Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Design case studies

Introduction to concurrency and GUIs

Josh Bloch **Charlie Garrod** Darya Melicher

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Reading due today: UML and Patterns 26.1 and 26.4
- Homework 4b due Thursday, October 18th
  - Homework 4a feedback coming tomorrow or Thursday
- PA voter registration deadline: today!

Key concepts from last Thursday

- Class invariants must be maintained
  - Make defensive copies where required
- Immutable classes have many advantages
- Testing is critical to software quality
  - Good tests have high power-to-weight ratio

Key concepts from last week's recitation

- Discovering design patterns
- Observer design pattern

Observer pattern (a.k.a. publish/subscribe)

- Problem: Must notify other objects (observers) without becoming dependent on the objects receiving the notification
- Solution: Define a small interface to define how observers receive a notification, and only depend on the interface
- Consequences:
  - Loose coupling between observers and the source of the notifications
  - Notifications can cause a cascade effect

See edu.cmu.cs.cs214.rec06.alarmclock.AlarmListener…

Learning goals for today

- Understand basic Java techniques and challenges for concurrent programming
- Understand thread model in Swing
- Understand the design challenges and common solutions for Graphical User Interfaces (GUIs)
- Understand event-based programming
- Understand and recognize the design patterns used and how those design patterns achieve design goals.
  - Observer pattern

Today

- The observer pattern
- Introduction to concurrency
- Introduction to GUIs

A *thread* is a thread of execution

- Multiple threads in the same program concurrently
- Threads share the same memory address space

Threads vs. processes

- Threads are lightweight; processes are heavyweight
- Threads share address space; processes don't
- Threads require synchronization; processes don't
- It's unsafe to kill threads; safe to kill processes

Reasons to use threads

- Performance needed for blocking activities
- Performance on multi-core processors
- Natural concurrency in the real-world
- Existing multi-threaded, managed run-time environments

A simple threads example

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

A simple threads example

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    Runnable greeter = () -> System.out.println("Hi mom!");
    for (int i = 0; i < n; i++) {
        new Thread(greeter).start();
    }
}
```

A simple threads example

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    for (int i = 0; i < n; i++) {
        new Thread(() -> System.out.println("Hi mom!")).start();
    }
}
```

Aside: Anonymous inner class scope in Java

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    for (int i = 0; i < n; i++) {
        new Thread(() -> System.out.println("T" + i)).start();
    }
}
```

won't compile
because i mutates

Aside: Anonymous inner class scope in Java

```
public interface Runnable {  // java.lang.Runnable
    public void run();
}

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);  // Number of threads;

    for (int i = 0; i < n; i++) {
        int j = i;  // j unchanging within each loop
        new Thread(() -> System.out.println("T" + j)).start();
    }
}
```

j is *effectively final*

Aside?: Design with inner class scope in Java

Threads for performance

- Naïve multi-threading on a simple parallel computation

Number of threads	Seconds to run
1	22.0
2	13.5
3	11.7
4	10.8

Shared mutable state requires synchronization

- Three basic choices:
  1. Don't mutate: share only immutable state
  2. Don't share: isolate mutable state in individual threads
  3. If you must share mutable state: synchronize properly

The challenge of synchronization

- Not enough synchronization: *safety failure*
  - Incorrect computation
- Too much synchronization: *liveness failure*
  - No computation at all

Today

- The observer pattern
- Introduction to concurrency
- Introduction to GUIs

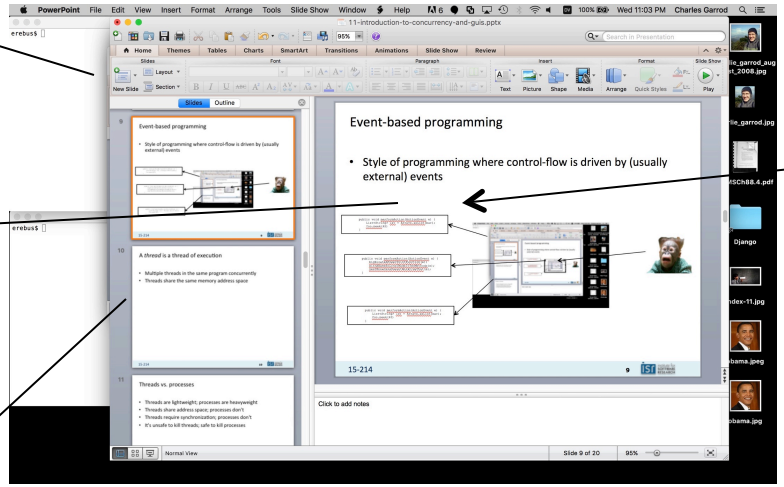
Event-based programming

- Style of programming where control-flow is driven by (usually external) events

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

```
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```

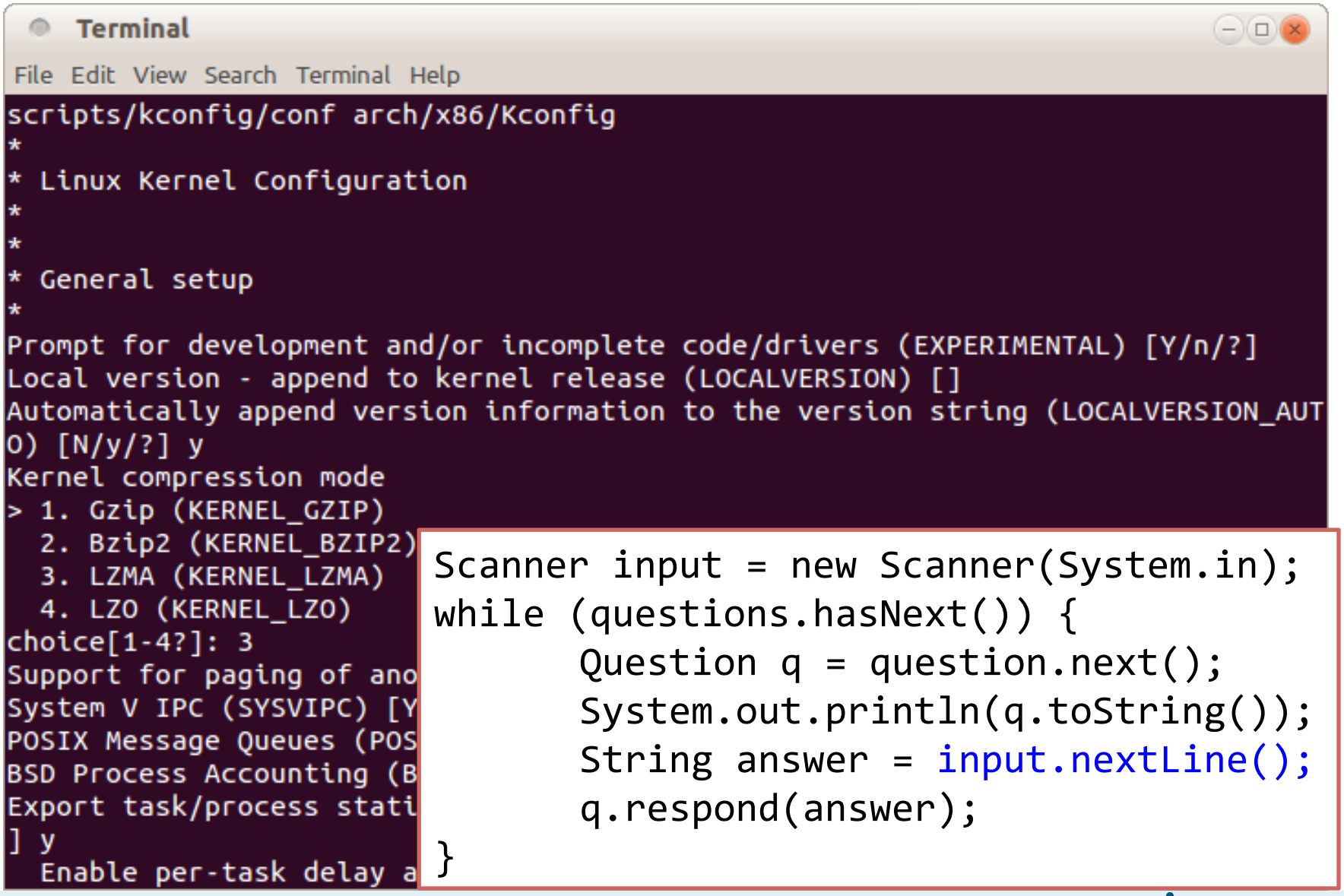


Examples of events in GUIs

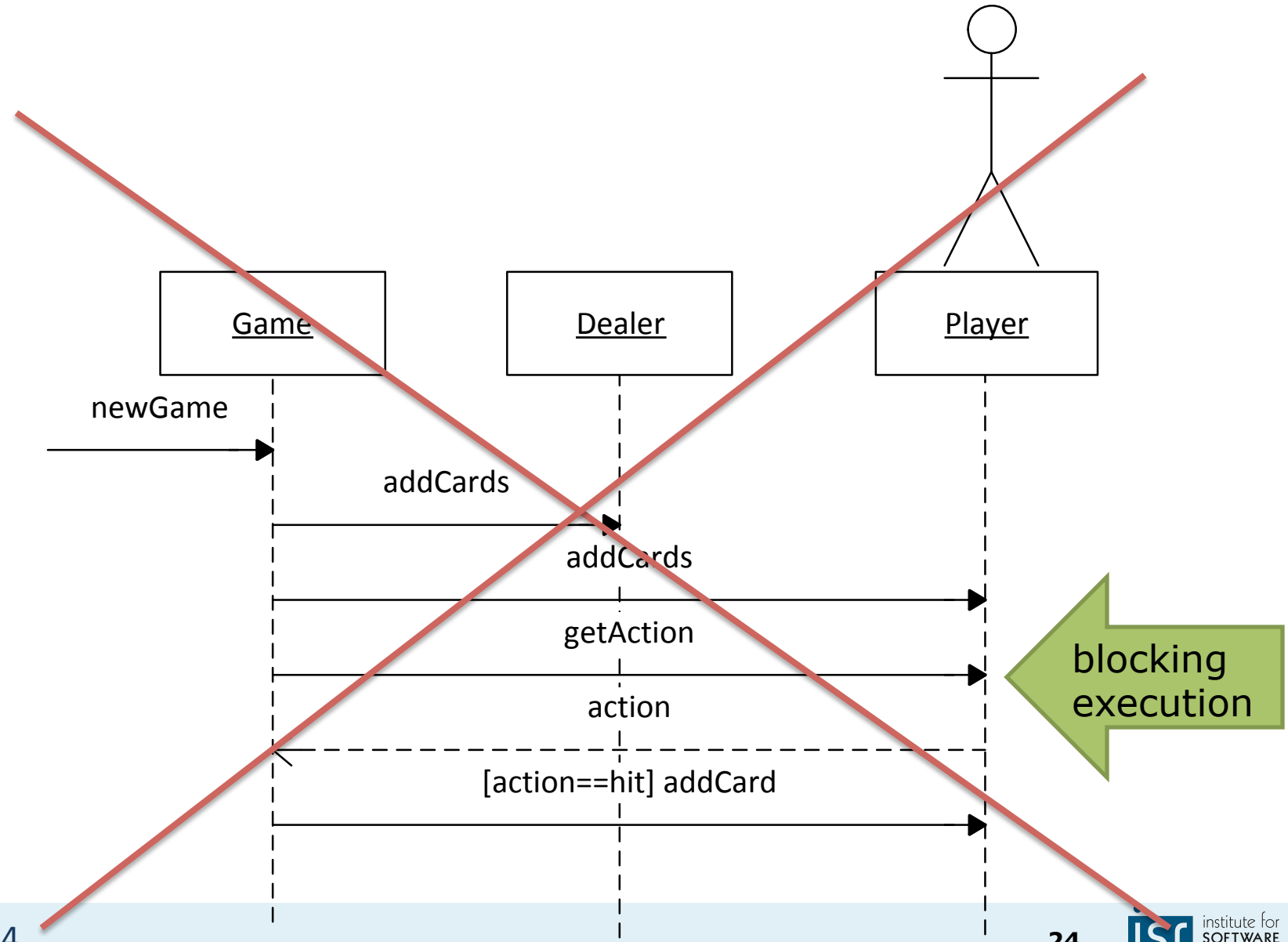
- User clicks a button, presses a key
- User selects an item from a list, an item from a menu
- Mouse hovers over a widget, focus changes
- Scrolling, mouse wheel turned
- Resizing a window, hiding a window
- Drag and drop

- A packet arrives from a web service, connection drops, …
- System shutdown, …

Blocking interaction with command-line interfaces

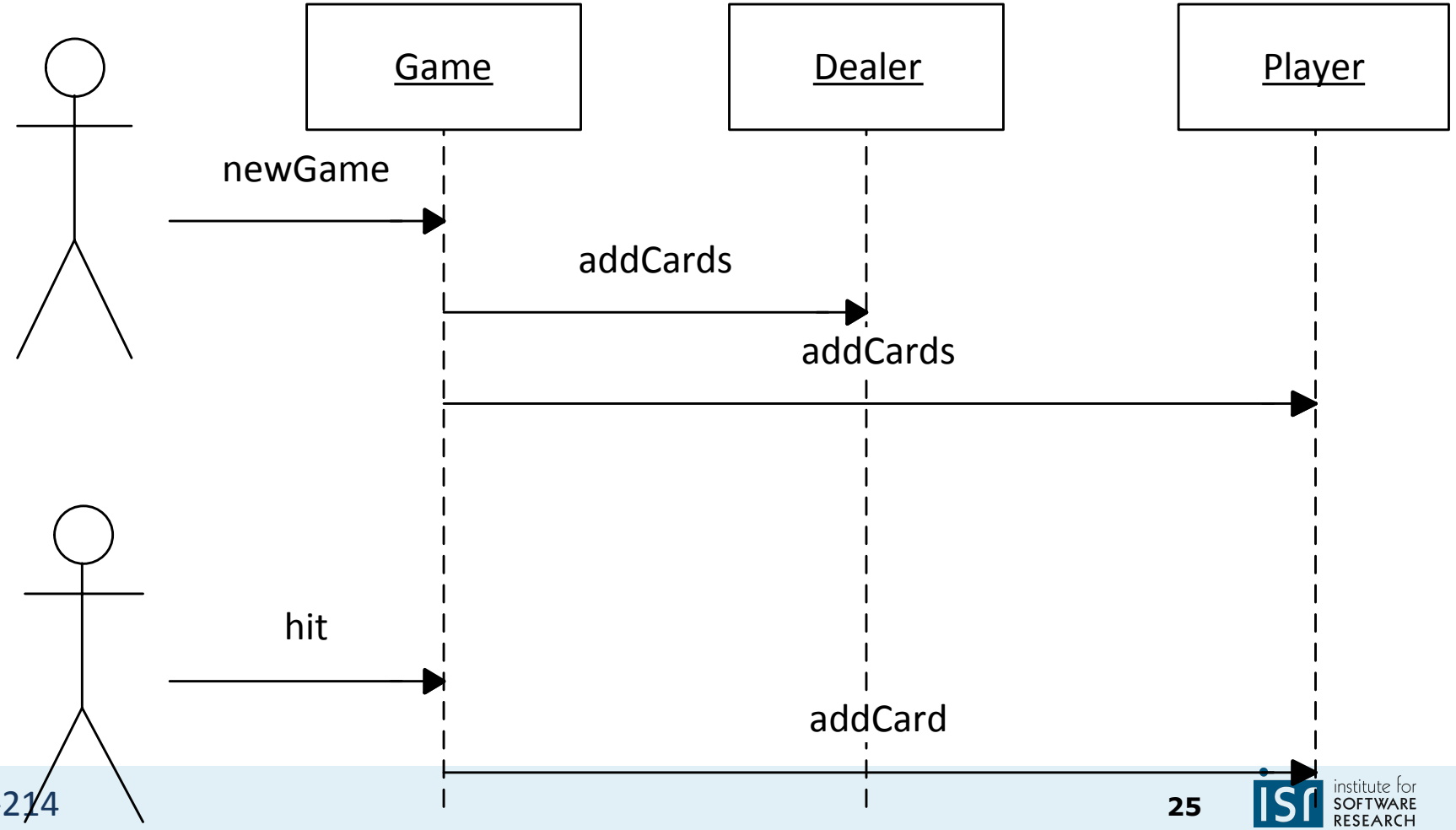


Blocking interactions with users



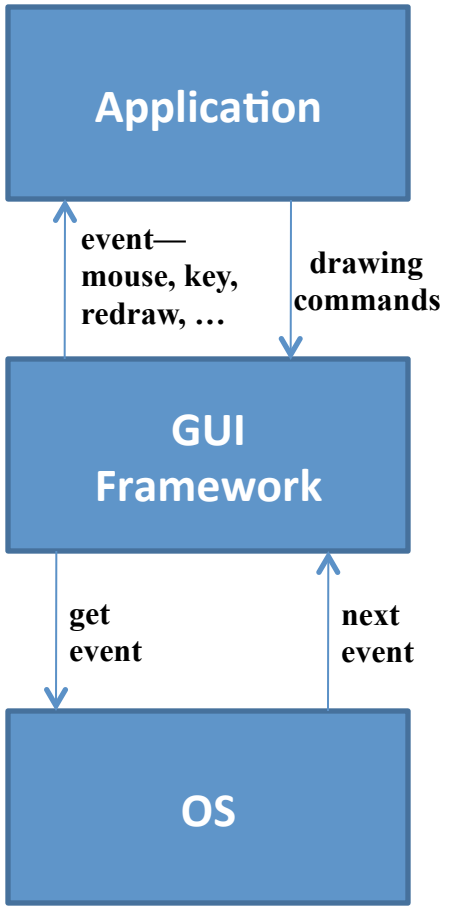
Interactions with users through events

- Do not block waiting for user response
- Instead, react to user events



An event-based GUI with a GUI framework

- Setup phase
  - Describe how the GUI window should look
  - Register observers to handle events
- Execution
  - Framework gets events from OS, processes events
    - Your code is mostly just event handlers



See edu.cmu.cs.cs214.rec06.alarmclock.AlarmWindow…

GUI frameworks in Java

- AWT – obsolete except as a part of Swing
- Swing – the most widely used, by far
- SWT – Little used outside of Eclipse
- JavaFX – Billed as a replacement for Swing
  - Released 2008 – has yet to gain traction
- A bunch of modern (web & mobile) frameworks
  - e.g., Android

GUI programming is inherently multi-threaded

- Swing *Event dispatch thread* (EDT) handles all GUI events
  - Mouse events, keyboard events, timer events, etc.
- No other time-consuming activity allowed on the EDT
  - Violating this rule can cause liveness failures

Ensuring all GUI activity is on the EDT

- Never make a Swing call from any other thread
  - "Swing calls" include Swing constructors
- If not on EDT, make Swing calls with `invokeLater`:

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(() -> new Test().setVisible(true));
}
```

Callbacks execute on the EDT

- You are a guest on the Event Dispatch Thread!
  - Don’t abuse the privilege
- If > a few ms of work to do, do it *off* the EDT
  - `javax.swing.SwingWorker` designed for this purpose

Components of a Swing application

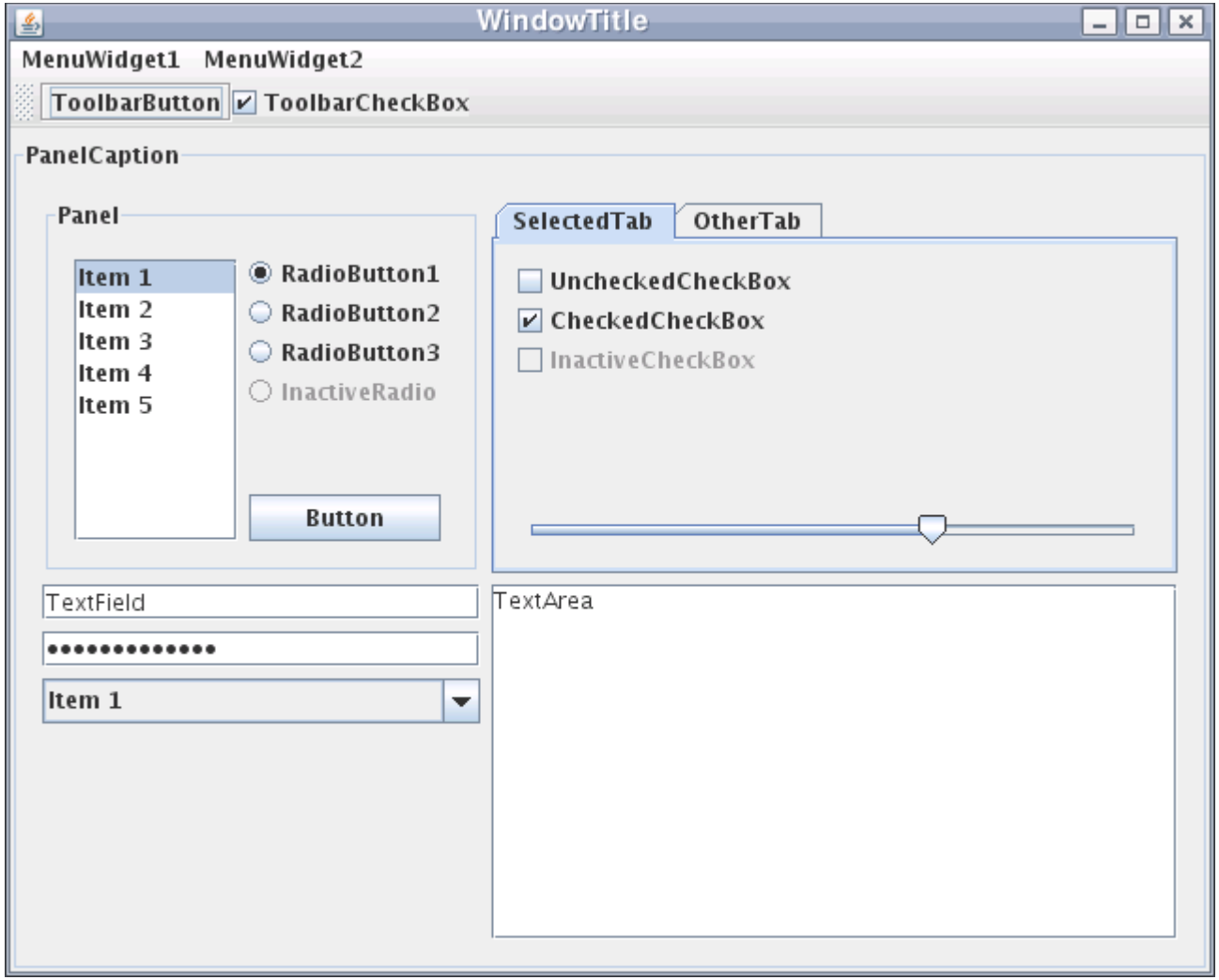
JFrame

JPanel

JButton

JTextField

...



Swing has many *widgets*

- `JLabel`
- `JButton`
- `JCheckBox`
- `JChoice`
- `JRadioButton`
- `JTextField`
- `JTextArea`
- `JList`
- `JScrollBar`
- … and more

- `JFrame` is the Swing Window
- `JPanel` (a.k.a. a pane) is the container to which you add your components (or other containers)

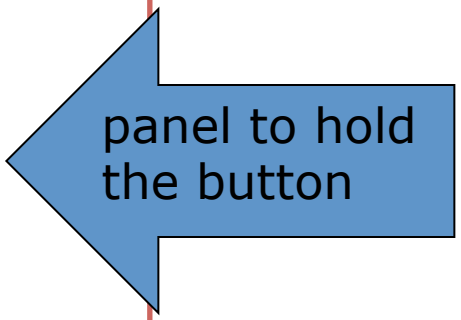
To create a simple Swing application

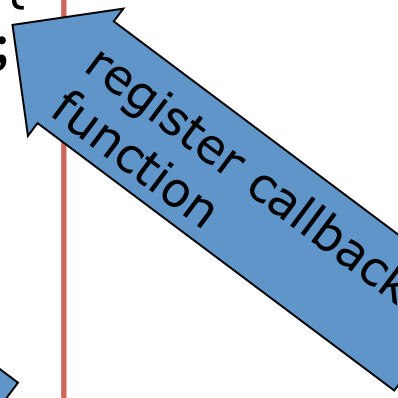
- Make a window (a `JFrame`)
- Make a container (a `JPanel`)
  - Put it in the window
- Add components (buttons, boxes, etc.) to the container
  - Use layouts to control positioning
  - Set up observers (a.k.a. listeners) to respond to events
  - Optionally, write custom widgets with application-specific display logic
- Set up the window to display the container

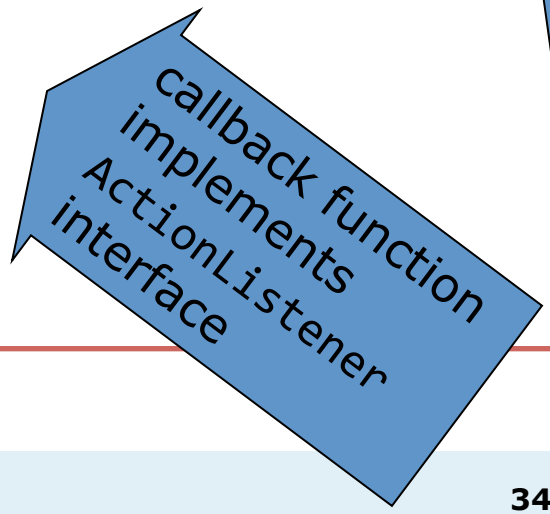
- Then wait for events to arrive…

E.g., creating a button

```
//static public void main…
JFrame window = …

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            System.out.println("Button clicked");
      }
});
panel.add(button);

window.setVisible(true);
```

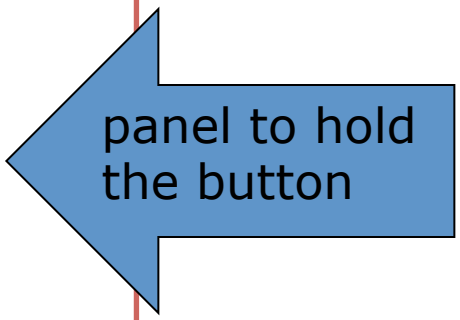


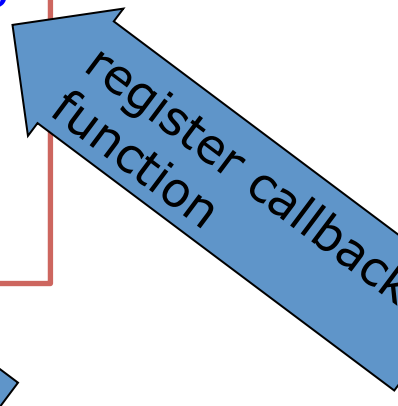


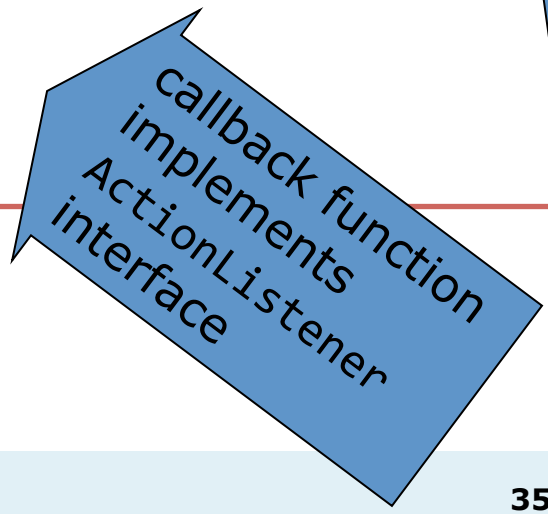


E.g., creating a button

```
//static public void main…
JFrame window = …

JPanel panel = new JPanel();
window.setContentPane(panel);

JButton button = new JButton(“Click me”);
button.addActionListener( (e) -> {
            System.out.println(“Button clicked");
});
panel.add(button);

window.setVisible(true);
```







The javax.swing.ActionListener

- Listeners are objects with callback functions
  - Can be registered to handle events on widgets
  - All registered widgets are called if event occurs

```
interface ActionListener {
        void actionPerformed(ActionEvent e);
}
```

```
class ActionEvent {
        int when;
        String actionCommand;
        int modifiers;
        Object source();
        int id;
        …
}
```

Button design discussion

- Button implementation should be reusable but customizable
  - Different button label, different event-handling
- Must decouple button's action from the button itself
- Listeners are separate independent objects
  - A single button can have multiple listeners
  - Multiple buttons can share the same listener

Swing has many event listener interfaces

- ActionListener
- AdjustmentListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- TreeExpansionListener
- TextListener
- WindowListener
- …

```
class ActionEvent {
        int when;
        String actionCommand;
        int modifiers;
        Object source();
        int id;
```

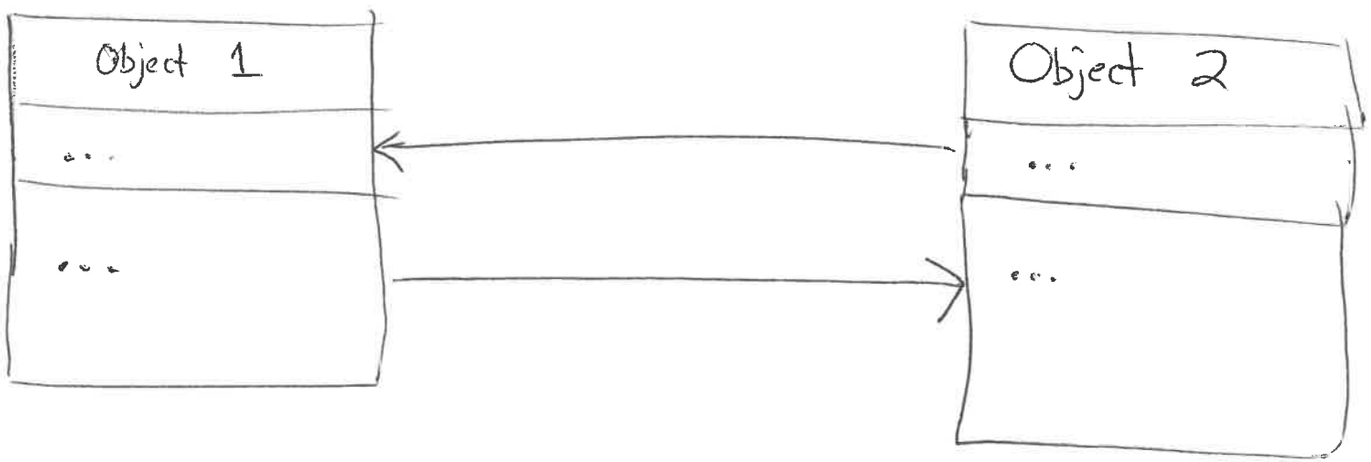
```
interface ActionListener {
        void actionPerformed(ActionEvent e);
}
```

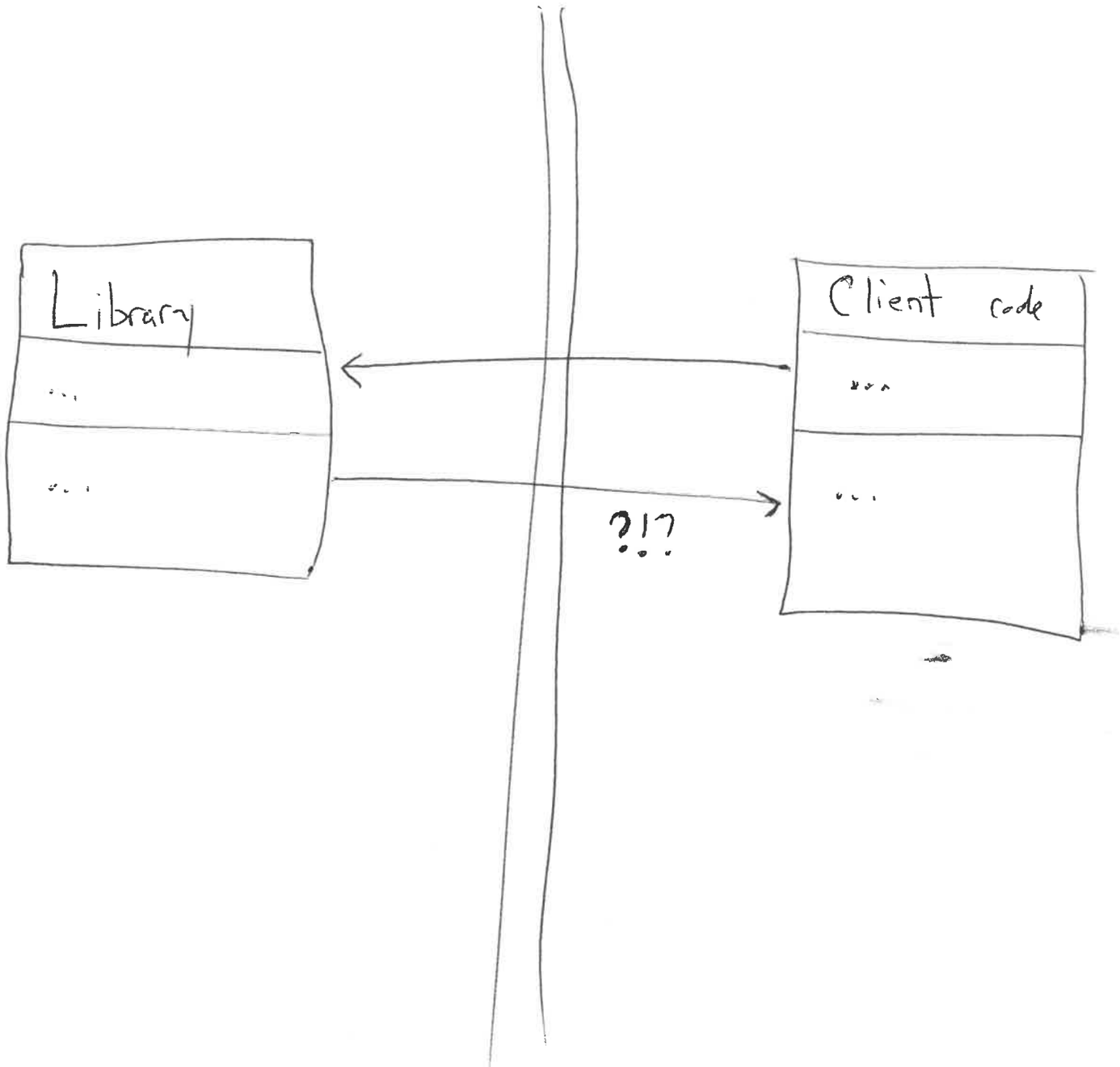
Design discussion: Decoupling your game from your GUI

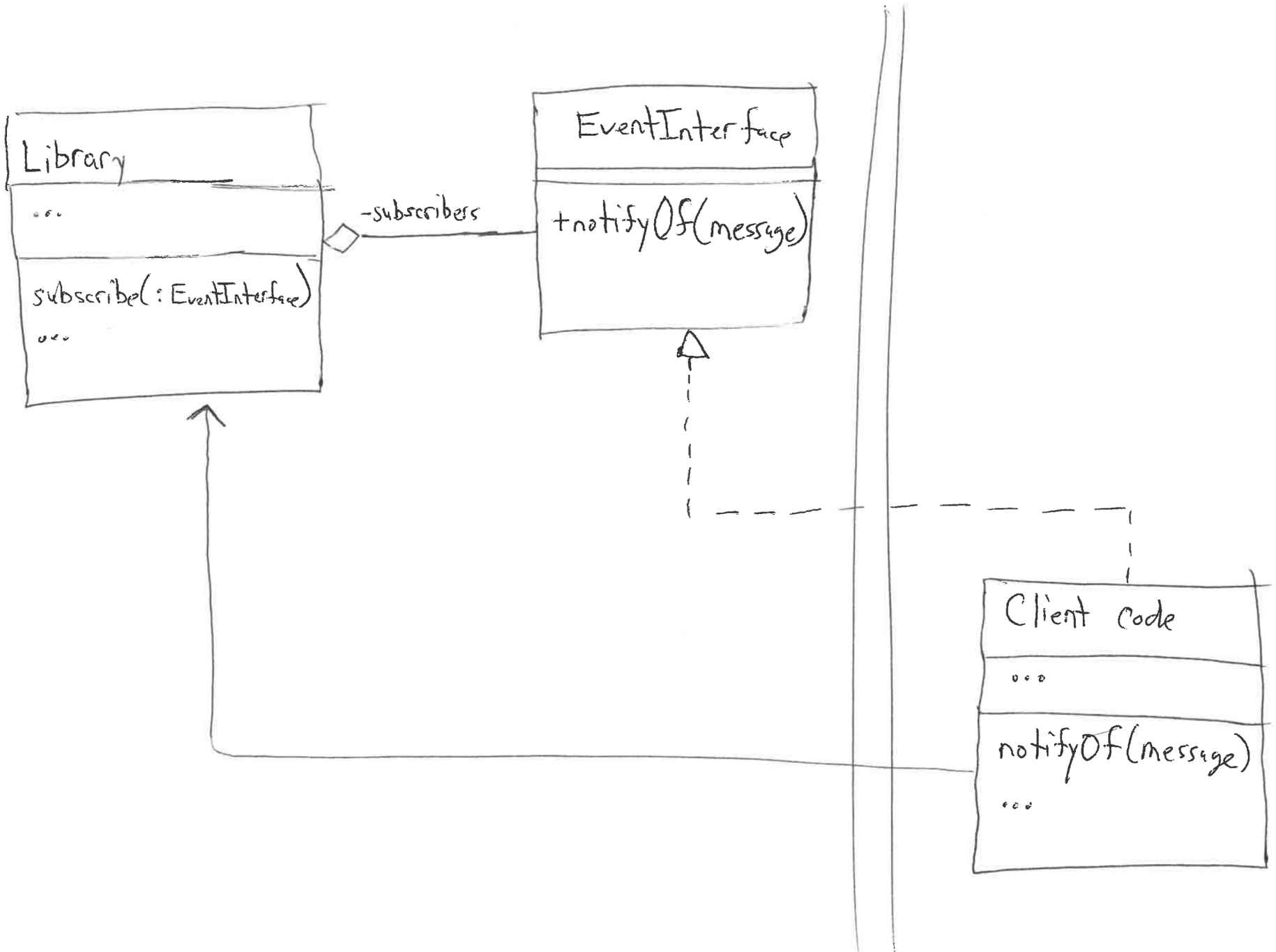
Summary

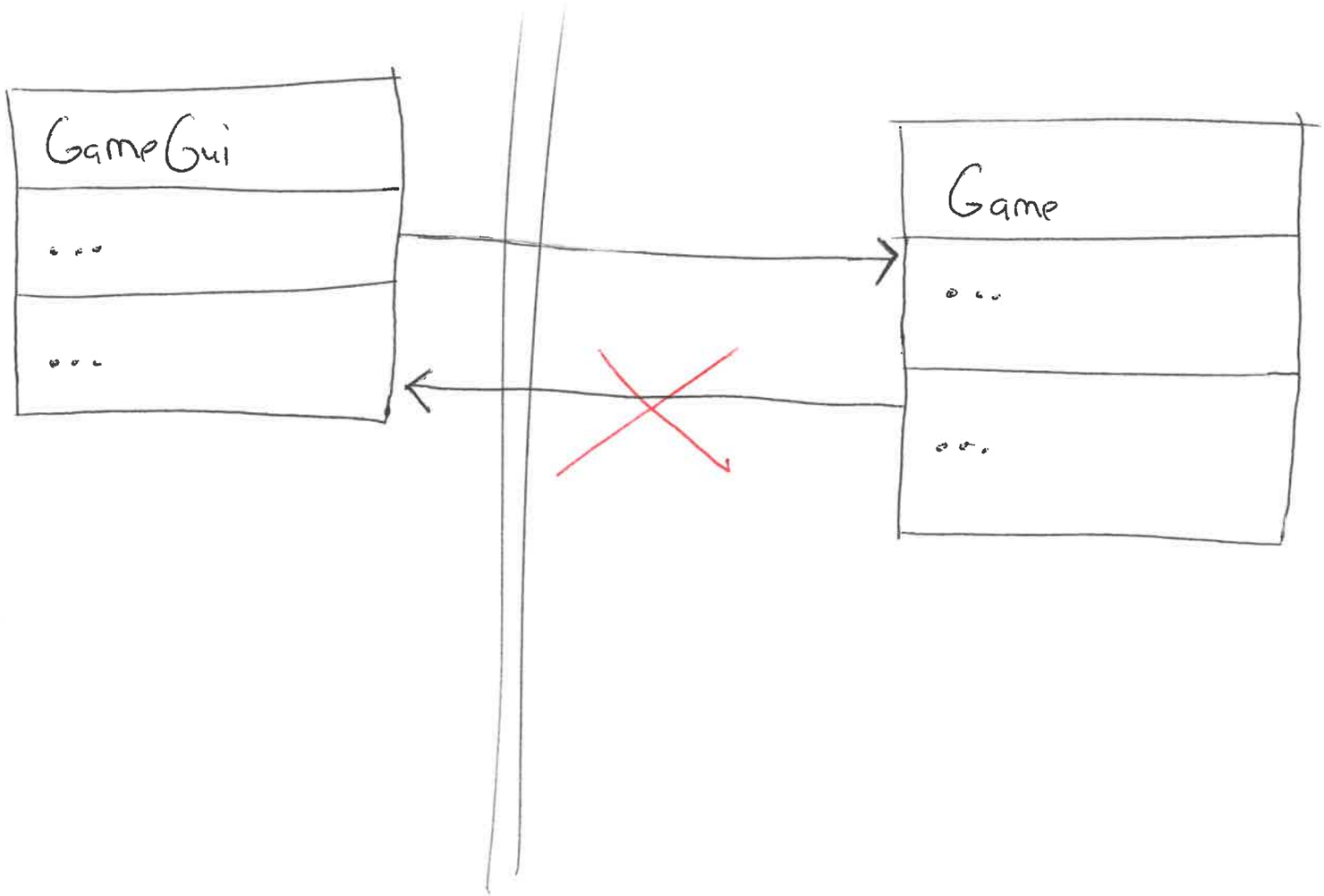
- Use the observer pattern to decouple two-way dependences
- Multi-threaded programming is genuinely hard
  - Neither under- nor over-synchronize
  - Immutable types are your friend
- GUI programming is inherently multi-threaded
  - Swing calls must be made on the event dispatch thread
  - No other significant work should be done on the EDT

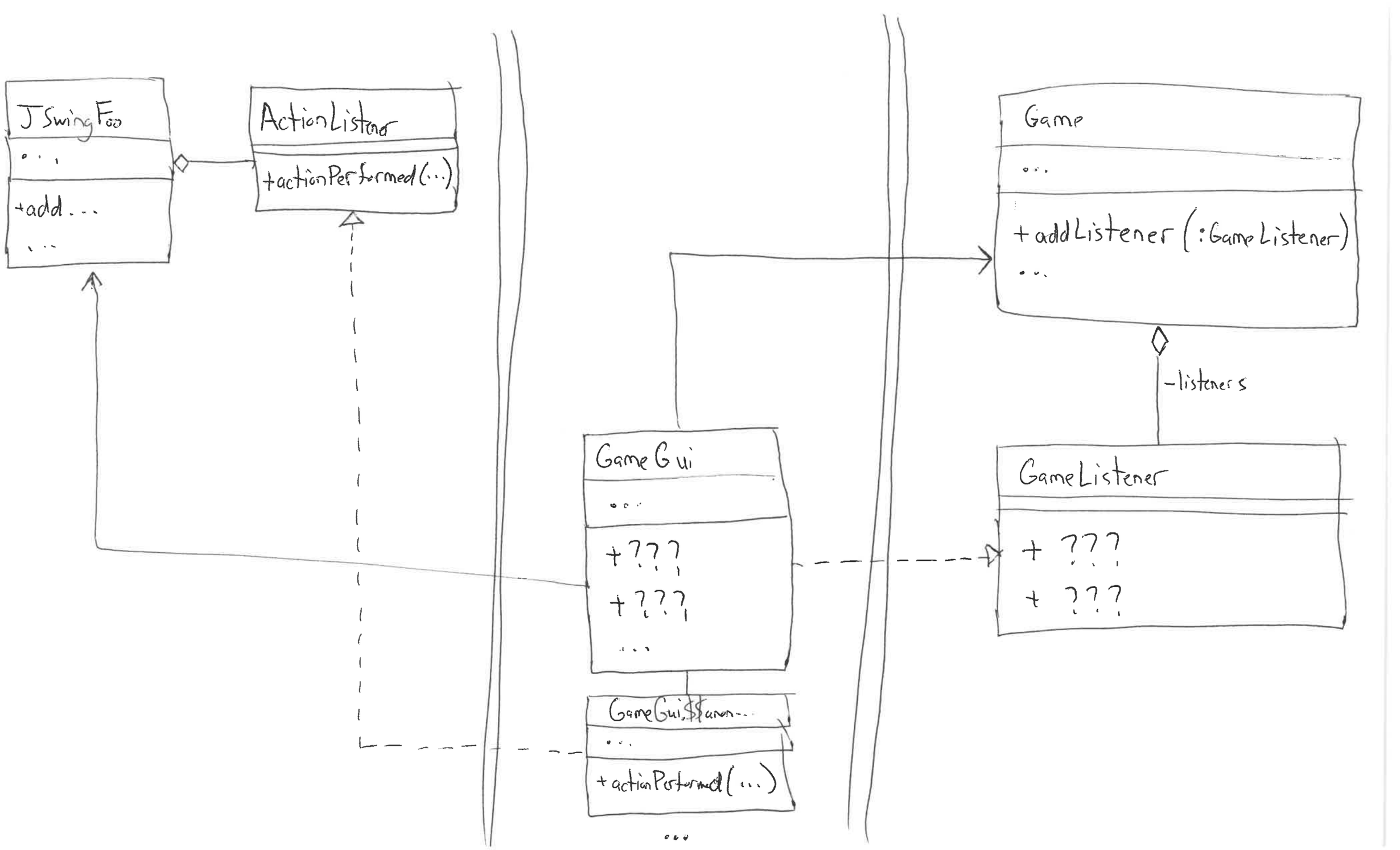
Paper slides from lecture are scanned below..


Object 1
Object 2


Library
...
...
Client code
...
...
?!?


Library
...
subscribe(: EventInterface)
...
-subscribers
EventInterface
+notifyOf(message)
Client code
...
notifyOf(message)
...


GameGui
...
...
Game
...
...


JSwingFoo
...
+add...
...
ActionListener
+actionPerformed(...)
Game
...
+addListener(:GameListener)
...
-listeners
GameGui
...
+???
+???
...
GameListener
+???
+???
GameGui$$anon...
...
+actionPerformed(...)
...