7. GUI Construction
GUI Construction

Sources

> David Flanagan, Java Foundation Classes in a Nutshell, O’Reilly.
> [http://java.sun.com/docs/books/tutorial/uiswing](http://java.sun.com/docs/books/tutorial/uiswing)
> [ant.apache.org](http://ant.apache.org)
Roadmap

> Model-View-Controller (MVC)
> Swing Components, Containers and Layout Managers
> Events and Listeners
> Observers and Observables
> Jar files, Ant and Javadoc
> Epilogue: distributing the game
Roadmap

> Model-View-Controller (MVC)
> Swing Components, Containers and Layout Managers
> Events and Listeners
> Observers and Observables
> Jar files, Ant and Javadoc
> Epilogue: distributing the game
A Graphical TicTacToe?

Our existing TicTacToe implementation is very limited:
> single-user at a time
> textual input and display

We would like to migrate it towards an interactive game:
> running the game with *graphical display and mouse input*
Version 6 of our game implements a *model of the game*, without a GUI. The GameGUI will implement a *graphical view and a controller for GUI events*.

*The MVC paradigm separates an application from its GUI so that multiple views can be dynamically connected and updated.*
Roadmap

- Model-View-Controller (MVC)
- **Swing Components, Containers and Layout Managers**
- Events and Listeners
- Observers and Observables
- Jar files, Ant and Javadoc
- Epilogue: distributing the game
The `java.awt` package defines GUI components, containers and their layout managers.

**NB:** There are also many graphics classes to define colours, fonts, images etc.
Swing JComponents

The javax.swing package defines GUI components that can adapt their “look and feel” to the current platform.
Swing Containers and Containment

Swing Containers may contain other Components

```java
Jbutton b = new JButton("Push me");
JPanel p = new JPanel();
p.add(b);
```
The **Layout Manager** defines how the components are arranged in a container (size and position).

```java
JPanel p = new JPanel(new BorderLayout());
```

```java
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
```

http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html
An example: GridLayout

A `GridLayout` places components in a grid of cells.

> Each component takes up all the space in a cell.

> Each cell is the same size

```java
GridLayout experimentLayout = new GridLayout(0, 2);
...
compsToExperiment.setLayout(experimentLayout);
compsToExperiment.add(new JButton("Button 1");
compsToExperiment.add(new JButton("Button 2");
```
The GameGUI is a JFrame using a BorderLayout (with a centre and up to four border components), and containing a JButton (“North”), a JPanel (“Center”) and a JLabel (“South”).

The central Panel itself contains a grid of squares (Panels) and uses a GridLayout.

*NB:* GameGUI and Place are the only classes that differ for AWT & Swing.
Laying out the GameGUI

class GameGUI extends JFrame implements Observer {
    ...
    public GameGUI(String title) throws HeadlessException {
        super(title);
        game = makeGame();
        ...
        this.setSize(...);
        add("North", makeControls());
        add("Center", makeGrid());
        label = new JLabel();
        add("South", label);
        showFeedBack(game.currentPlayer().mark() + " plays");
        ...
        this.show();
    }
}
Helper methods

As usual, we introduce helper methods to hide the details of GUI construction ...

```java
private Component makeControls() {
    JButton again = new JButton("New game");
    ...
    return again;
}
```
Roadmap

> Model-View-Controller (MVC)
> Swing Components, Containers and Layout Managers
> **Events and Listeners**
> Observers and Observables
> Jar files, Ant and Javadoc
> Epilogue: distributing the game
Interactivity with Events

To make your GUI do something you need to handle events

- An event is typically a *user action* — a mouse click, key stroke, etc.
- The Java Event model is used by Java AWT and Swing
  (java.awt.AWTEvent and javax.swing.event)
The program is always responsive to *user interaction*, no matter what it is doing.

The runtime of the Swing framework creates *threads* — you don’t explicitly create them.

The *Event Dispatch thread* is responsible for *event handling*. 
Instead of actively checking for GUI events, you can define *callback methods* that will be invoked when your GUI objects receive events:

**Hardware events** ...
(MotionEvent, KeyEvent, ...)  ... *are handled by subscribed Listener objects*

AWT/Swing Components *publish* events and (possibly multiple) Listeners *subscribe* interest in them.

http://java.sun.com/docs/books/tutorial/uiswing/events/index.html
Events and Listeners (II)

Every AWT and Swing component publishes a variety of different events (see java.awt.event) with associated Listener interfaces.

<table>
<thead>
<tr>
<th>Component</th>
<th>Events</th>
<th>Listener Interface</th>
<th>Listener methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>JButton</td>
<td>ActionEvent</td>
<td>ActionListener</td>
<td>actionPerformed()</td>
</tr>
<tr>
<td>JComponent</td>
<td>MouseEvent</td>
<td>MouseListener</td>
<td>mouseClicked()</td>
</tr>
<tr>
<td></td>
<td>MouseMotionEvent</td>
<td>MouseMotionListener</td>
<td>mouseDragged()</td>
</tr>
<tr>
<td></td>
<td>KeyEvent</td>
<td>KeyListener</td>
<td>keyPressed()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>keyReleased()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>keyTyped()</td>
</tr>
</tbody>
</table>

...
Listening for Button events

When we create the “New game” Button, we attach an `ActionListener` with the `Button.addActionListener()` method:

```java
private Component makeControls() {
    Button again = new Button("New game");
    again.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            showFeedback("starting new game ...");
            newGame(); // NB: has access to methods
        }
    });
    return again;
}
```

We instantiate an *anonymous inner class* to avoid defining a named subclass of `ActionListener`. 

© O. Nierstrasz

7.21
Gracefully cleaning up

A `WindowAdapter` provides an *empty implementation* of the `WindowListener` interface (!)

```java
public class GameGUI extends JFrame implements Observer {

    // ...

    public GameGUI(String title) throws HeadlessException {

        // ...

        this.addWindowListener(new WindowAdapter()
            {
                public void windowClosing(WindowEvent e) {

                    GameGUI.this.dispose();
                    // NB: disambiguate “this”!
                }
            });

        this.show();
    }
}
```
Listening for mouse clicks

We also attach a MouseListener to each Place on the board.

```java
private Component makeGrid() {
    ... 
    Panel grid = new Panel();
    grid.setLayout(new GridLayout(3, 3));
    places = new Place[3][3];
    for (Row row : Row.values()) {
        for (Column column : Column.values()) {
            Place p = new Place(column, row);
            p.addMouseListener(new PlaceListener(p, this));
            ... 
        }
    }
    return grid;
}
```
The PlaceListener

MouseListener is another convenience class that defines *empty* MouseListener methods

```java
public class PlaceListener extends MouseAdapter {
    private final Place place;
    private final GameGui gui;
    public PlaceListener(Place myPlace, GameGUI myGui) {
        place = myPlace;
        gui = myGui;
    }
    ...
```
The PlaceListener ...

*We only have to define the `mouseClicked()` method:*

```java
public void mouseClicked(MouseEvent e) {
    ...
    if (!game.notOver()) {
        try {
            ((GUIplayer) game.currentPlayer()).move(col,row);
            gui.showFeedBack(game.currentPlayer().mark() + " plays");
        } catch (AssertionException err) {
            gui.showFeedBack("Invalid move ignored ...");
        }
        if (!game.notOver()) {
            gui.showFeedBack("Game over -- " + game.winner() + " wins!");
        }
    } else {
        gui.showFeedBack("The game is over!");
    }
}
```
Roadmap

> Model-View-Controller (MVC)
> Swing Components, Containers and Layout Managers
> Events and Listeners
> **Observers and Observables**
> Jar files, Ant and Javadoc
> Epilogue: distributing the game
The Observer Pattern

> Also known as the *publish/subscribe* design pattern - to observe the state of an object in a program.

> One or more objects (called *observers*) are registered to *observe* an event which may be raised in an observable object (the *observable* object or *subject*).

> The *observable* object or *subject* which may raise an event maintains a collection of *observers*. 
Our BoardGame Implementation
A class can implement the `java.util.Observer` interface when it wants to be informed of changes in `Observable` objects.

An `Observable` object can have one or more `Observers`.

After an observable instance changes, calling `notifyObservers()` causes all observers to be notified by means of their `update()` method.
public class GameGUI extends JFrame implements Observer {

    ... 

    public GameGUI(String title) throws HeadlessException {
        super(title);
        game = makeGame();
        game.addObserver(this); // notify GameGui if state change

        ...
    }
Observing the BoardGame

In our case, the GameGUI represents a View, so plays the role of an Observer of the BoardGame TicTacToe:

```java
public class GameGUI extends JFrame implements Observer {
    ...
    public void update(Observable o, Object arg) {
        Move move = (Move) arg; // Downcast Object type
        showFeedBack("got an update: " + move);
        places[move.col][move.row].setMove(move.player);
    }
    ...
```
Observing the BoardGame ...

The BoardGame represents the *Model*, so plays the role of an *Observable* (i.e. the subject being observed):

```java
public abstract class AbstractBoardGame
    extends Observable implements BoardGame
{
    ...
    public void move(int col, int row, Player p) {
        ...
        setChanged();
        notifyObservers(new Move(col, row, p));
    }
}
```
Handy way of Communicating changes

A **Move** instance bundles together information about a change of state in a **BoardGame**:

```java
public class Move {
    public final int col, row; // NB: public, but final
    public final Player player;
    public Move(int col, int row, Player player) {
        this.col = col; this.row = row;
        this.player = player;
    }
    public String toString() {
        return "Move(" + col + "," + row + "," + player + ")";
    }
}
```
Setting up the connections

When the GameGUI is created, the *model* (*BoardGame*), *view* (*GameGui*) and *controller* (*Place*) components are instantiated.

The GameGUI *subscribes itself as an Observer* to the game (observable), and subscribes a PlaceListener to MouseEvents for each Place on the view of the BoardGame.
Playing the game

Mouse clicks are propagated from a Place (controller) to the BoardGame (model):

If the corresponding move is valid, the model’s state changes, and the GameGUI updates the Place (view).
Checking user errors

> Assertion failures are generally a sign of errors in our program
  — However we cannot guarantee the user will respect our contracts!
  — We need special *always-on* assertions to check user errors

```java
public void move(int col, int row, Player p)
    throws InvalidMoveException
{
    assert this.notOver();
    assert p == currentPlayer();
    userAssert(this.get(col, row).isNobody(),
               "That square is occupied!");
...
}

private void userAssert(Boolean condition, String message)
    throws InvalidMoveException
{
    if (!$condition) {
        throw new InvalidMoveException(message);
    }
}
```
Refactoring the BoardGame

Adding a GUI to the game affects many classes. We iteratively introduce changes, and *rerun our tests after every change.*

> **Shift responsibilities** between BoardGame and Player (both should be passive!)
  > introduce Player interface, InactivePlayer and StreamPlayer classes
  > move `getRow()` and `getCol()` from BoardGame to Player
  > move `BoardGame.update()` to `GameDriver.playGame()`
  > change BoardGame to hold a matrix of Player, not marks

> **Introduce GUI classes** (GameGUI, Place, PlaceListener)
  > Introduce GUIplayer
  > PlaceListener triggers GUIplayer to move

> BoardGame must be **observable**
  > Introduce Move class to communicate changes from BoardGame to Observer

> Check user assertions!
Roadmap

> Model-View-Controller (MVC)
> Swing Components, Containers and Layout Managers
> Events and Listeners
> Observers and Observables
> Jar files, Ant and Javadoc
> Epilogue: distributing the game
Jar files

We would like to bundle the Java class files of our application into a single, executable file

— A *jar* is a Java Archive
— The *manifest* file specifies the main class to execute

```
Manifest-Version: 1.0
Main-Class: tictactoe.gui.GameGUI
```

*We could build the jar manually, but it would be better to automate the process …*

(http://java.sun.com/docs/books/tutorial/deployment/jar/)
Ant

Ant is a Java-based make-like utility that uses XML to specify dependencies and build rules.

You can specify in a “build.xml”:

- the *name* of a project
- the *default target* to create
- the *basedir* for the files of the project
- *dependencies* for each target
- *tasks* to execute to create targets
- You can extend ant with your own tasks
- Ant is included in eclipse

(Each task is run by an object that implements a particular Task interface.)

(http://ant.apache.org/manual/index.html)
A Typical build.xml

```xml
<project name="TicTacToeGUI" default="all" basedir="."/>
  <!-- set global properties for this build -->
  <property name="src" value="src"/>
  <property name="build" value="build"/>
  <property name="doc" value="doc"/>
  <property name="jar" value="TicTacToeGUI.jar"/>

  <target name="all" depends="jar,jdoc"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
    <copy todir="${build}/tictactoe/gui/images">
      <fileset dir="${src}/tictactoe/gui/images"/>
    </copy>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"
      source="1.5" target="1.5"
      classpath="junit.jar"/>
  </target>
```

...
<target name="jdoc" depends="init">
    <!-- Generate the javadoc -->
    <javadoc destdir="${doc}" source="1.5">
        <fileset dir="${src}" includes="**/*.java"/>
    </javadoc>
</target>

<target name="jar" depends="compile">
    <jar jarfile="${jar}"
        manifest="${src}/tictactoe/gui/manifest-run" basedir="${build}"/>
</target>

<target name="run" depends="jar">
    <java fork="true" jar="${jar}"/>
</target>

<target name="clean">
    <!-- Delete the ${build} directory -->
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
    <delete>
        <fileset dir="." includes="TicTacToeGUI.jar"/>
    </delete>
</target>
</project>
Running Ant

%% ant jar
Buildfile: build.xml
init:
    [mkdir] Created dir: /Scratch/P2-Examples/build
    [mkdir] Created dir: /Scratch/P2-Examples/doc
compile:
    [javac] Compiling 18 source files to /Scratch/P2-Examples/build
jar:
    [jar] Building jar: /Scratch/P2-Examples/TicTacToeGUI.jar
BUILD SUCCESSFUL
Total time: 5 seconds

Ant assumes that the build file is called build.xml
Javadoc generates API documentation in HTML format for specified Java source files.

- Each class, interface and each public or protected method may be preceded by “javadoc comments” between /** and */.
- Comments may contain special tag values (e.g., @author) and (some) HTML tags.
package p2.tictactoe;

/**
 * Minimal interface for Player classes that get moves from user
 * and forward them to the game.
 * @author $Author: oscar $
 * @version $Id: Player.java,v 1.5 2005/02/22 15:08:04 oscar Exp $
 */

class Player {
    /**
     * @return the char representation of this Player
     * @see AbstractBoardGame#toString
     */
    public char mark();
    ...
}
Javadoc output

public interface Player

Minimal interface for Player classes that get moves from user and forward them to the game.

Version:
SId: Player.java,v 1.5 2005/02/22 15:08:04 oscar Exp S
Author:
SAuthor: oscar S

Method Summary

<table>
<thead>
<tr>
<th>boolean</th>
<th>isNobody()</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>mark()</td>
</tr>
<tr>
<td>void</td>
<td>setGame(BoadGame game)</td>
</tr>
</tbody>
</table>

Let this player join a particular game.
GUI objects in practice ...

*Consider using Java webstart*
> Download whole applications in a secure way

*Consider other GUI frameworks (eg SWT from eclipse)*
> org.eclipse.swt.* provides a set of native (operating system specific) components that work the same on all platforms.

*Use a GUI builder*
> Interactively build your GUI rather than programming it — add the hooks later. (e.g. http://jgb.sourceforge.net/index.php)
Roadmap

- Model-View-Controller (MVC)
- Swing Components, Containers and Layout Managers
- Events and Listeners
- Observers and Observables
- Jar files, Ant and Javadoc
- Epilogue: distributing the game
A Networked TicTacToe?

We now have a usable GUI for our game, but it still supports only a single user.

We would like to support:

- players on *separate machines*
- each running the game GUI locally
- with a remote "*game server*" managing the state of the game
The concept
Remote Method Invocation

RMI allows an application to register a Java object under a public name with an RMI registry on the server machine.

A client may look up the service using the public name, and obtain a local object (stub) that acts as a proxy for the remote server object (represented by a skeleton).
Playing the game

- :GameObserver
- :GameGUI
- :Place
- :PlaceListener
- :GameProxy
- :Gomoku
- :PassivePlayer
- :WrappedObserver

Actions:
- 1.1a: mouseClicked(): :PlaceListener
- 1.1d: update(): :GameGUI
- 1.1.1d: setMove(): :Place
- 1.2.1b: move(): :PassivePlayer
- 1.2.1b: move(): :Gomoku
- 1.2.1b: update(): :WrappedObserver

Click event flow:
- User click
- :GameProxy
- :Place
- :WrapedObserver
- :GameGUI
- :GameObserver

Sequence:
- 1a: mouseClicked() → 1b: move() → 1c: update() → 1d: update()
What you should know!

- The TicTacToe game knows nothing about the GameGUI or Places. How is this achieved? Why is this a good thing?
- What are models, view and controllers?
- What is a Container, Component?
- What does a layout manager do?
- What are events and listeners? Who publishes and who subscribes to events?
- How does the Observer Pattern work?
- Ant
- javadoc
Can you answer these questions?

How could you make the game start up in a new Window?

What is the difference between an event listener and an observer?

The Move class has public instance variables — isn’t this a bad idea?

What kind of tests would you write for the GUI code?