# GUI construction

Alexandre Bergel
http://bergel.eu
17-11-2021

# Source

David Flanagan, Java Foundation Classes in a Nutshell, O'Reilly

http://java.sun.com/docs/books/tutorial/uiswing

http://eclipse.org

# Roadmap

1. Model-View-Controller (MVC)
2. AWT & Swing Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. SWT: The Standard Widget Toolkit

# Roadmap

1. **Model-View-Controller (MVC)**
2. AWT & Swing Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. SWT: The Standard Widget Toolkit

# A Graphical TicTacToe?

Our existing TicTacToe implementation is very limited:

- Textual input
- Textual display

We would like to migrate it towards an interactive game:

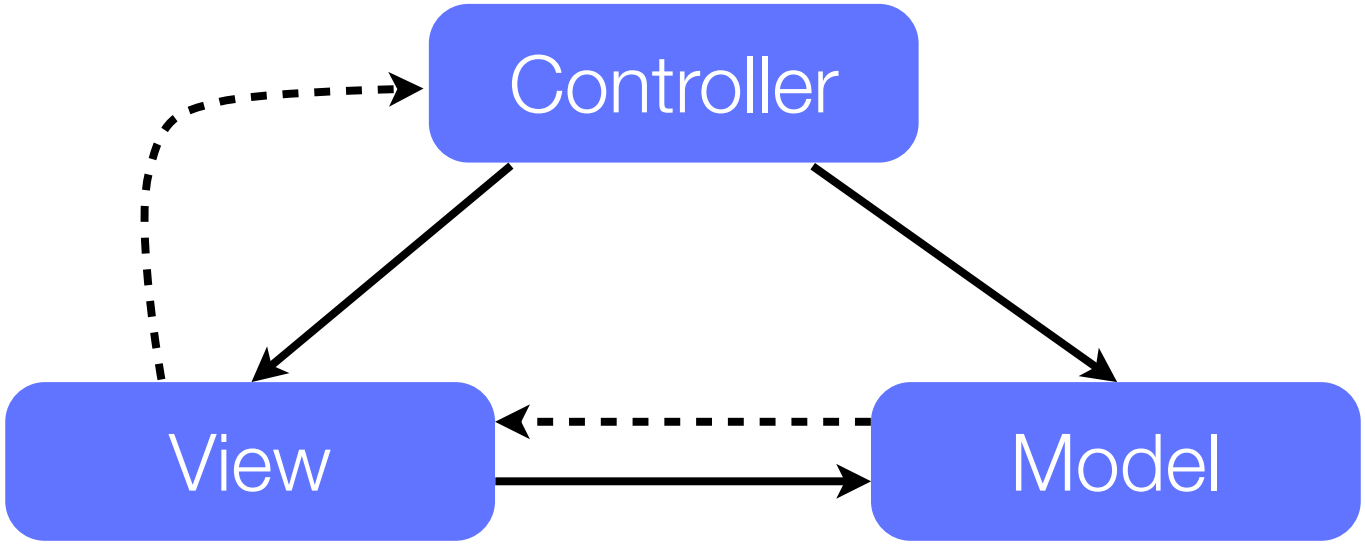
- running the game with graphical display and mouse input

# Model-View-Controller

Version 6 of our game implements *a model of the game*, without a GUI

The GameGUI class will implement *a graphical view* and a *controller for GUI events*

# Model-View-Controller

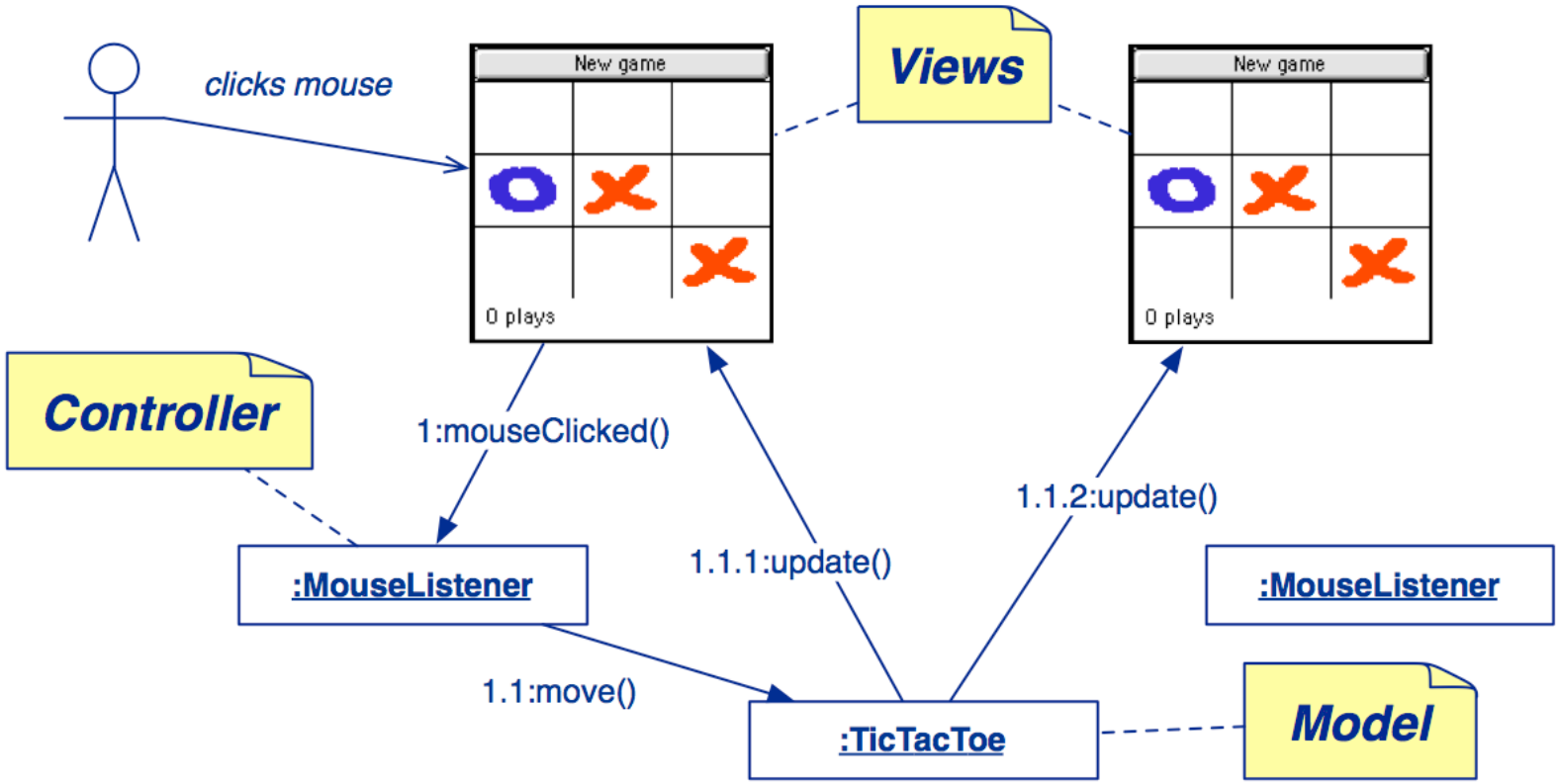



direct association

indirect association via an observer

Model-View-Controller is a software architecture. The MVC pattern separates an application from its GUI so that multiple views can be dynamically connected and updated.

# Model-View-Controller

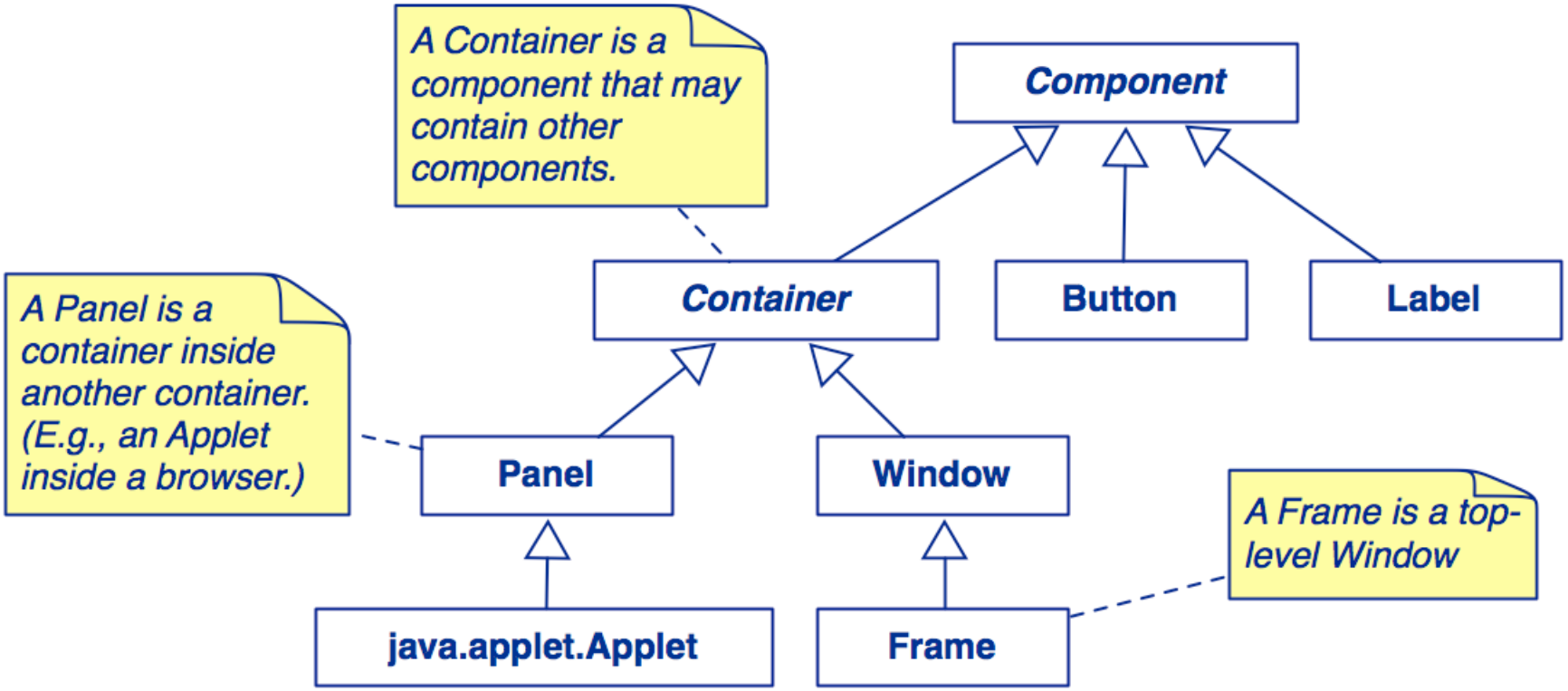


# Roadmap

1. Model-View-Controller (MVC)
2. **AWT & Swing Components, Containers and Layout Managers**
3. Events and Listeners
4. Observers and Observables
5. SWT: The Standard Widget Toolkit

# AWT Components and Containers

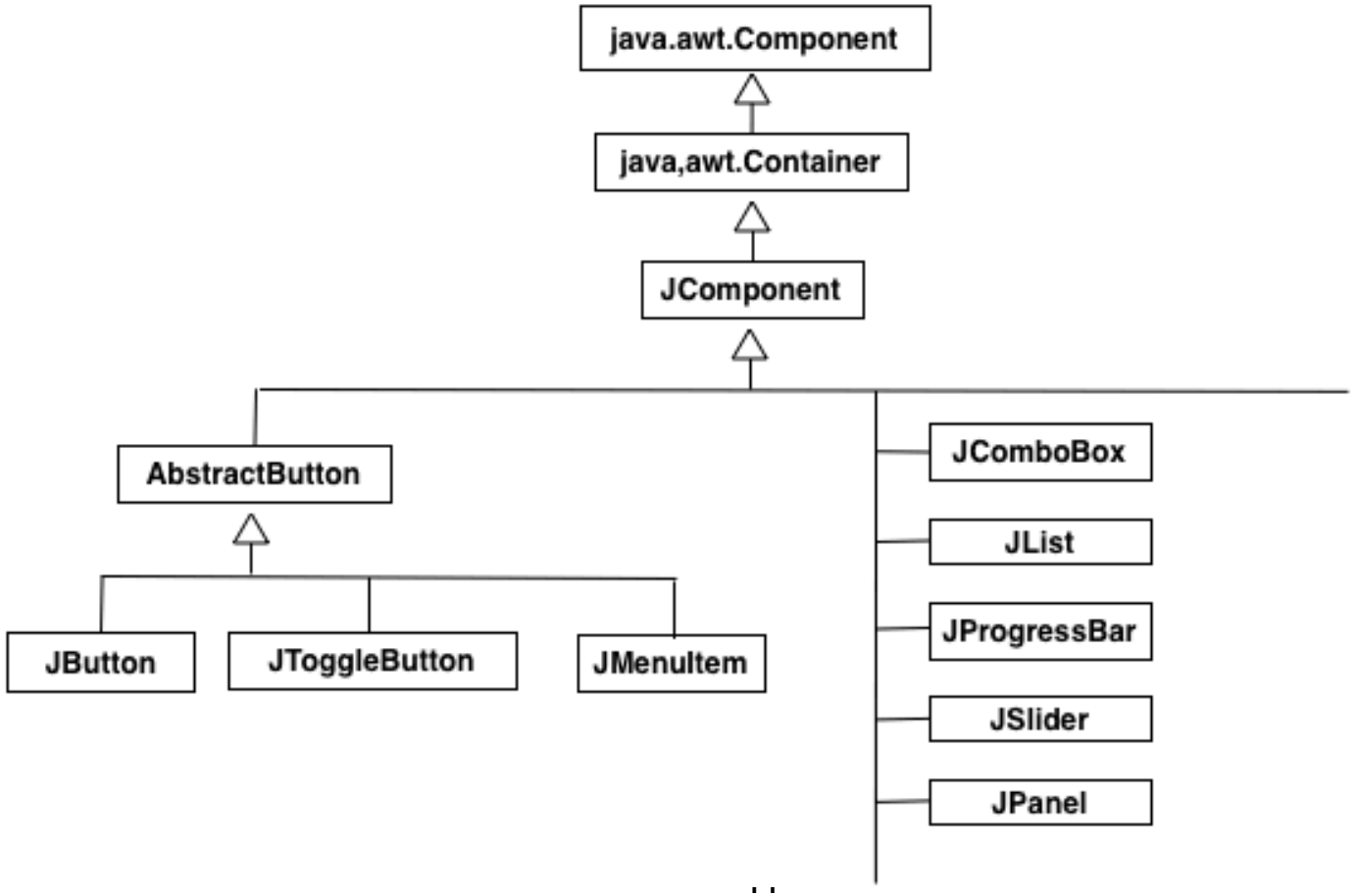
The java.awt package defines GUI components, containers and their layout managers.




There are also many graphics classes to define colors, fonts, images etc.

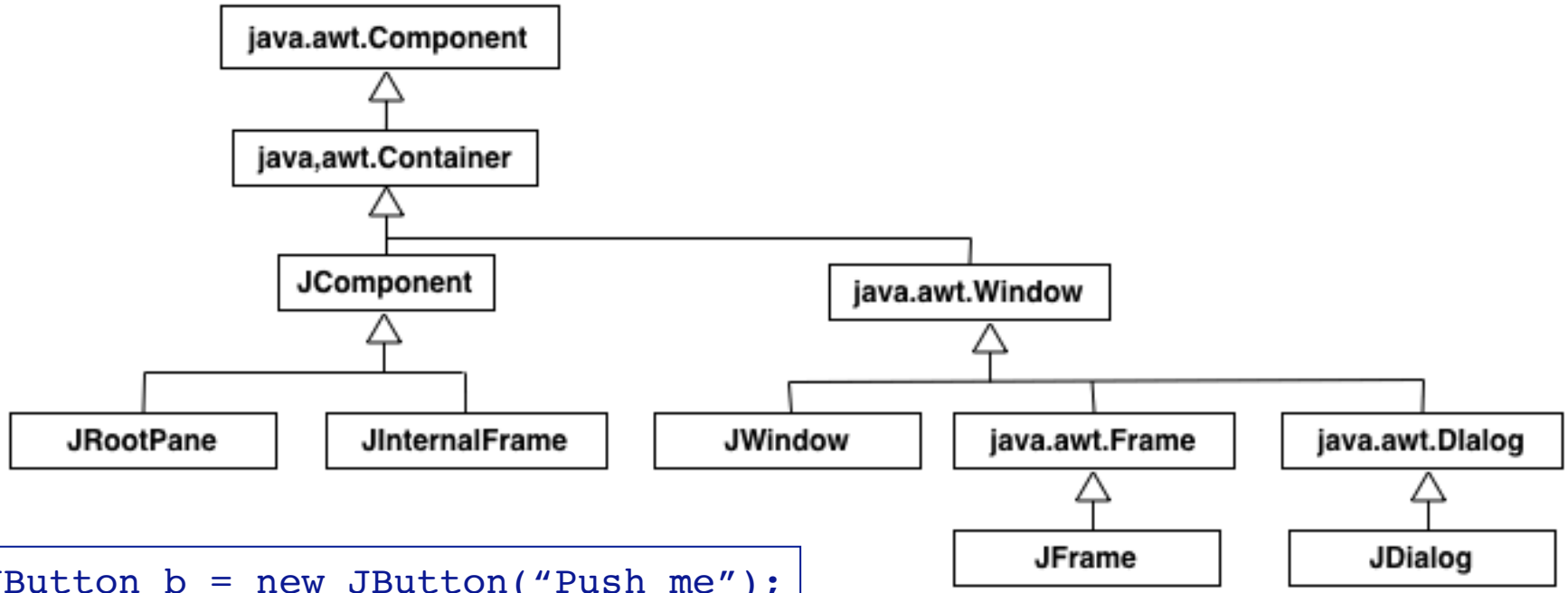
# Swing JComponents

The javax.swing package defines sophisticated GUI components



# Swing Containers and Containment

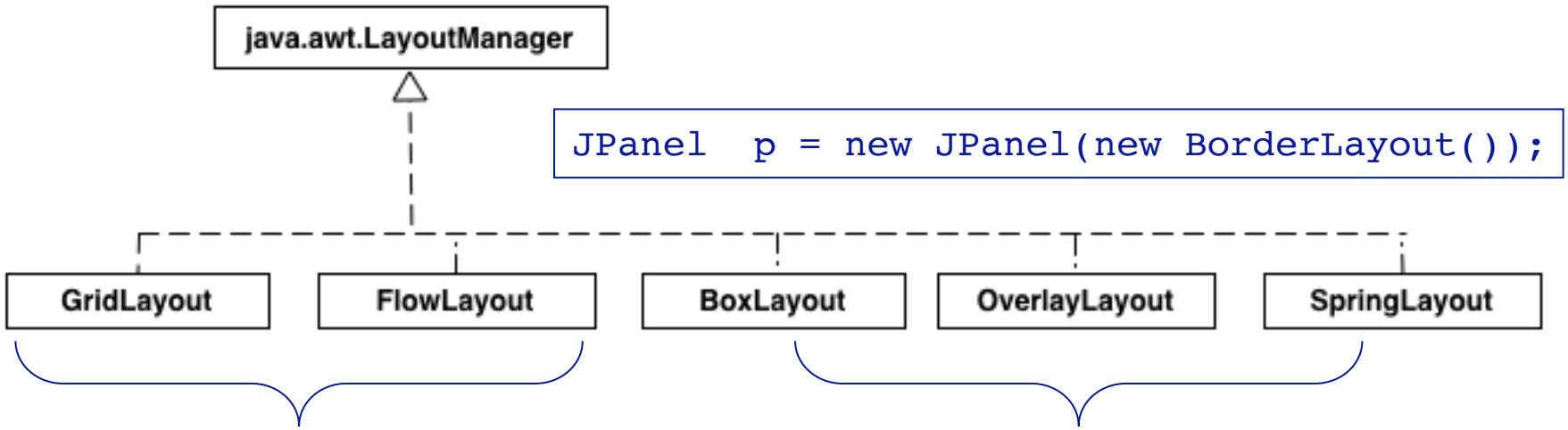
Swing Containers may contain other Components




```
JButton b = new JButton("Push me");
JPanel  p = new Jpanel();
p.add(b);
```

# Layout Management

The *Layout Manager* defines how the components are arranged in a container (size and position).




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

awt LayoutManagers

swing LayoutManagers

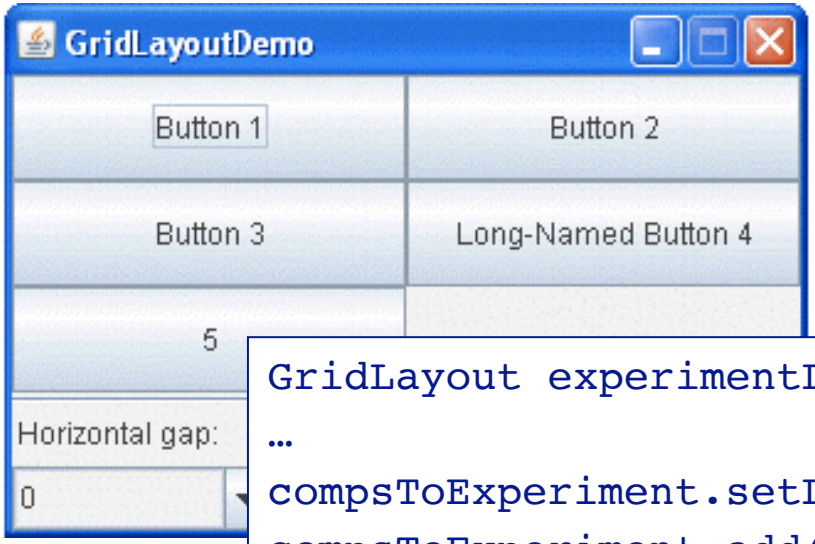
```
Container contentPane = frame.getContentPane();
contentPane.setLayout(new FlowLayout());
```

https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

# An example: GridLayout

A GridLayout places components in a grid of cells:

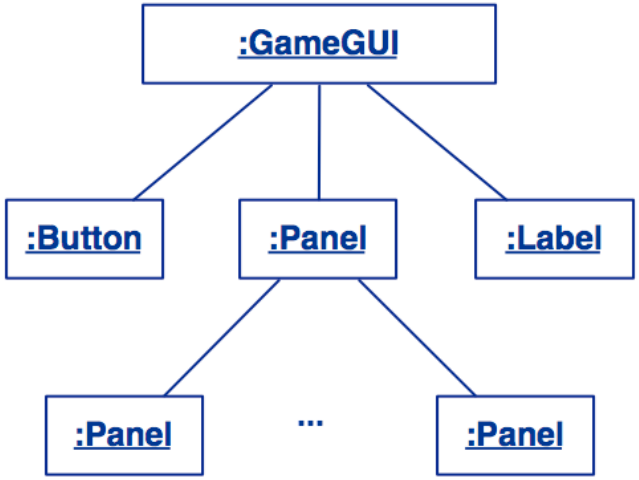
Each component takes up all the space in a cell.




```
GridLayout experimentLayout = new GridLayout(0,2);
…
compsToExperiment.setLayout(experimentLayout);
compsToExperiment.add(new JButton(“Button 1”);
compsToExperiment.add(new JButton(“Button 2”);
```

# The GameGUI

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.

# Laying out the GameGUI

```
public 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 ...

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

# Roadmap

1. Model-View-Controller (MVC)
2. AWT & Swing Components, Containers and Layout Managers
3. **Events and Listeners**
4. Observers and Observables
5. SWT: The Standard Widget Toolkit

# Interactivity with Events

To make your GUI do something you need to *handle* events

An event is *typically a user action* - mouse click, key stroke, etc

Java Event model is used by Java AWT and Swing (java.awt.AWTEvent and javax.swing.event)

# Concurrency and Swing

The program is always responsive to *user interaction*, no matter what it is doing

The runtime of the Swing framework creates threads

- you do not explicitly create them
- remember the difference between a framework and a library?

The *Event Dispatch* thread is responsible for *event handling*

# Events and Listeners (I)

Instead of actively checking for GUI events, you can define *callback methods* that will be invoked when your GUI objects receive events:

*Hardware events ...*
(`MouseEvent`, `KeyEvent`, ...)




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

http://docs.oracle.com/javase/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)

| *Component* | *Events* | *Listener Interface* | *Listener methods* |
|---|---|---|---|
| **JButton** | **ActionEvent** | *ActionListener* | actionPerformed() |
| **JComponent** | **MouseEvent** | *MouseListener* | mouseClicked() |
| | | | mouseEntered() |
| | | | mouseExited() |
| | | | mousePressed() |
| | | | mouseReleased() |
| | | *MouseMotionListener* | mouseDragged() |
| | | | mouseMoved() |
| | **KeyEvent** | *KeyListener* | keyPressed() |
| | | | keyReleased() |
| | | | keyTyped() |
| ... | | | |

# Listening for Button events

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

```
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
      }                          // of enclosing class!
   });
   return again;
}
```

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

# Listening for Button events

When we create the "New… *…ch an ActionListener* with… method:

Instance an unnamed subclass of ActionListener()

```
private Component makeControl…
   Button again = new Button("N…
   again.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         showFeedBack("starting new game ...");
         newGame();              // NB: has access to methods
      }                          // of enclosing class!
   });
   return again;
}
```

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

# About inner classes

Inner classes are useful when you want to punctually create a particular objects

Consider the following example on creating operations

```
public interface Operation {
    int apply(Counter counter);
}
```

```
public class Counter {
    private int value;
    public Counter (int value) { this.value = value; }

    public int getValue() {
        return value;
    }

    public int apply(Operation op) {
        return op.apply(this);
    }

    public static void main(String[] argv) {
        Counter c = new Counter(10);
        System.out.println(c.apply(new Operation() {
            public int apply(Counter c) {
                return c.getValue() * 10;
            }
        }));
    }
}
```

# Nested classes in Java

A nested class is a class defined within another class

Nested classes are divided into two categories: static and non static

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

# Nested classes in Java

```
class OuterClass {
    static class StaticNestedClass {
    }
    class InnerClass {
    }
}

OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

The class OuterClass does not need to be instantiated.
StaticNestedClass can access any static fields and methods from OuterClass.

# Nested classes in Java

```
class OuterClass {
    static class StaticNestedClass {
    }
    class InnerClass {
    }
}

OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject =
                              outerObject.new InnerClass();
```

An instance of OuterClass is needed to instantiate InnerClass. InnerClass has direct access to the methods and fields of its enclosing instance.

# Why Use Nested Classes?

It is a way of logically grouping classes that are only used in one place

It increases encapsulation

It can lead to more readable code

# Gracefully cleaning up

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

```
public class GameUI extends JFrame implements Observer {
   …
   public GameUI(String title) throws HeadlessException {
      …
      this.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            GameUI.this.dispose();
            // NB: disambiguate "this"!
         }
      });
      this.show();
   }
```

# Gracefully cleaning up

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

```
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();
   }
```

Release all native screen resources

# Gracefully cleaning up

thrown when no display is available (e.g., on a server)

```
public class GameUI extends JFrame implements Observer {
   …
   public GameUI(String title) throws HeadlessException {
      …
      this.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            GameUI.this.dispose();
            // NB: disambiguate "this"!
         }
      });
      this.show();
   }
```

# Listening for mouse clicks

We also attach a MouseListener to each Place on the board

```
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

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

```
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:*

```
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

1. Model-View-Controller (MVC)
2. AWT & Swing Components, Containers and Layout Managers
3. Events and Listeners
4. **Observers and Observables**
5. SWT: The Standard Widget Toolkit

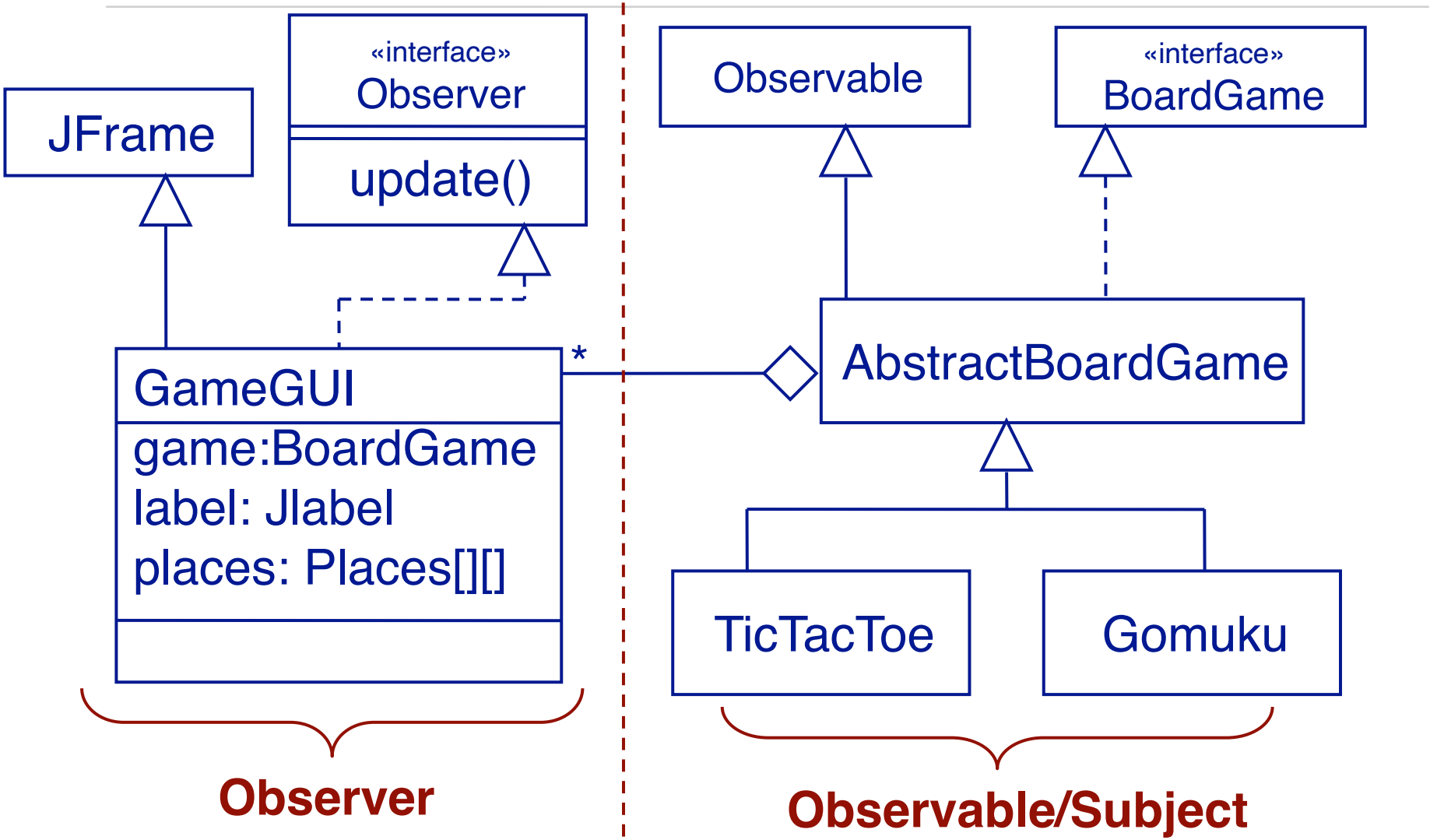
# The Observer Pattern (remember?)

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

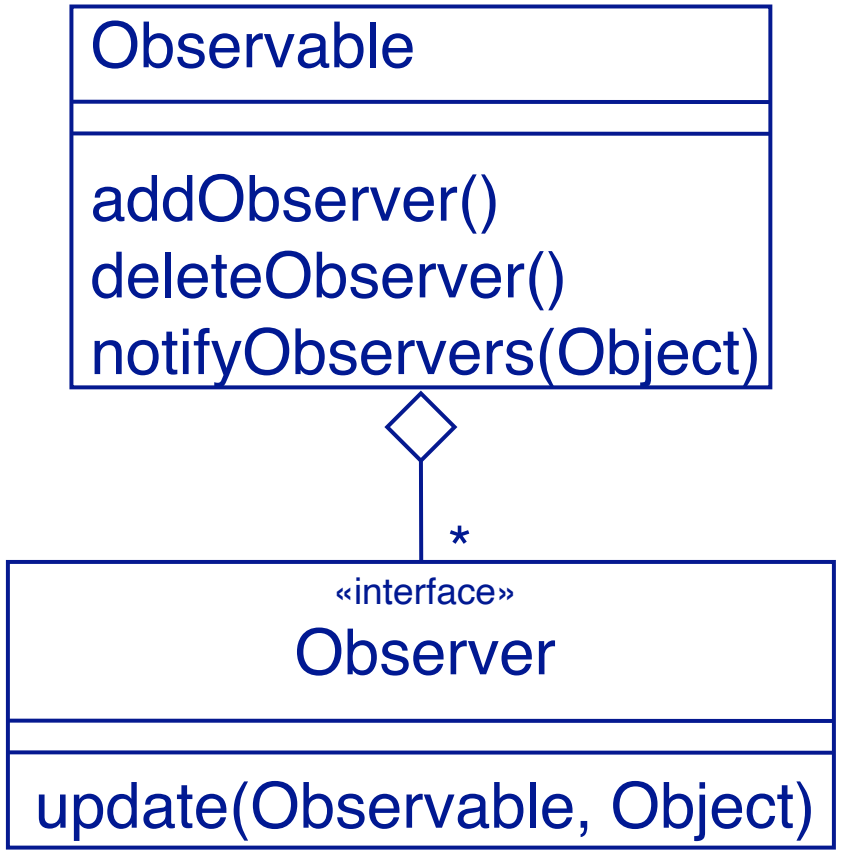


# Observers and Observables

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.



# Adding Observers to the Observable

```
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:

```
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):

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

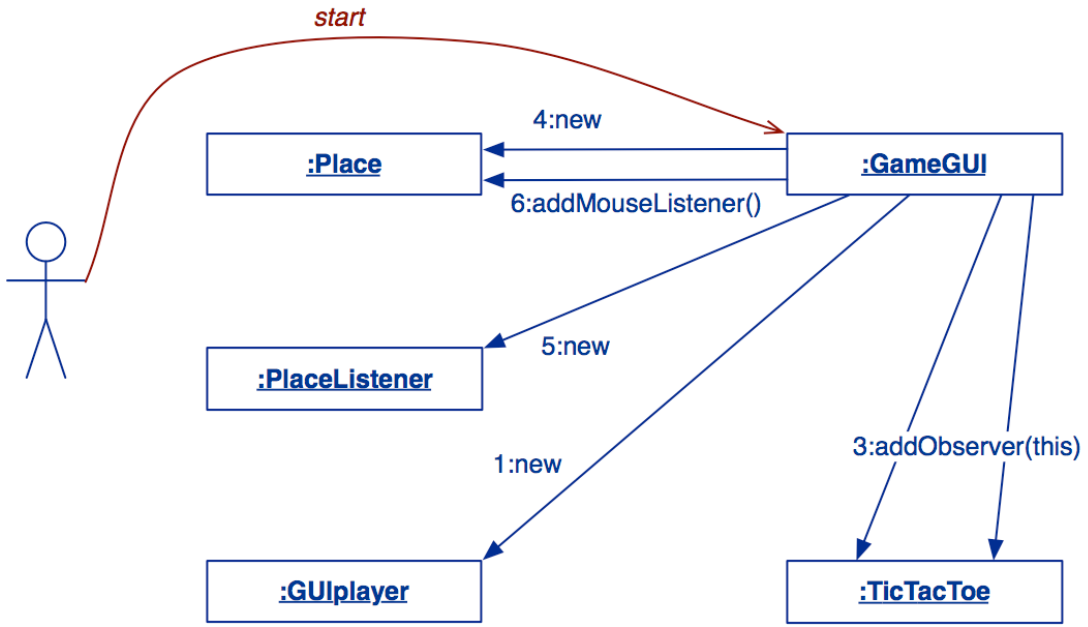
# Handy way of Communicating changes

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

```
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

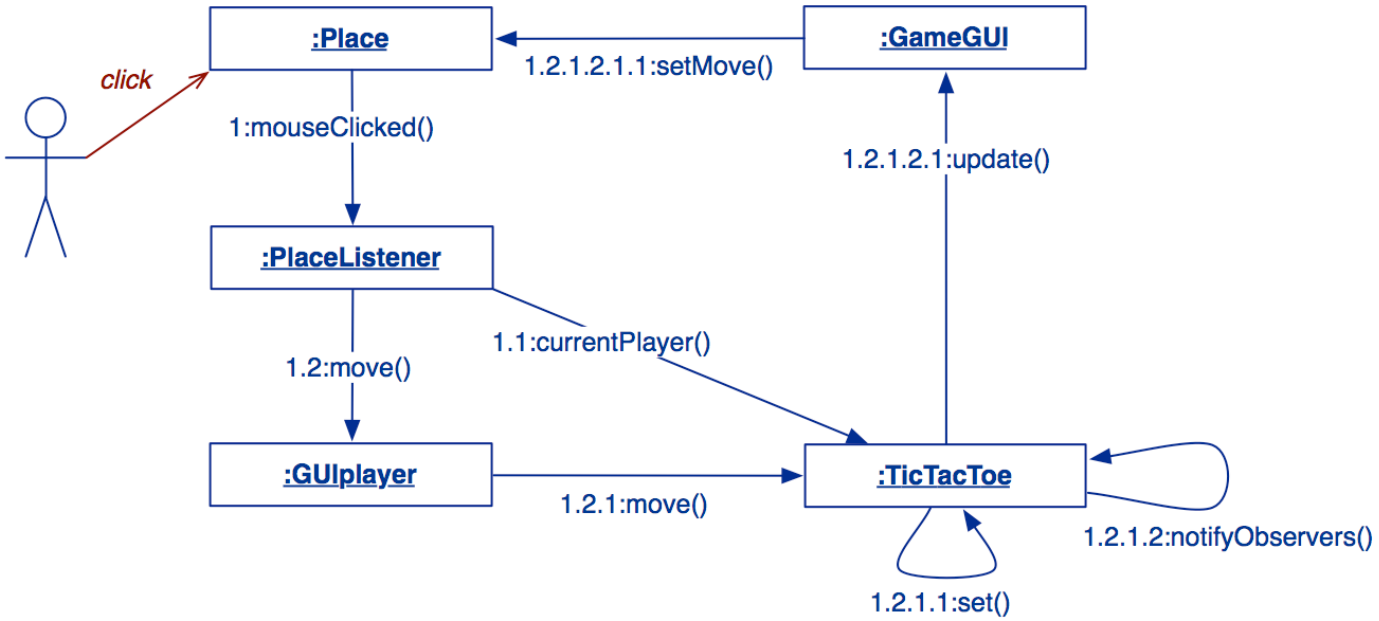
When the GameGUI is created, the *model* (*BoardGame*), *view* (*GameGui*) and *controller* (*Place*) components are instantiated




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

```
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

# Refactoring the BoardGame

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

1. Model-View-Controller (MVC)
2. AWT & Swing Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. **SWT: The Standard Widget Toolkit**

# AWT & Swing

There are many existing libraries for doing GUI in Java

In 1995, Java 1.0 was released with AWT

- AWT is very limited: few widgets are supported, no look and feel

In 1996, Swing was released

- Swing has a better event mechanism, look and feel
- Again, very limited: unclear architecture as it is built on top of AWT, and more importantly, widgets are not native and said to be slow

Both are still supported in the last version of Java

However, Oracle is making JavaFX standard

# SWT

Standard Widget Toolkit (SWT) is a competing toolkit originally developed by IBM and now maintained by the Eclipse community

If you need to program an Eclipse plugin, then you probably need SWT

# JavaFX

JavaFX was said to be the a new mainstream UI framework for Java

But is now an external Java project

An application made with JavaFX looks like native

JavaFX can also export a UI to the web

http://docs.oracle.com/javafx/2/webview/jfxpub-webview.htm

JavaFX is nicely integrated with OpenGL

# What you should know!

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?

What Ant, javadoc are for?

The TicTacToe game knows nothing about the GameGUI or Places. How is this achieved? Why is this a good thing?

# Can you answer to 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?

# License

http://creativecommons.org/licenses/by-sa/2.5

creative commons

COMMONS DEED

**Attribution-ShareAlike 2.5**

**You are free:**

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

**Under the following conditions:**

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

**Your fair use and other rights are in no way affected by the above.**