ELEC 372
Engineering Software Structures

Week 9: Software Architecture

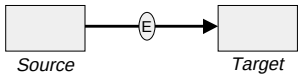
Lecture 1

Examples [Shaw&Garlan]

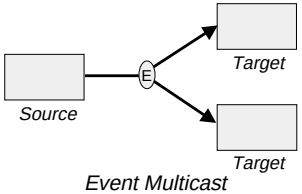
- Pipes and Filters
- Layers
- Event-Based
- Data-Centered
- Interpreter
- PAC

Event-Driven Architecture Style

- Source sends message *(event)* to target
- Target can be 0 or more components attached to event connector
- Source has no direct knowledge of target, only has knowledge of connector
  - Therefore targets can change statically or dynamically



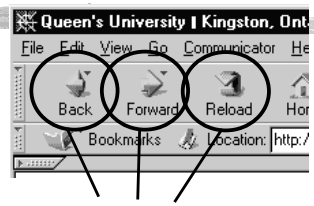
Event Unicast



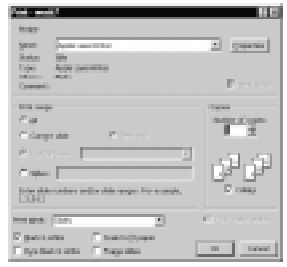
Event Multicast

Example: Graphical User Interface (GUI) Components

- Graphical user interfaces are composed of a set of primitive components (or *widgets*)
  - Buttons, scroll bars, menus, dialogue boxes, text entry fields
- These widgets should be customizable and reusable in any context, not hand-coded for every application
- Therefore, should not have knowledge of the architecture in which they are placed



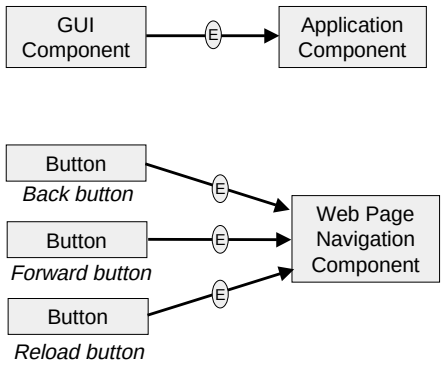
Button



Dialogue Box

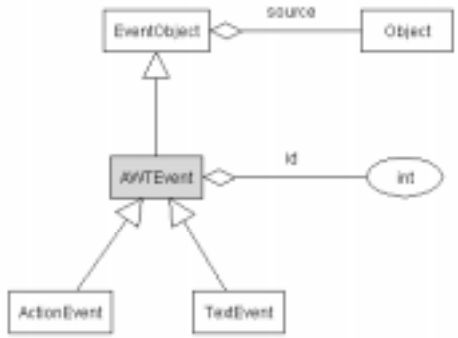
GUI Components and Events

- GUI components issue *(raise)* events when they are invoked
  - Buttons issue *buttonClicked* events
  - Menus issue *menuSelection* events
  - Scroll bars issue scroll *events*
  - Text fields issue *text changed* events
  - *etc...*



Java Implementation of Events

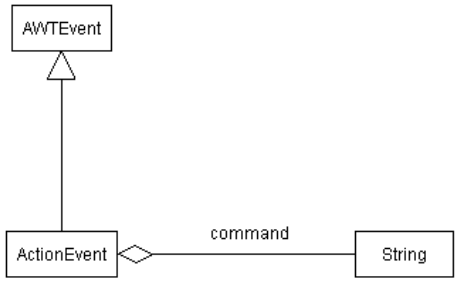
- All events subclass *EventObject*
  - Only field is source of event (e.g., the button that raised the event)
- All GUI events subclass AWTEvent
  - Only field is a unique integer identifier
- Eg
  - ActionEvent for button clicks, menu selections, etc
  - Text Event for entry in text field, etc



See also http://www.javasoft.com/j2se/1.3/docs/api/index.html

Action Event Class

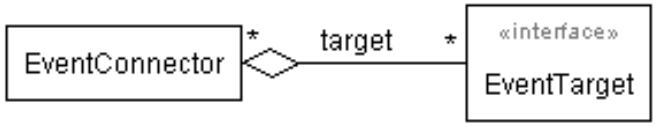
- E.g., button, menu events
- *command* field represents string form of event that occurred



Implementing Event Connector

- Method seen by source
  - void raiseEvent (EventObject e)
- Method seen by target
  - void connectAsTarget (EventTarget targetComponent)
- Method seen by connector in target
  - void performEvent (EventObject e)

Implementing Event Connector




```
// Target implements this method. It is called by the
// event connector whenever a new event is fired.
public interface EventTarget {
        public void performEvent (EventObject e);
}
```

```
// Code sketch for event connector.
public class EventConnector {
    // There are zero or more targets represented as a vector
    private Vector targets = new Vector();
    // When the source raises an event, the "performEvent"
    // method of each target is invoked
    public void raiseEvent (EventObject e) {
        for (int i=0; i<targets.size(); i++) {
            EventTarget t = (EventTarget) targets.elementAt (i);
            t.performEvent (e);
        }
    }

    // Each target must register itself as a target of
    // this connector. Targets are simply added to the
    // "targets" vector.
    public void connectAsTarget (EventTarget targetComponent) {
        targets.add (targetComponent);
    }
}
```

Event-Based Architecture: Domain

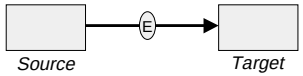
- This style solves the problem of connecting components where
  - There may be more than one target so that multicast is useful
  - The source component is built to be reused in many contexts

Properties

- Modifiability
- Performance

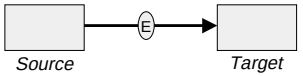
Modifiability

- Source has no reference to target
- Therefore changes in
  - target
  - number of targets
- do not impact source



Performance

- Communication between source and target requires indirection (through connector)
- May have minor impact on performance
- Advanced compilers (e.g. Java hotspot) compile away this indirection

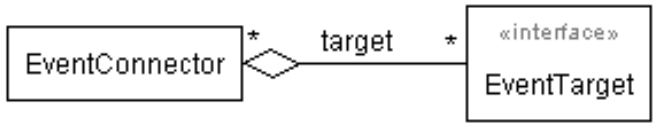


ELEC 372
Engineering Software Structures

Week 9: Software Architecture

Lecture 1

Implementing Event Connector

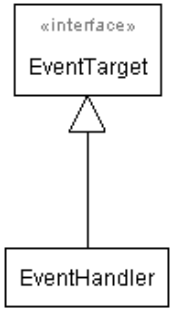



```
// Target implements this method. It is called by the
// event connector whenever a new event is fired.
public interface EventTarget {
        public void performEvent (EventObject e);
}
```

Example Implementation of the EventTarget Interface

```
public interface EventTarget {
    public void performEvent (EventObject e);
}

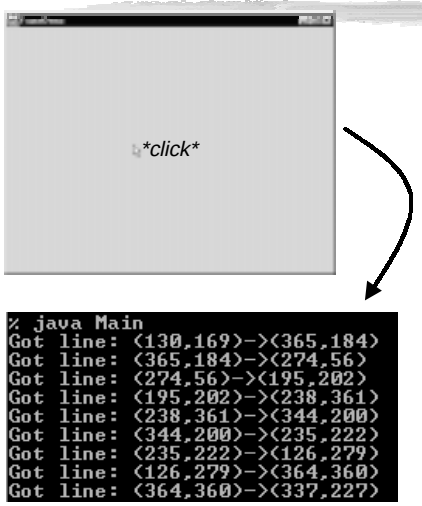
// This class implements the EventTarget interface. This class
// can contain any fields and methods, but must include an
// an implementation of the performEvent method
public class HandleEvent implements EventTarget {
    ...
    public void performEvent (EventObject e) {
        System.out.println ("Got event " + e);
    }
    ...
}
```



Interfaces

- Classes may implement any number of interfaces
  - but recall that in Java, classes may only inherit from one other class

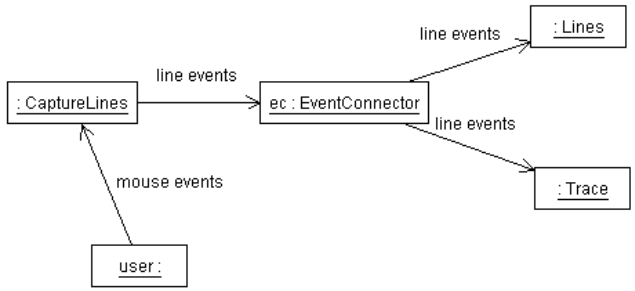
Example Application using Event-Driven Architecture

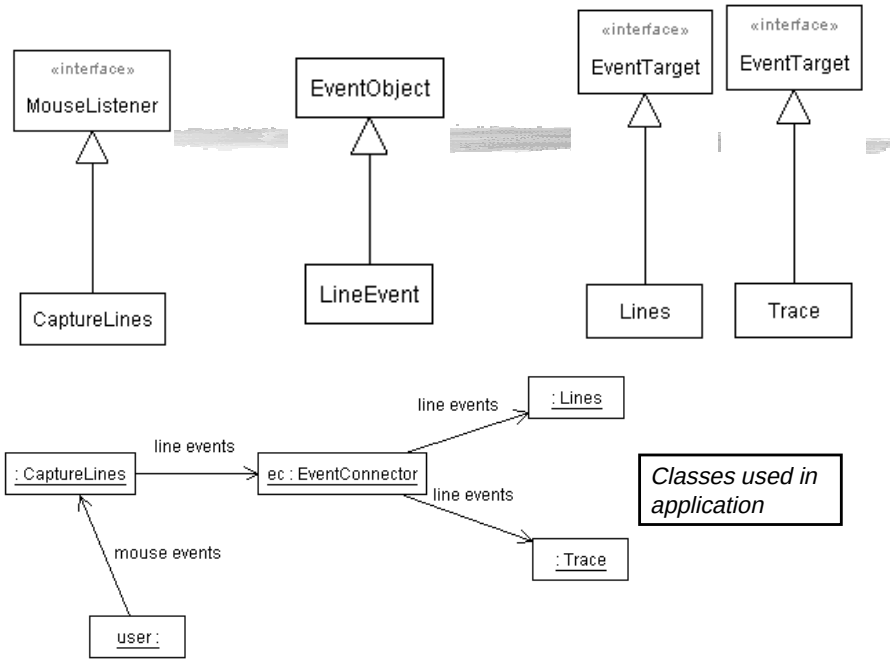


- First step in implementing a drawing program
  - User clicks on canvas
  - Successive pairs generate lines
  - Lines saved in list
  - Trace shows lines that were generated

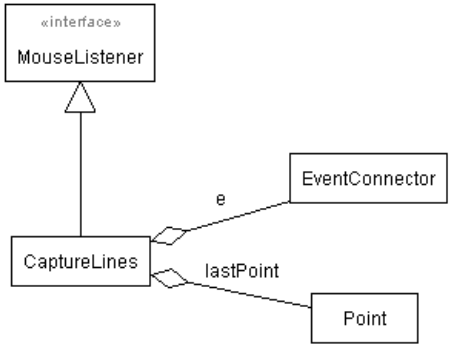
An example

- *CaptureLines* component records mouse clicks on a canvas
  - each pair of clicks generates a line
  - LineEvent events sent to an event connector
- *Lines* component collects these lines into a vector
- *Trace* component prints out line events as they occur
  - useful for debugging



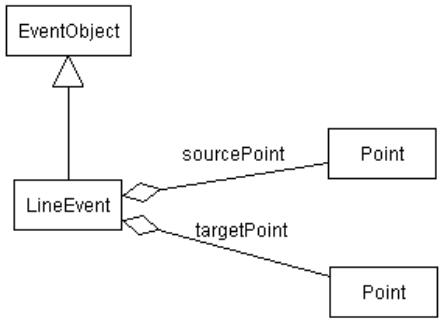


CaptureLines



- Captures mouse click events from user
- Maintains last point clicked
- Creates *LineEvent* containing last point and new point

LineEvent



- The line runs from *sourcePoint* to *targetPoint*
- Each point is a coordinate, containing an (x,y) pair

```
class CaptureLines implements MouseListener {
    private EventConnector e;
    private Point lastPoint = null;
    JFrame frame;

    public void mouseClicked(MouseEvent mevt) {
      Point newPoint = mevt.getPoint();
      if (lastPoint == null) {
          lastPoint = newPoint;
      } else {
          LineEvent levt =
               new LineEvent (
                    this, lastPoint, newPoint);
          lastPoint = newPoint;

          e.raiseEvent (levt);
      }
    }
    ...
```

```
class CaptureLines implements MouseListener {
    ...
    public CaptureLines (EventConnector e) {
      this.e = e;
      JFrame frame = new JFrame("FrameDemo");
      frame.addMouseListener (this);
      frame.setSize (500, 400);
      frame.setVisible (true);
    }
}
```

```
//  Records lines into a list of lines

public class Lines implements EventTarget {
    // The lines that have been added
    private Vector lines = new Vector();

    // Add a new line into the list of lines
    public void performEvent (EventObject e) {
      lines.add (e);
    }

    // Return the list of lines
    public Vector getLines() {
      return lines;
    }
}
```

```
import java.util.EventObject;
import java.awt.Point;

//  Simply traces events as they occur.

public class Trace implements EventTarget {
    public void performEvent (EventObject e) {
      LineEvent nextLine = (LineEvent) e;
      Point p1 = nextLine.getSourcePoint();
      Point p2 = nextLine.getTargetPoint();

      System.out.println ("Got line: "
          + "(" + Math.round(p1.getX())
          + "," + Math.round (p1.getY()) + ")"
          + "->"
          + "(" + Math.round(p2.getX()) + ","
          + Math.round(p2.getY()) + ")" );
    }
}
```

```
import java.util.EventObject;
import java.awt.Point;
public class LineEvent extends EventObject {
    private Point sourcePoint;
    private Point targetPoint;
    public Point getSourcePoint () {
      return sourcePoint;
    }
    public Point getTargetPoint () {
      return targetPoint;
    }
    public LineEvent (Object source,
        Point sourcePoint, Point targetPoint) {
      super(source);
      this.sourcePoint = sourcePoint;
      this.targetPoint = targetPoint;
    }
}
```

```
import java.util.EventObject;

// Target implements this method. It is called by the
// event connector whenever a new event is fired.

public interface EventTarget {
    public void performEvent (EventObject e);
}
```

```
import java.util.Vector;
import java.util.EventObject;

public class EventConnector {
    private Vector targets = new Vector();

    public void raiseEvent (EventObject e) {
      for (int i=0; i<targets.size(); i++) {
        EventTarget t =
          (EventTarget) targets.elementAt (i);
        t.performEvent (e);
      }
    }

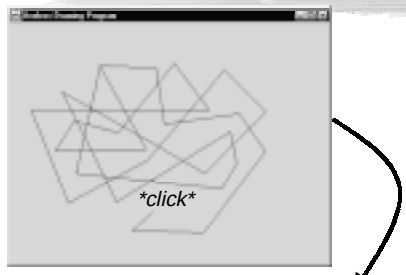
    public void connectAsTarget (
             EventTarget targetComponent) {
      targets.add (targetComponent);
    }
}
```

ELEC 372
Engineering Software Structures

Week 9: Software Architecture

Lecture 3

Example Application using Event-Driven Architecture

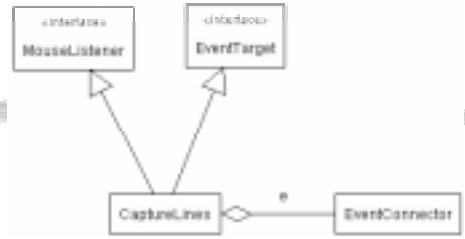


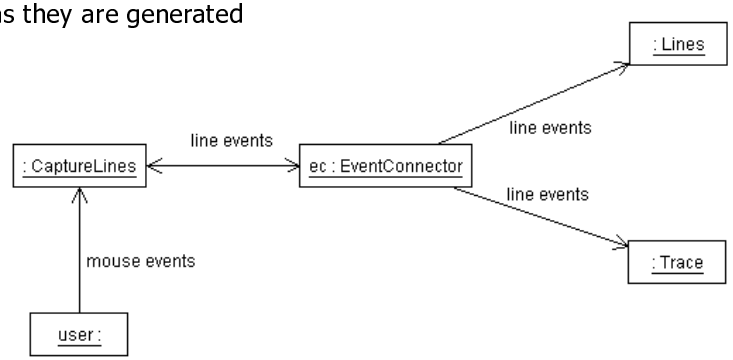

```
% java Main
Got line: (130,169)->(365,184)
Got line: (365,184)->(274,56)
Got line: (274,56)->(195,202)
Got line: (195,202)->(238,361)
Got line: (238,361)->(344,200)
Got line: (344,200)->(235,222)
Got line: (235,222)->(126,279)
Got line: (126,279)->(364,360)
Got line: (364,360)->(337,227)
```

- Second step in implementing a drawing program
  - User clicks on canvas
  - Successive pairs generate lines
  - Lines saved in list
  - Trace shows lines that were generated
  - Lines drawn on canvas

An example

- *CaptureLines* component becomes a target of the event connector
- *performEvent* method added to CaptureLines to draw lines as they are generated






```
class CaptureLines implements MouseListener, EventTarget {
    private EventConnector e;
    ...
    public void performEvent (EventObject e) {
        LineEvent nextLine = (LineEvent) e;

        Graphics g = frame.getGraphics();

        // Extract the points of the line from the event
         Point p1 = nextLine.getSourcePoint();
         Point p2 = nextLine.getTargetPoint();

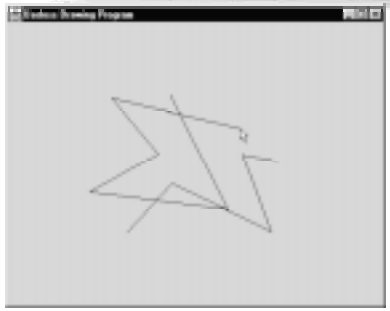
        int x1=(int)Math.round (p1.getX());
        int y1=(int)Math.round (p1.getY());
        int x2=(int)Math.round (p2.getX());
        int y2=(int)Math.round (p2.getY());

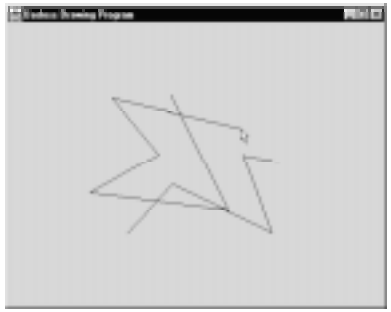
        // Draw the line in the window
        g.drawLine (x1, y1, x2, y2);
    }
    ...
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create the connector
        EventConnector ec = new EventConnector();

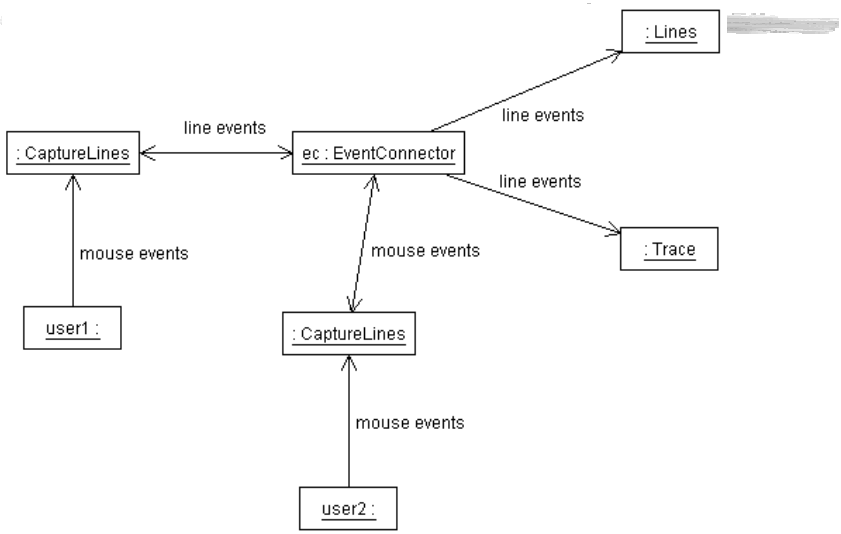
        // Create the line event generator
        CaptureLines captureComponent = new CaptureLines (ec);
        ...
        // The third target is the line generator itself
        ec.connectAsTarget (captureComponent);
    }
}
```

Modification: Make this program multiuser by adding a second window





Architectural Modification to Support Second Window




```
public class Main {
    public static void main(String[] args) {
        // Create the connector
        EventConnector ec = new EventConnector();

        // Create the line event generator
        CaptureLines captureComponent = new CaptureLines (ec);
        ...
        // The third target is the line generator itself
        ec.connectAsTarget (captureComponent);

        // And let's create another window - we can draw lines there
        // as well!
        CaptureLines anotherCaptureComponent = new CaptureLines (ec);
        ec.connectAsTarget (anotherCaptureComponent);
    }
}
```

Summary

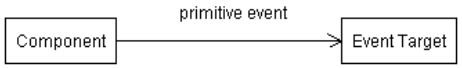
- Event-based architecture style has good modifiability properties
- We saw
  - Addition of drawing functionality to program
    - Required connection of *CaptureLines* as **target** as well as **source** of event connector
    - Required addition of **performEvent** in *CaptureLines* to actually draw the line
  - Addition of another window
    - Required creation of new instance of *CaptureLines*
    - Required connection of this new instance as source and target of event connector

Example: Event-Based Programming in the AWT

- Java has two graphics libraries
  - *Abstract Windowing Toolkit* (AWT)
  - Since Java 2 platform, *Swing* builds on AWT
- Both are event-based
- Do not have explicit event connectors
  - Event connectors built into source components
- Primitive events generated by runtime system
  - Mouse clicks, mouse motion, key clicks

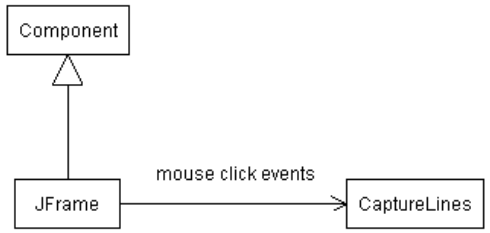
Subscribing to Mouse Events

- All GUI components are subclasses of *java.awt.Component*
- Mouse/keyboard events sent from components to all interested target components



Mouse Events in Our Application

- Windows are implemented using *javax.swing.JFrame* component
- *CaptureLines* component is target of these events
- In order to receive these events, *CaptureLines* must register with the *JFrame* component



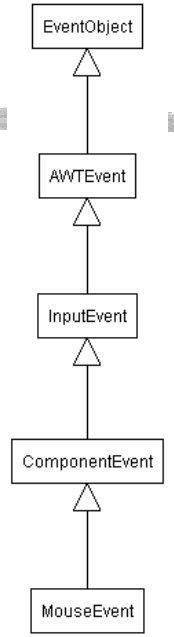
CaptureLines registers itself as a Target for MouseEvents generated by *JFrame*

```
class CaptureLines implements MouseListener, EventTarget {
    ...
    JFrame frame;

    public CaptureLines (EventConnector e) {
        ...
        // Create a window in which a user can click
        frame = new JFrame("Useless Drawing Program");
        ...
        // Adds this object as a target for mouse events
        // generated in this window. This method requires
        // an object of type MouseListener.
        frame.addMouseListener (this);
        ...
    }
}
```

Mouse Events

- Provides information on
  - Location where mouse was clicked
  - Number of clicks
    - Distinguish between double and single-click



MouseListener Interface

```
// Generates a mouse click event
void mouseClicked(MouseEvent e)

// Generates an event when the mouse moves over the component
void mouseEntered(MouseEvent e)

// Generates an event when the mouse moves off the component
void mouseExited(MouseEvent e)

// Generates an event when a mouse button is depressed
void mousePressed(MouseEvent e)

// Generates an event when a mouse button is released
void mouseReleased(MouseEvent e)
```

```
class CaptureLines implements MouseListener, EventTarget {
    ...
    public void mouseClicked(MouseEvent mevt) {
        Point newPoint = mevt.getPoint();
        if (lastPoint == null) {
            // This is the first click - record this point
            lastPoint = newPoint;
        } else {
            // Create a line event recording a line from the last point
            // to this new point, then record the new point.
            LineEvent levt = new LineEvent (this, lastPoint, newPoint);
            lastPoint = newPoint;

            // Fire the event
            e.raiseEvent (levt);
        }
    }

    // These are other mouse events that may arise; we ignore these
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e)  {}
    ...
}
```