- Source code for demonstration application can be found at http://www.ociweb.com/javasig/
  - look for the Knowledge Base, under July, 1998