Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Designing (Sub)systems

Introduction to concurrency and GUls

Michael Hilton  Bogdan Vasilescu
Administrivia

• Homework 4b due Thursday, March 7th
  – Homework 4a feedback coming soon
• Reading due Tuesday: UML and Patterns Chapters 26.1 and 26.4
  – Adapter pattern
  – Factory pattern
• Still need Midterm 1?
Part 1: Design at a Class Level

Design for Change: Information Hiding, Contracts, Unit Testing, Design Patterns

Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns

Part 2: Designing (Sub)systems

Understanding the Problem

Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies

Testing Subsystems

Design for Reuse at Scale: Frameworks and APIs

Part 3: Designing Concurrent Systems

Concurrency Primitives, Synchronization

Designing Abstractions for Concurrency
Key concepts from Thursday
Key concepts from Thursday

• Revise, revise, revise: Refactoring and anti-patterns
• Test driven development
Key concepts from last 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...
Today

• Observer pattern
• Introduction to concurrency
• Introduction to GUIs
MULTITHREADED PROGRAMMING BASICS
What is a thread?

- Short for *thread of execution*
- Multiple threads can run in the 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 their own
- Threads require synchronization; processes don’t
  - Threads hold locks while mutating objects
- 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
  – In Java threads are a fact of life
    • Example: garbage collector runs in its own thread
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

```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;

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

```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("Hi mom!")).start();
    }
}
```
Aside: Anonymous inner class scope in Java

```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

```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
Threads for performance

• Generating cryptarithmetic from a 344-word corpus
  – Test all consecutive 3-word sequences: \( A + B = C \) (342 possibilities)

<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>
Shared mutable state requires synchronization

- If not properly synchronized, all bets are off!
- 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
    • Changes aren’t guaranteed to propagate thread to thread
    • Program can observe inconsistencies
    • Critical invariants can be corrupted

• Too much synchronization: *liveness failure*
  – No computation at all
    • Deadlock or other liveness failure
Today

- Observer pattern
- Introduction to concurrency
- Introduction to GUIs
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);
    yesIKnowJavaDoesntWorklikeThat(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
- 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

```java
Scanner input = new Scanner(System.in);
while (questions.hasNext()) {
    Question q = questions.next();
    System.out.println(q.toString());
    String answer = input.nextLine();
    q.respond(answer);
}
```
Blocking interactions with users
Interactions with users through events

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

Diagram:
- User interacts with Game
  - User initiates newGame
  - Game adds cards to Dealer
  - Dealer adds cards to Player
  - Player hits and receives addCard

Roles:
- User
- Game
- Dealer
- Player
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 PROGRAMMING
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

- **Event-driven programming**

- Swing *Event dispatch thread* (EDT) handles all GUI events
  - Mouse events, keyboard events, timer events, etc.

- Program registers observers ("listeners")

- No other time-consuming activity allowed on the EDT
  - Violating this rule can cause liveness failures
Ensuring all GUI activity is on the EDT

• Violating this rule can cause safety failures
  – Never make a Swing call from any other thread
    • "Swing calls" include Swing constructors

• If not on EDT, make Swing calls with `invokeLater()`
Aside: invokeLater

```java
public void actionPerformed(ActionEvent e) {
    new Thread(new Runnable() {
        final String text = readHugeFile();
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                textArea.setText(text);
            }
        }).start();
    }
}
```

https://alvinalexander.com/java/java-swingutilities-invoke-later-example-edt
Ensuring all GUI activity is on the EDT

• Violating this rule can cause safety failures
  – Never make a Swing call from any other thread
    • "Swing calls" include Swing constructors

• If not on EDT, make Swing calls with `invokeLater()`

• The code that initialises our GUI must also take place in an `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 have more than a few ms of work to do, do it off the EDT
  – javax.swing.SwingWorker designed for this purpose

• Typical scenario:
  – long running task in a background thread
  – provide updates to the UI either when done, or while processing.
SwingWorker sample usage

```java
final JLabel label;
class MeaningOfLifeFinder extends SwingWorker<String, Object> {
    @Override
    public String doInBackground() {
        return findTheMeaningOfLife();
    }

    @Override
    protected void done() {
        try {
            label.setText(get());
        } catch (Exception ignore) {
        }
    }
}

(new MeaningOfLifeFinder()).execute();
```

https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html
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

```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);
```
E.g., 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((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

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

```java
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
- ...

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

```java
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```
Summary: Swing constraints

- Time-consuming tasks should not be run on the Event Dispatch Thread. Otherwise the application becomes unresponsive.
- Swing components should be accessed on the Event Dispatch Thread only.
- Helpers: invokeLater, SwingWorker
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

• Use the observer pattern to decouple two-way dependencies
• 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