Principles of Software Construction:

Introduction to Multithreading and GUI Programming

Christian Kaestner and Bogdan Vasilescu
Administrivia

• Homework 4a due **tonight**
  – Please follow naming conventions
  – Please mark last commit
• Homework 4b due March 9
• Reading for Tuesday: Adapter and Abstr. Factory
Key concept from yesterday's recitation

- Discovering design patterns
- The Observer pattern
The Observer design pattern

```
<<stereotype>>
Subject
{abstract}
+Attach(observer: Observer): void
+Detach(observer: Observer): void
+Notify()

<<stereotype>>
Observer
{interface}
+Update(): abstract

Notify()
{
    for all o on observers {
        o.Update();
    }
}

<<stereotype>>
ConcreteSubject
+subjectState: State
+GetState(): State
+setState(state: State): void

<<stereotype>>
ConcreteObserver
+Update()
The Observer design pattern

- **Applicability**
  - When an abstraction has two interdependent aspects and you want to reuse both
  - When state change to one object requires notifying others, without becoming dependent on them

- **Consequences**
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect
EVENT-BASED PROGRAMMING
Event-based programming

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

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

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

```java
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, expands a tree
- Mouse hovers over a widget, focus changes
- Scrolling, mouse wheel turned
- Resizing a window, hiding a window
- Drag and drop

- A package arrives from a web service, connection drops, ...
- System shutdown, ...
Interaction with command-line

Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = question.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
GUIs without event-based programming

```plaintext
while (true) {
    if (isKeyDown("Alt+Q"))
        break;
    if (isKeyDown("F1"))
        openHelp();
    if (isMouseDown(10 ...)
        startMovingWindow();
    ...
}
```
Event-based GUIs

//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email =
    emailField.getText();
emaillist.add(email);
Event-based GUIs

```java
//static public void main...
JFrame window = ...
window.setDefaultCloseOperation(
    WindowConstants.EXIT_ON_CLOSE);
window.setVisible(true);

//on add-button click:
String email = emailField.getText();
emaillist.add(email);

//on remove-button click:
int pos = emaillist.getSelectedItem();
if (pos>=0) emaillist.delete(pos);
```
(Blocking) Interactions with users

- newGame
- addCards
- addAction
- action
- [action==hit] addCard
Interactions with users through events

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

— e.g.:

- Game
- Dealer
- Player

newGame

addCards

addCards

hit

addCard
Typically use a GUI framework

• Register code (a.k.a. callbacks, observers) to handle events
• Operating system/GUI framework detects events, determines which components are registered to handle the event and calls the event handlers
• Program exits by calling some exit method
Programming an event-based GUI

• Setup phase
  – Describe how the GUI window should look
  – Use libraries for windows, widgets, and layout
  – Embed specialized code for later use
  – Register callbacks

• Execution
  – Framework gets raw events from OS (e.g., mouse clicks, key presses, window becomes visible)
  – Framework processes events (e.g., click at 10,40: which widget)
  – Triggers callback functions of corresponding widgets (if registered)
Example: The AlarmWindow

- ...edu.cmu.cs.cs214.rec06.alarmclock.AlarmWindow
  - Creates a JFrame with a JPanel to go in it
  - Creates a text label and a button
  - Makes the window (and its contents) visible when the alarm goes off

- When the dismiss button is clicked, its event handler hides the window
Example: The CustomerManagementUI

• ...rec06.customerlist.gui.CustomerManagementUI
  – Creates a JFrame with a JPanel to go in it
  – Makes the window (and its contents) visible
• ...rec06.customerlist.gui.CustomerManagementPanel
  – Creates numerous labels and text fields, a customerAddButton
  – Registers an event handler for the customerAddButton
• When the customerAddButton is clicked, its event handler gets the text from the text fields and adds a customer to the list
MULTITHREADED PROGRAMMING BASICS
What is a thread?

• Short for *thread of execution*
• Multiple threads run in same program concurrently
• Threads share the same address space
  – Changes made by one thread may be read by others
• Multithreaded programming
  – Also known as shared-memory multiprocessing
Threads vs. processes

• Threads are lightweight; processes heavyweight
• Threads share address space; processes have own
• Threads require synchronization; processes don’t
  – Threads hold locks while mutating objects
• It’s unsafe to kill threads; safe to kill processes
Why use threads?

• Performance in the face of blocking activities
  – Consider a web server
• Performance on multiprocessors
• Cleanly dealing with natural concurrency
• In Java threads are a fact of life
  – Example: garbage collector runs in its own thread
Example: generating cryptarithms

```java
static List<String> cryptarithms(String[] words, int start, int end) {
    List<String> result = new ArrayList<>();
    String[] tokens = new String[] {"", "+", "", ".", "};

    for (int i = start; i < end - 2; i++) {
        tokens[0] = words[i];  tokens[2] = words[i + 1];
        tokens[4] = words[i + 2];
        try {
            Cryptarithm c = new Cryptarithm(tokens);
            if (c.solve().size() == 1)
                result.add(c.toString());
        } catch (IllegalArgumentException e) {
            // too many letters; ignore
        }
    }
    return result;
}
```
Single-threaded driver

public static void main(String[] args) {
    long startTime = System.nanoTime();
    List<String> cryptarithms = cryptarithms(words, 0, words.length);
    long endTime = System.nanoTime();

    System.out.printf("Time: %ds\n", (endTime - startTime)/1e9);
    System.out.println(cryptarithms);
}
public static void main(String[] args) throws InterruptedException {
    int n = Integer.parseInt(args[0]);  // Number of threads
    long startTime = System.nanoTime();
    int wordsPerThread = words.length / n;
    Thread[] threads = new Thread[n];
    Object[] results = new Object[4];
    for (int i = 0; i < n; i++) {  // Create the threads
        int start = i == 0 ? 0 : i * wordsPerThread - 2;
        int end = i == n-1 ? words.length : (i + 1) * wordsPerThread;
        int j = i; // Only constants can be captured by lambdas
        threads[i] = new Thread(() -> {
            results[j] = cryptarithms(words, start, end);
        });
    }
    for (Thread t : threads) t.start();
    for (Thread t : threads) t.join();
    long endTime = System.nanoTime();
    System.out.printf("Time: %ds\n", (endTime - startTime)/1e9);
    System.out.println(Arrays.toString(results));
}
Cryptarithm generation performance

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Seconds to run</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.0</td>
</tr>
<tr>
<td>2</td>
<td>13.5</td>
</tr>
<tr>
<td>3</td>
<td>11.7</td>
</tr>
<tr>
<td>4</td>
<td>10.8</td>
</tr>
</tbody>
</table>

Generating all cryptarithms from a corpus of 344 words
• Test all consecutive 3-word sequences (342 possibilities)
• Test machine is this crappy old laptop (2 cores, 4 hyperthreads)
• I did *not* follow benchmarking best practices!
What requires synchronization?

- Shared mutable state
- If not properly synchronized, all bets are off!
- You have three 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
Synchronization is tricky

• Too little and you risk safety failure
  – Changes aren’t guaranteed to propagate thread to thread
  – Program can observe inconsistencies
  – Critical invariants can be corrupted

• Too much and program may run slowly or not at all
  – Deadlock or other liveness failure
Contention kills performance

• Synchronized is the opposite of concurrent!
• Highly concurrent code is possible to write
  – But it’s very difficult to get right
  – If you get it wrong you’re toast
• Let Doug Lea write it for you!
  – ConcurrentHashMap
  – Executor framework
  – See java.util.concurrent
Safety vs. liveness

• Safety failure – incorrect computation
  – Can be subtle or blatant
• Liveness failure – no computation at all
• Temptation to favor liveness over safety
  – Don’t succumb!
• Safety failures offer a false sense of security
• Liveness failures force you to confront the bug
Synchronization in cryptarithm example

• How did we avoid synchronization in our multithreaded cryptarithm generator?
  • *Embarrassingly parallelizable computation*
  • Each thread is entirely independent of the others
    – They try different cryptarithms
    – And write results to different array elements
• No shared mutable state to speak of
  – Main thread implicitly syncs with workers with `join`
GUI PROGRAMMING
There are many Java GUI frameworks

• 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 multithreaded

- **Event-driven programming**
- Event dispatch thread (EDT) handles all GUI events
  - Mouse events, keyboard events, timer events, etc.
- Program registers callbacks ("listeners")
  - Function objects invoked in response to events
  - Observer pattern
Ground rules for GUI programming

1. All GUI activity is on event dispatch thread
2. No other time-consuming activity on this thread
   – Blocking calls (e.g., I/O) absolutely forbidden

• Many GUI programs violate these rules
  – They are broken
• Violating rule 1 can cause safety failures
• Violating rule 2 can cause liveness failures
Ensuring all GUI activity is on EDT

• Never make a Swing call from any other thread
• Swing calls includes Swing constructors
• If not on EDT, make Swing calls with invokeLater:

```java
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 you do, liveness will suffer
  – Your program will become non-responsive
  – Your users will become angry
• If > a few ms of work to do, do it off the EDT
  – javax.swing.SwingWorker designed for this purpose
Swing has many widgets

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

- JFrame is the Swing Window
- JPanel (aka 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...
Creating a button

```java
//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);
```
Creating a button

```java
//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);
```
Creating a button, Java 8

```java
//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);
ActionListeners

- Listeners are objects with callback functions
- Listeners 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;
    ...
}
```
ActionListeners

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

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

```java
class ActionEvent {
    class AbstractButton extends JComponent {
        private List<ActionListener> listeners;
        public void addActionListener(ActionListener l) {
            listeners.add(l);
        }
        protected void fireActionPerformed(ActionEvent e) {
            for (ActionListener l: listeners)
                l.actionPerformed(e);
        }
    }
}
```
ActionListeners

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

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

```java
class ActionEvent {

class AbstractButton extends JComponent {
    private List<ActionListener> listeners;
    public void addActionListener(ActionListener l) {
        listeners.add(l);
    }
    protected void fireActionPerformed(ActionEvent e) {
        for (ActionListener l: listeners)
            l.actionPerformed(e);
    }
}
```
Recall the observer design pattern
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
- ...and on and on...

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

```java
class ActionEvent {
    int when;
    String actionCommand;
    int modifiers;
    Object source();
}
```
DECOUPLING THE GUI
A GUI design challenge

• Consider a blackjack game, implemented by a Game class:
  – Player clicks “hit” and expects a new card
  – When should the GUI update the screen?
A GUI design challenge, extended

• What if we want to show the points won?
Game updates GUI?

- What if points change for reasons not started by the GUI? (or computations take a long time and should not block)
Game updates GUI?

• Let the Game tell the GUI that something happened
Game updates GUI?

- Let the Game tell the GUI that something happened

Problem: This couples the World to the GUI implementation.
Core implementation vs. GUI

• Core implementation: Application logic
  – Computing some result, updating data

• GUI
  – Graphical representation of data
  – Source of user interactions

• Design guideline: *Avoid coupling the GUI with core application*
  – Multiple UIs with single core implementation
  – Test core without UI
  – *Design for change, design for reuse, design for division of labor; low coupling, high cohesion*
Decoupling with the Observer pattern

- Let the Game tell *all* interested components about updates
Separating application core and GUI, a summary

• Reduce coupling: do not allow core to depend on UI

• Create and test the core without a GUI
  – Use the Observer pattern to communicate information from the core (Model) to the GUI (View)
An architectural pattern: Model-View-Controller (MVC)

- Manage inputs from user: mouse, keyboard, menu, etc.
- Manage display of information on the screen
- Manage data related to the application domain
Model-View-Controller (MVC)

Passive model

Active model

For help writing Swing code

• Sun wrote a good tutorial
  – http://docs.oracle.com/javase/tutorial/uiswing/
• The many components shown with examples
  – http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html
• Listeners supported by each component
  – http://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html
Summary

• Multithreaded programming is genuinely hard
  – But it’s a fact of life in Java
• Neither under- nor over-synchronize
  – Immutable types are your best friend
  – java.util.concurrent is your next-best friend
• GUI programming is limited form of multithreading
  – Swing calls must be made on event dispatch thread
  – No other significant work should be done on EDT
• GUIs are full of design patterns