Graphical User Interfaces

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University

Acknowledgments

- “GUI programming” slides, U. Virginia
  - `www.cs.virginia.edu/javaprogramdesign/slides/5.0/gui.ppt`
- UTEP course, Advanced Object-Oriented Programming, Fall 2009
  - `www.cs.utep.edu/cheon/cs3331/notes/gui.ppt`

Human Computer Interaction

- Original computer interfaces
  - Sequences of numbers/characters:
    punch cards
    paper tape
    console switches and lights
    line printers
    magnetic tape
    typewriter terminals (teletypes)
  - Original interactive interfaces were based on sequential streams of characters
  - Well suited to "console-based programming" which focuses on sequences of text input and output
    Unix shell (bash)
    Windows command line

Text processing template in Console-based programming

```
while (! input.endOfFile) {
     read input;
     process input;
   }
```

Sometimes end-of-file is handled as an exception:

```
   while (true) {
     try {
       read input;
       process input;
     }
     catch(EOFException e)
   { break; }
}
```

Console program

```
Method main() {
    statement1;
    statement2;
    ...
    statementm;
}
```

Console programs begin and end in method main()

Graphical interaction

- Labeled buttons
- Text boxes for character input
- Mouse actions for selection, cutting, pasting, tracking/drawing
- What is the programmatic interface for graphical interaction (*i.e.*, what does the program receiving graphical input see)?

A sequence of *events*

- What are events?

GUI-based programming

GUI Program

```
main() {
   GUI gui = new GUI();
}

GUI Constructor() {
   constructor$_1$;
   constructor$_2$;
   ...
   constructor$_n$;
}

Action Performer() {
   action$_1$;
   action$_2$;
   ...
   action$_k$;
}
```

GUI program begins in method main(). The method creates a new instance of the GUI by invoking the GUI constructor. On completion, the event dispatching loop is begun

Constructor configures the components of the GUI. It also registers the listener-performer for user interactions

Event-dispatching loop

```
do
   if an event occurs
      then signal its
      action listeners
until program ends
```

The event-dispatching loop watches for user interactions with the GUI. When an event occurs, its listener-performers are notified

The action performer implements the task of the GUI. After it completes, the event-dispatching loop is restarted

Event Handling

- Mechanism to write control code
- Composed of:
  - Event
  - Event source
  - Event listener (or handler)

Event Handling (Cont.)

- Event
  - A way for GUI components to communicate with the rest of application
  - Implemented as instances of event classes (e.g., ActionEvent)
- Event source
  - Components generating events
  - Examples: buttons, check boxes, dialog boxes, mouse canvases, etc.

Event Handling (Cont.)

- Event listener (or handler)
  - Objects that receives and processes events
  - Must implement an appropriate *listener* interface
  - Must inform the source its interest in handling a certain type of events (by registering)
  - May listen to several sources and different types of events

What are Events?

- In an OO language, they are objects describing graphical inputs.
- A GUI (graphical user interface) library defines and supports the event system.
- The nitty gritty systems level code supporting the event library is based on interrupt-handling in the operating system (Comp 221, Comp 421)
- From the perspective of high-level language programming, events are are simply the elements of a much richer input stream.
- Very close connection between text processing and event processing

Event Processing Template

-
  ```
  while (true) {
     get next event;
     process event;
  }
  ```
- Processing event may terminate application. The event may be “close this application”; code simply performs any necessary clean-up and breaks.
- This event processing template is performed in a single dedicated *event dispatch thread*

Simple GUI Threading

- How do GUI events get recognized and processed?
- A GUI system *requires* a dedicated thread that performs the event loop we showed earlier:

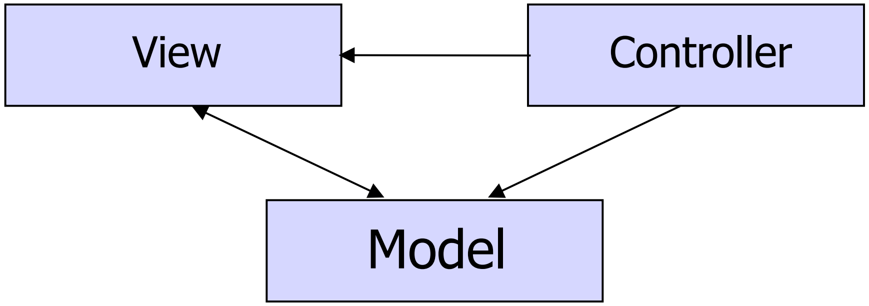
```
while (true) {
   get next event;
   process event;
 }
```

- How do we avoid perils of multi-threading? The event thread is started by the GUI library when the GUI is activated. Prior to that time, no event handling thread exists.
- In the Model-View-Controller pattern, the controller usually dies immediately after activating the GUI so there is never more than one thread executing in the program (unless the GUI code introduces new threads)!

The Model View Controller pattern for Graphical User Interfaces

(Section 3.1.1 of EOOPD notes)

- Three components
  - Model (core application with no external interface)
  - View (one or more graphical frames that interact with the user)
  - Controller ("main" program that constructs model and view, and links them together)



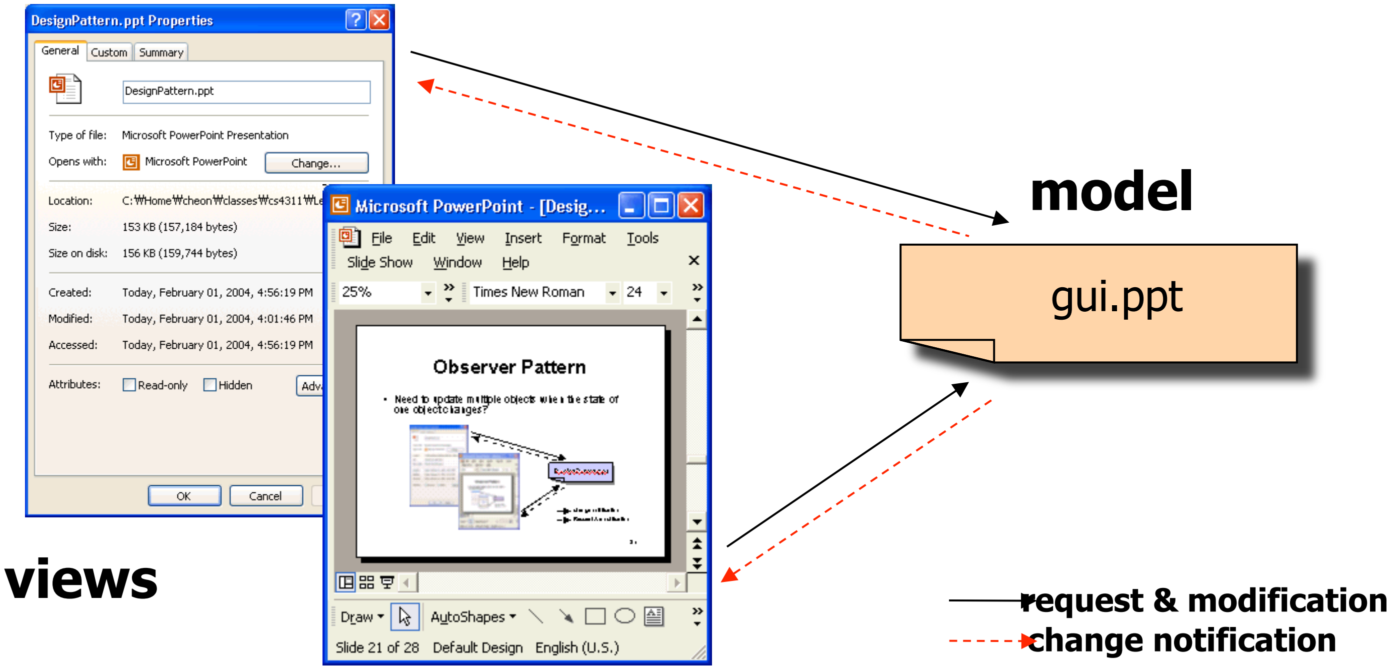
MVC startup sequence

- On startup, the controller (which in sophisticated uses of MVC should be an object) runs.
- The controller creates the model (which should be structured as an object) and the GUI (called the “view”) which is also organized as an object (*e.g*. a JFrame)
    - The GUI includes an event-handling thread which is sometimes referred to as the “GUI thread” or “event thread”.
- The controller links the view to the model by attaching listeners (commands) to GUI elements (buttons, etc) that are run when that GUI element is activated. The listener code performs some action on the model. The model is passive; it does not talk to the GUI.
- In simple uses of the MVC pattern, the controller activates the GUI and immediately terminates.

Motivation for MVC

- We partition GUI applications using the MVC pattern so that new views can potentially be created without changing the logic (model) of the underlying application.
- This is a specific example of the general design concept called *de-coupling*: partition an application into independent components that interact only through explicitly declared interfaces.
- In the Laundry program, we collapsed the model and view into a single program unit for the sake of brevity but we lost flexibility (changing view components, multiple view components) as a result.

Example of Model with Multiple Views



DesignPattern.ppt Properties

General | Custom | Summary

DesignPattern.ppt

Type of file: Microsoft PowerPoint Presentation

Opens with: Microsoft PowerPoint

Location: C:₩Home₩cheon₩classes₩cs4311₩Le

Size: 153 KB (157,184 bytes)

Size on disk: 156 KB (159,744 bytes)

Created: Today, February 01, 2004, 4:56:19 PM

Modified: Today, February 01, 2004, 4:01:46 PM

Accessed: Today, February 01, 2004, 4:56:19 PM

Attributes: ☐ Read-only ☐ Hidden

Microsoft PowerPoint - [Desig...

25%

Times New Roman 24

Observer Pattern

- Need to update multiple objects when the state of one object changes?

Slide 21 of 28 Default Design English (U.S.)

model

gui.ppt

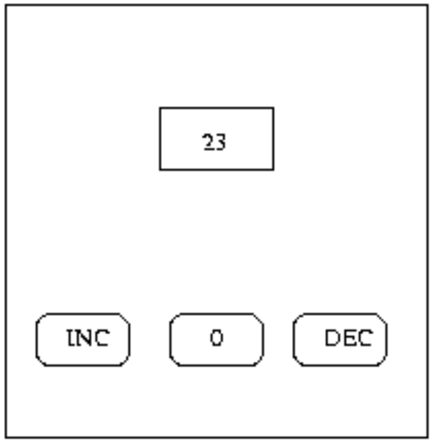
views

request & modification

change notification

Click Counter Example (Sections 3.1.2 – 3.1.4)

- Counter is initialized to 0
- Possible actions
  - INC --- increment (active when value < MAX)
  - DEC --- decrement (active when value > 0)
  - ZERO --- reset (active when value > 0)



init() method in Controller Class: note use of anonymous inner classes …

```
counter = new ClickCounter();
view = new ClickCounterView(this); view.setMinimumState();
view.setValueDisplay(counter.toString());
view.addIncListener(new ActionListener(){
           public void actionPerformed(ActionEvent event) {
               if (counter.isAtMaximum()) return;
               if (counter.isAtMinimum()) view.setCountingState();
               counter.inc(); view.setValueDisplay(counter.toString());
               if (counter.isAtMaximum()) view.setMaximumState();
           }
    });
view.addDecListener(new ActionListener(){
           public void actionPerformed(ActionEvent event) {
               if (counter.isAtMinimum()) return;
               if (counter.isAtMaximum()) view.setCountingState();
               counter.dec(); view.setValueDisplay(counter.toString());
               if (counter.isAtMinimum()) view.setMinimumState();
           }
    });
view.addResetListener(new ActionListener(){
           public void actionPerformed(ActionEvent event) {
               counter.reset(); view.setMinimumState();
           }
    });
```

… which are more robust than use of naming conventions

Event	Listener	Adapter
ActionEvent	*ActionListener*	
ComponentEvent	*ComponentListener*	ComponentAdapter
FocusEvent	*FocusListener*	FocusAdapter
KeyEvent	*KeyListener*	KeyAdapter
MouseEvent	*MouseListener*	MouseAdapter
	MouseMotionListener	MouseMotionAdapter
WindowEvent	*WindowListener*	WindowAdapter
ItemEvent	*ItemListener*	
TextEvent	*TextListener*	
…		

Example of using naming convention instead of anonymous inner class (Don’t do this!)

```
public class Util {
   public static ComponentListener
       createComponentListener(int width, int height) {
       return new MyComponentListener(width, height);
   }

   private static class MyComponentListener extends ComponentAdapter {
       private int width, height;
       public MyComponentListener(int w, int h) { width = w; height = h; }
       public void componentResized(ComponentEvent e) {
           Component c = e.getComponent();
           if (c.getWidth() < width || c.getHeight() < height) {
              c.setSize(Math.max(width, c.getWidth()),
                               Math.max(height, c.getHeight()));
           }
       }
   } // MyComponentListener
}
```

Parallelism and Concurrency Challenges in GUI's

- Problem 1: GUI can become unresponsive if an event takes a long time to process
- Solution: execute time-consuming part of event handler in a separate (background) thread, but this raises the issue of thread interference (race conditions) within the GUI.
- Problem 2: Interference (race conditions) between main thread and GUI's event dispatch thread
- Solution: terminate main thread (the controller) after starting GUI.
- Problem 3: GUI framework does not show improved performance on multicore processors
- Solution: build a multithreaded GUI framework (challenging problem because of potential thread interference!)

Example of long-running task with user feedback in GUI
(3 levels of anonymous inner classes!)

```
private void longRunningTaskWithFeedback() {
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      button.setEnabled(false);
       label.setText("busy");
       new Thread(new Runnable() {
         public void run() {
           try { /* Do big computation */ }
           finally {
             GuiExecutor.instance().execute(
               new Runnable() {
                 public void run() {
                   button.setEnabled(true);
                   label.setText("idle");
                 }}); // Runnable for GuiExecutor
           }}}).start(); // Runnable for new Thread
    }}); // ActionListener
} // Source: http://www.javaconcurrencyinpractice.com/listings/ListenerExamples.java
```

Note: `GuiExecutor` is available as open source code from the above site, but is NOT part of the Java Concurrent utilities library.

Parallel Programs vs. Concurrent Programs

- Both terms are often used interchangeably, but there is an informal and useful distinction …
- A parallel program is one for which there is a "naturally equivalent" sequential program
  - Motivation for parallelism is to obtain improved performance on parallel processor
- A concurrent program is one which deals with "intrinsically distributed" entities
  - Motivation for parallelism/concurrency comes from application domain, *e.g.*, asynchronous GUI threads, an airline reservation system.

Confusing distinction when both kinds of programs are implemented in same language

- Same language constructs (*e.g.*, threads, locks) used in both cases for very different purposes
- Concurrent programs need these capabilities even when computers are single-core
  - Origin of a number of low-level concurrency primitives in system software and hardware

More Complex GUIs

- In some cases the model must be aware of the GUI, e.g., DrJava or a game playing program.
- The model includes an object supporting interface with limited GUI functionality. In some sense, this GUI interface is part of the API of the model. The model must include an operation (*e.g.*, a constructor or subsequent initializer method) that takes an argument of GUI interface type.
- The GUI interface should be developed as part of the model and include only what the model needs.