

Systems Design Issues

- ◆ So far, domain issues have driven design
- ◆ Today's software applications have multiple architecture issues:
 - Connection to a User Interface (UI)
 - Persistent storage mechanisms (DBs)
 - Communication & Coupling between subsystems (packages)
- ◆ Many other "patterns" have emerged in these areas to deal with the complexity

1

Three Tier Architecture

2

Multi-tiered Architectures

- ◆ Domain Logic is often Multi-tiered

3

Deployment

- ◆ Logical 3-tiered architectures can be deployed in various configurations
 - Presentation & application logic on a client machine, data storage on a server
 - Presentation on the client machine, application logic on an application server (servlets!), and storage on a database server
- ◆ Java motivates distributed architectures

4

Motivation for Multi-tiered

- ◆ Isolation of software logic into components that can be reused in other systems
- ◆ Distribution of tiers on different machines (or platforms) improves performance in Client-Server systems
- ◆ Assigning developers to tiers supports specialized expertise & ability to run parallel development efforts more easily

5

UML Packages

- ◆ A set of model elements, all within the same package namespace
 - includes classes, use cases, collaboration diagrams, other packages, etc.
- ◆ An entire system is considered to be within the scope of a single top-level package, the System package

6

UML Package Notation

Packages can appear in Logical views of the system, such as in layer diagrams, at right.

7

Packages & Dependencies

8

Identifying Packages

- Classes that collaborate to provide a service naturally tend to have high coupling
 - This is an indicator that they may belong together in the same package
 - If coupling is very strong, this can be viewed as high cohesion between elements in the same package!
- Conversely, low coupling in collaboration diagrams indicates elements belong to different packages

9

Layers & Partitions

- The layers of an architecture represent vertical tiers
- Partitions represent a horizontal division of relatively independent features of a layer

10

Visibility Between Packages

- Presentation packages have visibility to many of the domain classes
- Domain & Presentation packages have visibility to one class in a Service package
 - usually a Facade class pattern
- No packages have direct visibility to the Presentation layer
 - Only indirect communication through an Observer pattern

11

Visibility Between Packages

12

Model-View Separation

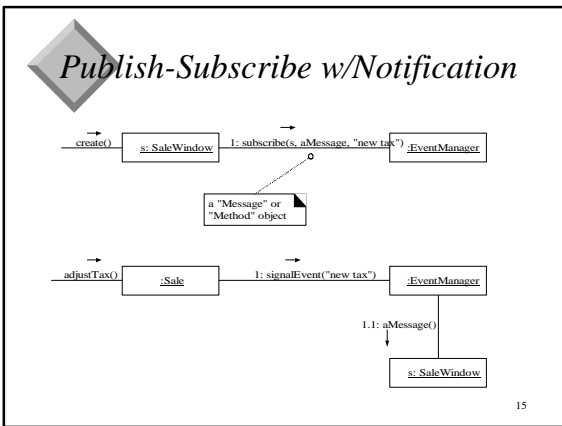
- ◆ *Model* is a term for domain layer objects
- ◆ *View* is a term for presentation layer objects
- ◆ *Separation* means that the domain and the presentation layers have no direct knowledge of one another!
- ◆ Q: So how does it work then?
- ◆ Ans: Indirect communication via brokers acting as intermediaries

13

Observer Pattern

- ◆ Also known as *Publish-Subscribe* pattern
- ◆ *Problem*: An event occurs within a *publisher* and other objects are dependent on this event (*subscribers* to the event). However, the publisher has no direct references to the subscribers.
- ◆ *Solution*: An *EventManager* class maintains mappings between publishers & subscribers. An event is published by signaling the *EventManager*, who notifies the subscribers by sending them a parameterized msg (i.e. *callback*)

14



Java's Event Delegation Model

- ◆ Implements the Observer pattern within the AWT (Abstract Windowing Toolkit)
- ◆ Subscribers implement an *EventListener* interface and *register* with event source objects (ie. publishers) for events of interest
- ◆ Source objects maintain a list of registered listeners and notify them when the event of interest occurs

16

Java's Event Delegation Model

- ◆ Notice there is no single *EventManager*
 - There used to be in JDK 1.0 event model
 - It was a performance "bottleneck" because:
 - It handled *all* events
 - It searched within containing objects for *event handlers* (a very time consuming process)
- ◆ JDK 1.1 replaced it with the new Event Delegation Model

17

Java 1.1 Event Handling Example

```

import java.awt.*;
import java.awt.event.*;

public class EventExample
{
    public static void main(String args[])
    {
        Button b;
        ActionListener al = new MyActionListener();
        Frame f = new Frame("Hello in 3 Langs.");
        f.add(b = new Button("Hola"), "North");
        b.setActionCommand("Hello");
        b.addActionListener(al);
        f.add(b = new Button("Aloha"), "Center");
        b.addActionListener(al);
        f.add(b = new Button("Adios"), "South");
        b.setActionCommand("Quit");
        b.addActionListener(al);
        f.pack(); f.show();
    }
}
    
```

18

Java 1.1 Event Handling Example

```

class MyActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String s = e.getActionCommand();
        if (s.equals("Quit"))
            System.exit(0); // normal exit
        else if (s.equals("Hello"))
            System.out.println("Bon Jour");
        else
            System.out.println(s + " selected");
    }
}
    
```

19

Java Inner Classes

- ◆ Starting with JDK 1.1, it is possible to define classes right at the scoping level at which you'd like to use them (i.e. everything in scope!)
- ◆ They are typically unnamed (anonymous), and can be of any visible class type
- ◆ The inner class has access to variables & objects in the outer scope
- ◆ Conveniently used to over-ride methods in existing classes, right in the place they are needed

20

Java Inner Class Event Handling

```

import java.awt.*;
import java.awt.event.*;

public class InnerClassHandler
{
    public static void main(String args[])
    {
        Frame f = new Frame("Hello in 1 Lang.");
        Button b = new Button("Hello");
        b.addActionListener
        (
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    System.out.println(e.getActionCommand());
                }
            }
        );
        f.add(b); f.pack(); f.show();
    }
}
    
```

An anonymous inner class!

21

Java AWT Adapter Classes

- ◆ An Adaptor class already handles the events for which it was designed by implementing the appropriate *EventListener* interface – i.e. *ActionListener*
- ◆ So, use inheritance to create a subclass of the adaptor and over-ride only the methods of interest – *MouseAdapter* already handles mouse events

22

Java Adapter Example

```

import java.awt.*;
import java.awt.event.*;

public class Draw extends java.applet.Applet
{
    public void init()
    {
        addMouseListener(new DrawAdapter(getGraphics()));
    }
}

class DrawAdapter extends MouseAdapter
{
    Graphics g;
    int savedX, savedY;
    DrawAdapter(Graphics g) { this.g = g; }
    public void mousePressed(MouseEvent e)
    {
        savedX = e.getX(); savedY = e.getY();
    }
    public void mouseReleased(MouseEvent e)
    {
        g.drawRect(savedX, savedY, e.getX() - savedX,
            e.getY() - savedY);
    }
}
    
```

23

Java Adapter Inner Class Example

```

import java.awt.*;
import java.awt.event.*;

public class Draw extends java.applet.Applet
{
    public void init()
    {
        addMouseListener
        (
            new MouseAdapter()
            {
                int savedX, savedY;
                public void mousePressed(MouseEvent e)
                {
                    savedX = e.getX();
                    savedY = e.getY();
                }
                public void mouseReleased(MouseEvent e)
                {
                    Graphics g = Draw.this.getGraphics();
                    g.drawRect(savedX, savedY, e.getX() -
                        savedX, e.getY() - savedY);
                }
            }
        );
    }
}
    
```

Inner Adapter Class

24

Advantages of Observer Patterns

- ◆ No direct coupling between senders and receivers
- ◆ A single event can be broadcast (multicast) to any number of subscribers
- ◆ The reaction to an event can be generalized in callback objects
- ◆ Each callback object could execute in its own thread (performance enhancement)

25

Application Coordinators

- ◆ Mediate between the Presentation layer and the Domain layer
- ◆ Deliver system event objects
- ◆ Open windows to display information from domain objects
- ◆ Manage transactions (commit, rollback)
- ◆ MFC used a Document View coordinator

26

