UNIVERSITÄT BERN

8. GUI Construction

Oscar Nierstrasz

GUI Construction

Sources

- David Flanagan, *Java in a Nutshell: 5th edition*, O'Reilly.
- David Flanagan, Java Foundation Classes in a Nutshell, O'Reilly
- http://java.sun.com/docs/books/tutorial/uiswing

- 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

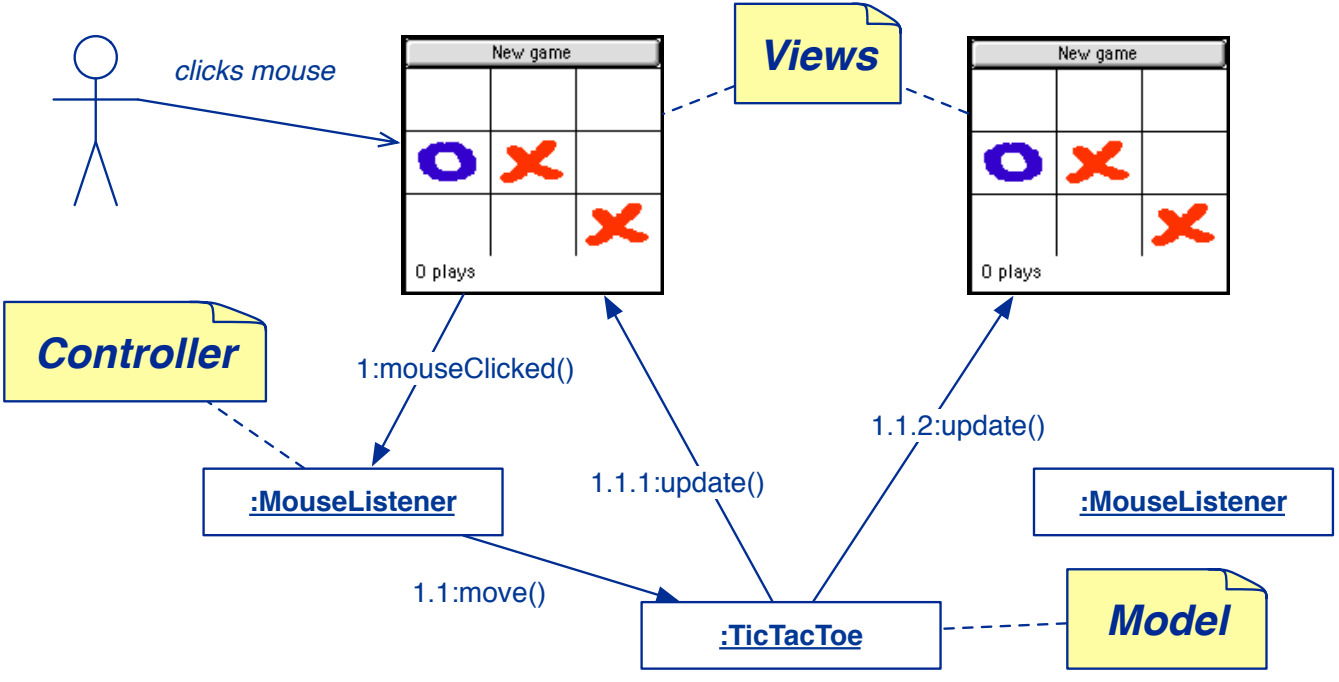
> textual input and display

We would like to migrate it towards an interactive game:

> running the game with *graphical display and mouse input*

Model-View-Controller

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.

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts:

- the *model*, responsible for the domain logic
- the *view*, responsible for the graphical display, and
- the *controller*, responsible for synchronizing the two.

MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

Input ⟶ Processing ⟶ Output

Controller ⟶ Model ⟶ View

In the diagram, a user clicks on a view of the game. This generates a `mouseClicked()` event, which is handled by the controller, a `MouseListener` object. This object interprets the mouse click by invoking `move()` on the model, a `TicTacToe` instance.

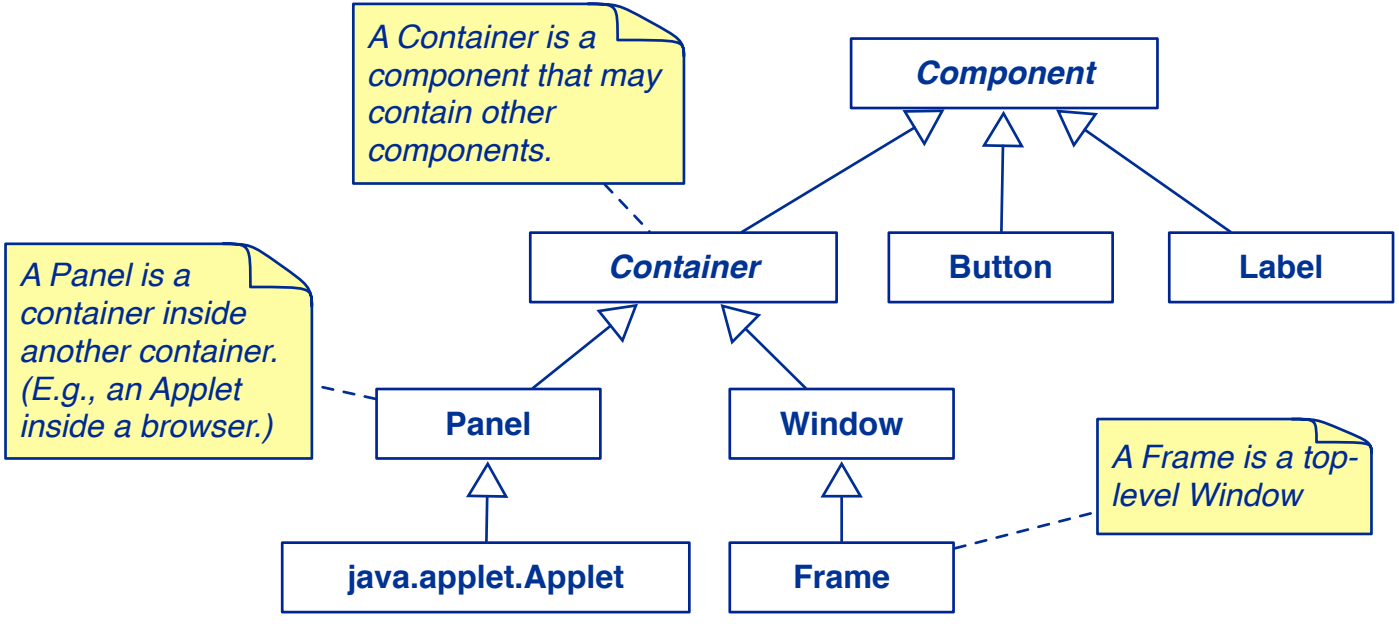
When the model changes state, it uses the *Publish/Subscribe* design pattern infrastructure to inform the views. These then update themselves to reflect the updated state.

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

AWT Components and Containers

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



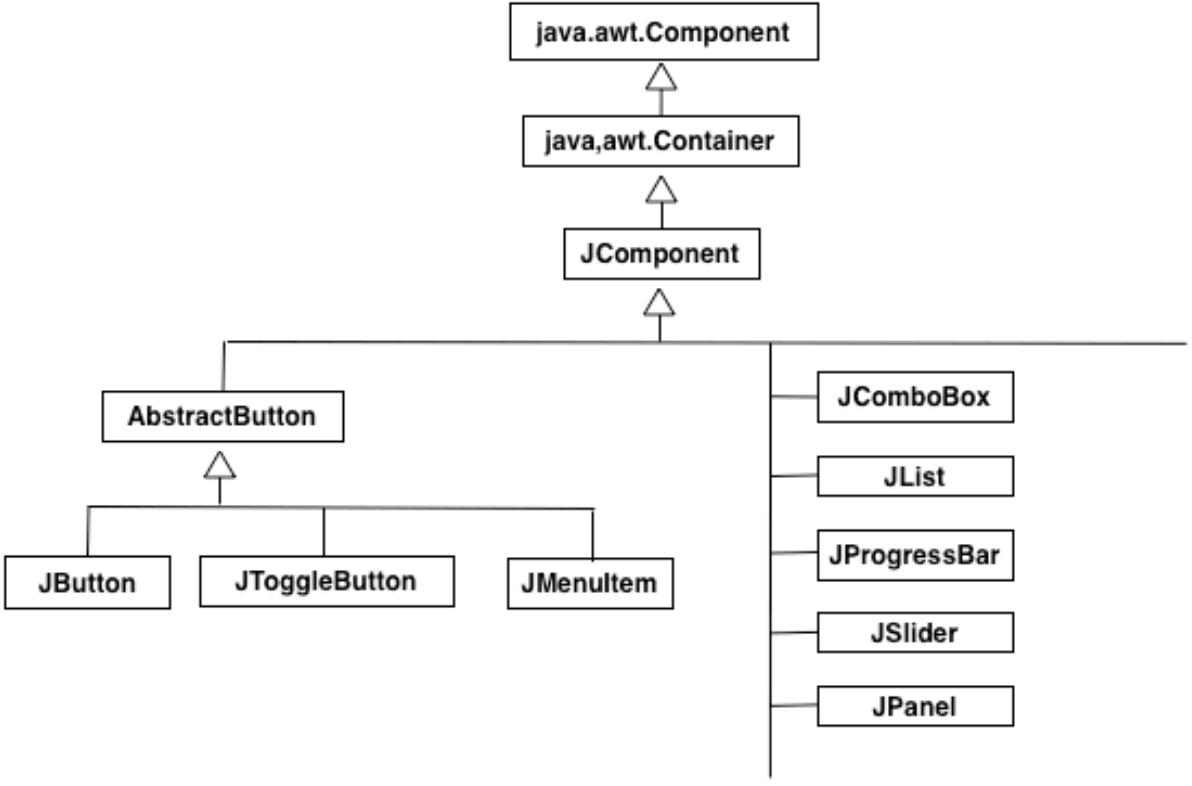
NB: There are also many graphics classes to define colours, fonts, images etc.

The Abstract Windowing Toolkit (AWT) provides basic facilities for creating graphical user interfaces (GUIs) and also for drawing graphics. The GUI features are layered on top of the native GUI system of the underlying platform. In other words if you create a graphical button, you will get a Windows button or a Mac button depending on the platform on which the application is running.

The root of the hierarchy is `Component` (every AWT component *is a* `Component`). Primitive components like `Button` or `Label` have no subparts. Composite components are all subclasses of `Container`.

Swing JComponents

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



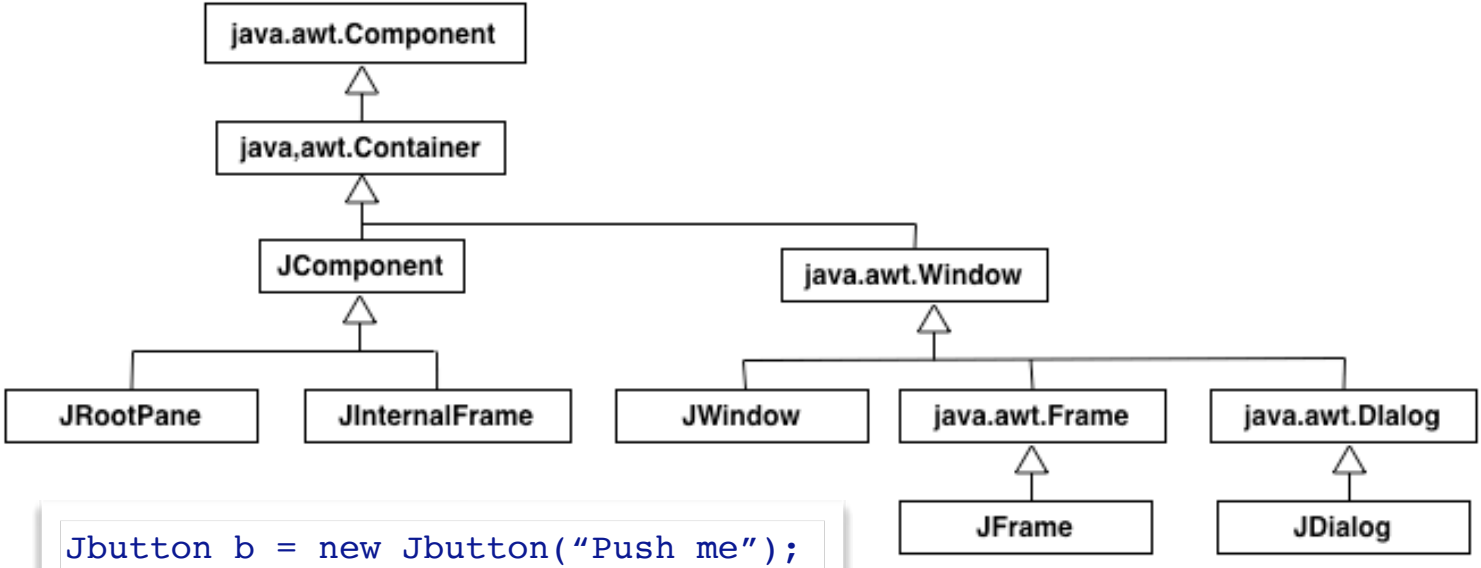
Swing is a newer GUI toolkit, an extension to AWT. GUIs built with Swing are meant to automatically emulate the native "look and feel" of the underlying platform, so instead of looking like a "Java app", they look like native Mac or Windows applications.

Swing components are "lightweight", in that they do not rely on underlying platform.

The root of the Swing hierarchy is `JComponent`. Since for technical reasons, `JComponent` inherits from `java.awt.Container`, this leads to a somewhat messy hierarchy — note that although `JButton` indirectly inherits from `Container`, it is not a composite (Swing) object.

Swing Containers and Containment

Swing Containers may contain other Components




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

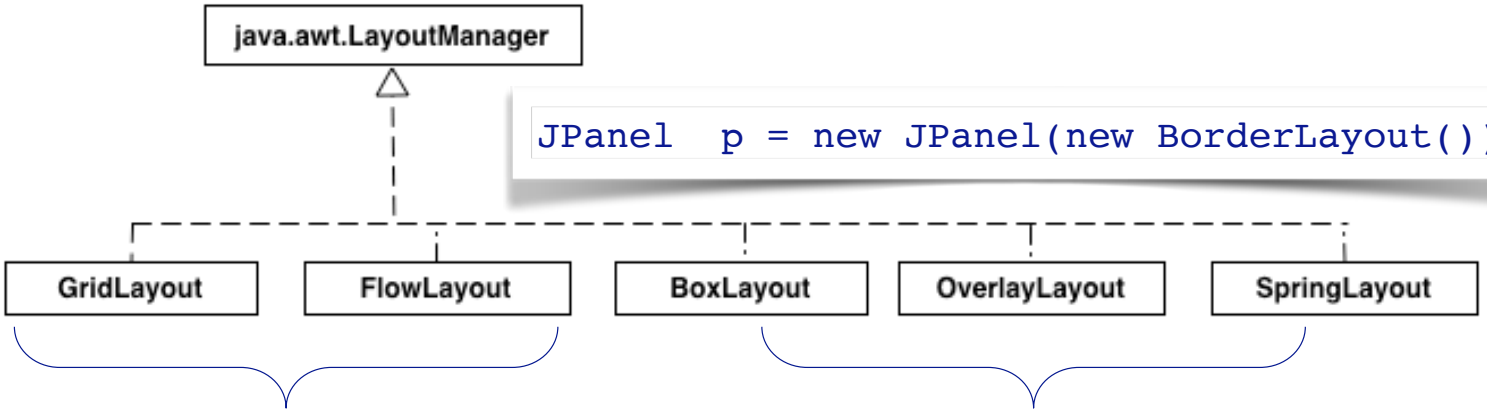
In order to create a GUI, components must be arranged inside a container. A container is a component that can contain other components. Main application windows, dialog boxes are commonly used containers.

When building a GUI you must create your components, create the containers that will hold the components and then add the components to the containers.

Top level containers `JFrame`, `JDialog`, `JWindow` do not inherit from `JComponent`. Instead they create a child `JRootPane` to hold the components.

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());
```

http://docs.oracle.com/javase/tutorial/uiswing/layout/using.html

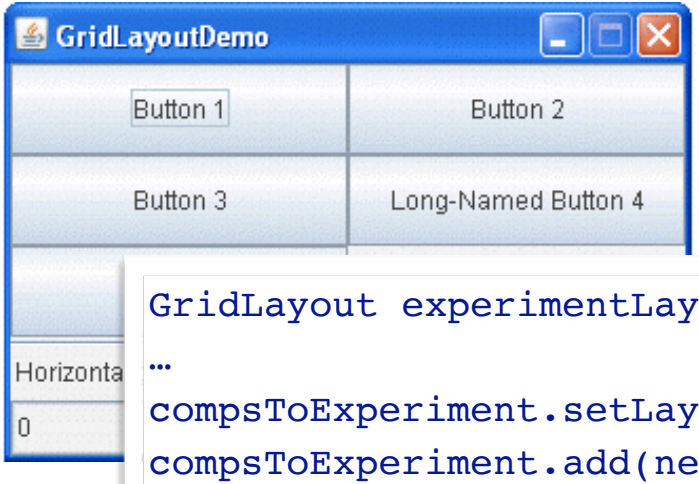
A layout manager is an object that implements the layout manager interface and determines the size and position of the components. Although the components can provide size and alignment hints, a container has the final say on the size and position of the components.

Some containers such as `JTabbedPane` and `JSplitPane` define a particular arrangement for their children. Others don't. When working with containers you must specify a `LayoutManager` object to arrange the children within a container.

An example: GridLayout

A GridLayout places components in a grid of cells.

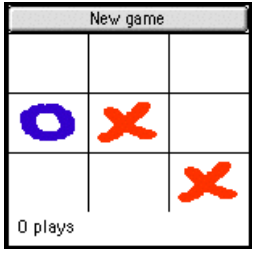
- Each component takes up all the space in a cell.
- Each cell is the same size

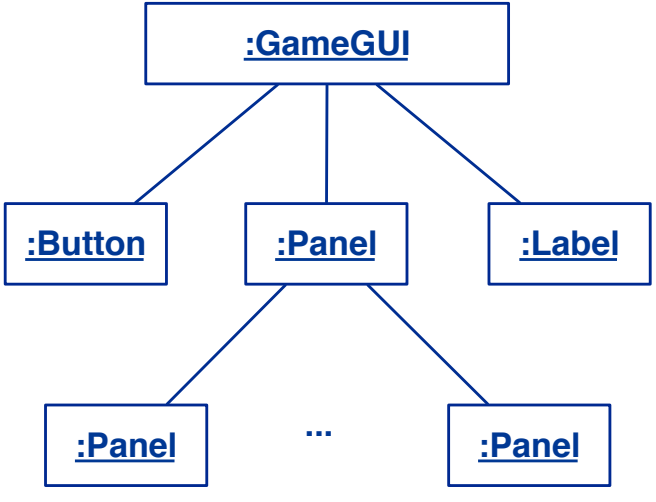



```
GridLayout experimentLayout = new GridLayout(0,2);
…
compsToExperiment.setLayout(experimentLayout);
compsToExperiment.add(new Jbutton("Button 1"));
compsToExperiment.add(new Jbutton("Button 2"));
```

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`.

NB: *GameGUI is the only class that differs substantially for AWT & Swing*

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();
   }
```

TicTacToe v7-swing

A class can implement the `Observer` interface when it wants to be informed of changes in `Observable` objects. We'll see more on this very shortly …

Aside: A `HeadlessException` is thrown when code that is dependent on a keyboard, display, or mouse is called in an environment that does not support a keyboard, display, or mouse.

Helper methods

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

```
protected 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`)

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 don't explicitly create them.
- The *Event Dispatch thread* is responsible for *event handling*.

Every program has a set of threads where the application logic begins. In standard programs there is just one thread — it invokes the main method of some “main” class.

In Swing programs the initial threads do not have a lot to do. Once the GUI is created the application is driven by the GUI events, each of which causes the execution of a short task on the event dispatch thread.

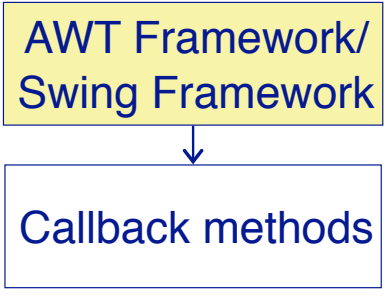
Each component registers event listener methods for events that component is interested in.

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`, ...)

... are handled by subscribed `Listener` objects



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

http://docs.oracle.com/javase/tutorial/uiswing/events/index.html

In procedural languages like C, *callbacks* are *pointers to handler functions* that will be called when the event is raised. In an object-oriented language, a callback is *an object that implements a given handler interface*. The technology may be different, but the underlying idea is the same: *when an event is raised, all registered handlers are invoked with arguments that specify the raised event*.

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()
...			

Every kind of event is associated with a particular listener interface. The listener (i.e., the callback object or event handler) simply implements the listener methods that it wants to react on.

For example, a `MouseListener` may decide to implement `mouseClicked()` while ignoring all other mouse events, i.e., by implementing empty listener methods for `mouseEntered()` etc.

Listening for Button events

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

```
protected 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`.

An *inner class* is simply a class nested inside another class.

An *anonymous inner class* is a nested class that has no name. It is defined at the precise point where the class is instantiated, by declaring a new instance of the parent class (or interface), and supplying in curly braces after the name of the parent the implementation of the anonymous subclass.

This feature is most often used when there is only *a single method to implement*. Note that an instance of an anonymous inner class has full *access to the scope of its enclosing object*, including all methods and instance variables (even private ones).

Aside: Java 8 supports *lambdas* (anonymous functions), which can be more concise that anonymous inner classes in some cases.

Anonymous functions in Java 8

Since an ActionListener is “just a function” (only one method), we can replace it in Java 8 with a so-called “lambda”:

```
protected Component makeControls() {
   Button again = new Button("New game");
   button.addActionListener(e -> (new
GameGUI(tictactoeFactory.getGame())).setVisible(true));
   return again;
}
```

A “lambda” is an “anonymous function” that takes an argument (e), and returns a value (following the arrow).

An “anonymous function” or *lambda expression* is a nameless function that is defined in-place. For example: `x -> 1` is the function that takes an argument `x` and returns `1`. `x->x` is the identify function that returns its argument `x`.

Anonymous functions are best known from so-called functional programming languages. Java 8 introduced them to reduce the complexity of anonymous inner classes in common cases where a simple function is needed (also to catch up to Scala, a language based on Java that already had lambdas).

See:

https://en.wikipedia.org/wiki/Anonymous_function#Java

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();
   }
```

The `WindowListener` interface requires seven callback methods to be implemented. If you are only interested in one of these seven events, instead of implementing six empty methods in addition to the one you really want, you can simply inherit from `WindowAdapter`, which provides seven empty methods, and override the one you want. This is especially handy when defining a very simply anonymous inner class, as is the case here.

We can also see another curious problem with inner classes: since we have one object nested within another, the inner `this` hides (or "shadows") the outer `this`. We explicitly refer to the outer object by prepending `this` with the name of its class (`GameGUI`).

Listening for mouse clicks

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

```
protected 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;
}
```

Since `PlaceListener` has an implementation that is longer than a couple of lines, we implement it as a proper class, rather than an anonymous inner class.

The PlaceListener

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

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

The PlaceListener ...

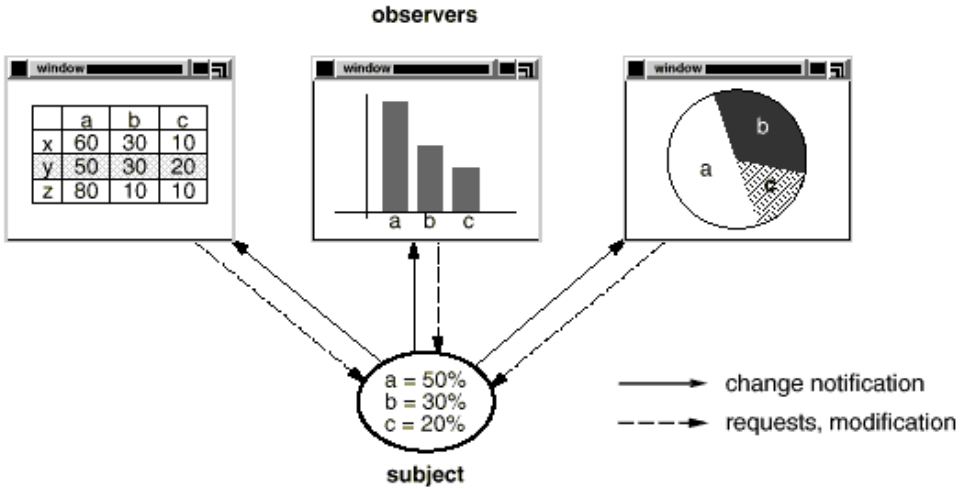
```
public void mouseClicked(MouseEvent e) {
    …
    if (game.notOver()) {
        try {
            ((GUIplayer) game.currentPlayer()).move(col,row);
            gui.showFeedBack(game.currentPlayer().mark() + " plays");
        } catch (AssertionError err) {
            gui.showFeedBack(err.getMessage());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

We only have to override the `mouseClicked()` method:

Roadmap

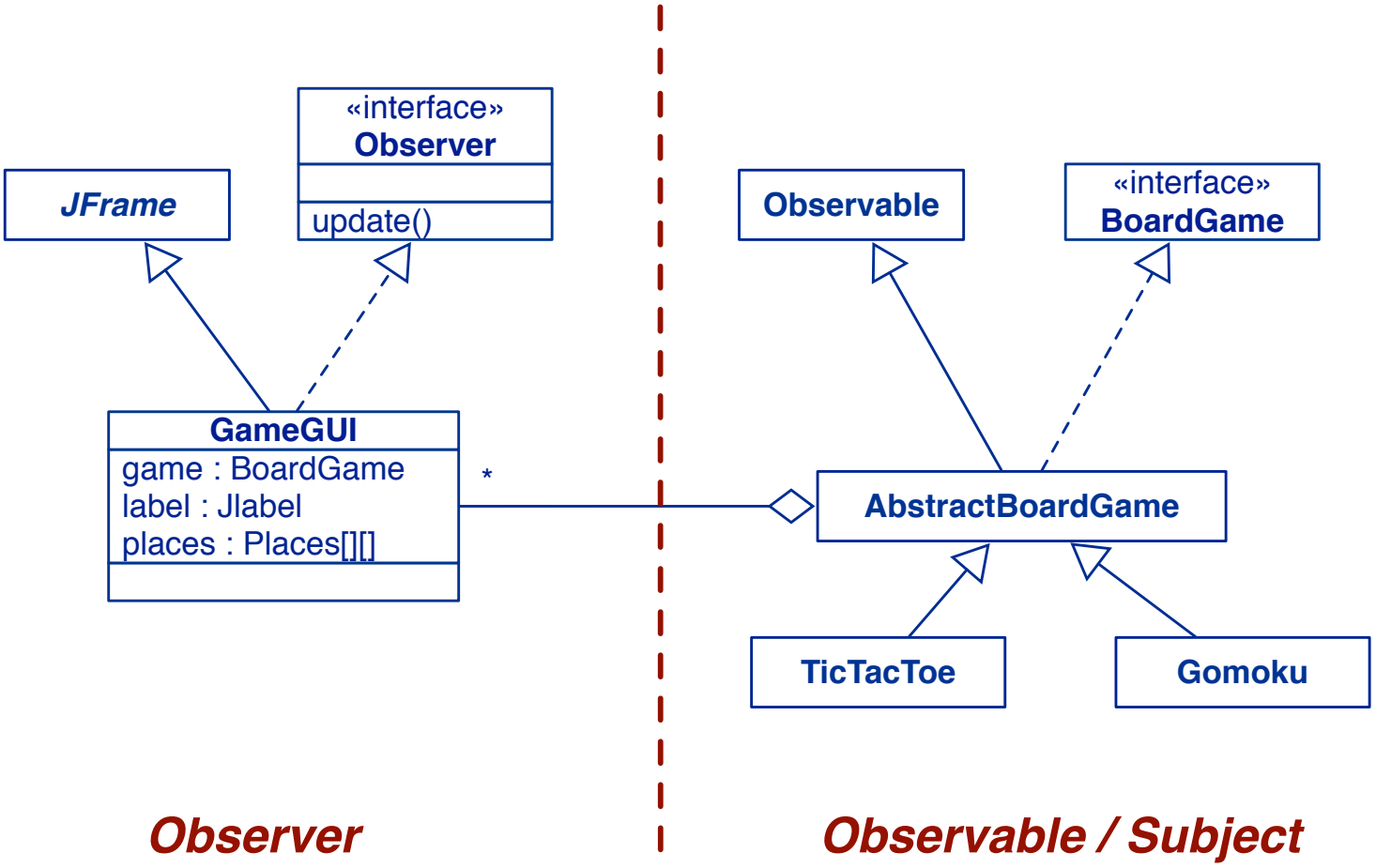
- Model-View-Controller (MVC)
- Swing Components, Containers and Layout Managers
- Events and Listeners
- **Observers and Observables**
- Jar files, Ant and Javadoc
- Epilogue: distributing the game

The Observer Pattern



- Also known as the *publish/subscribe* design pattern — to observe the state of an object in a program.
- *Observers* registered to *observe* state changes in an observable object, known as the *subject*.
- The *subject* maintains a set of *observers* to notify whenever its state changes.

Our BoardGame Implementation



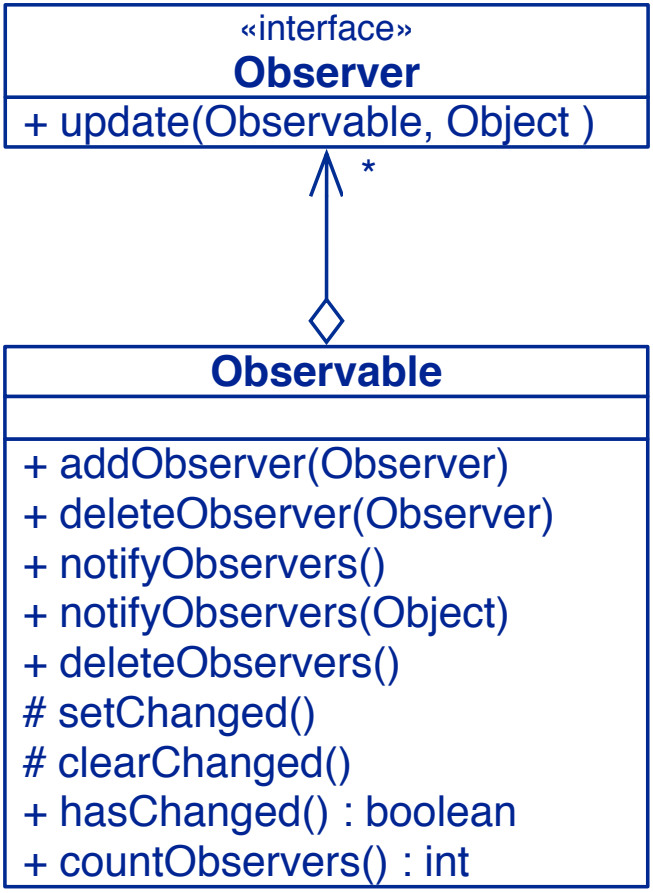
The `TicTacToe` game maintains a list of Observers to notify whenever its state changes. The `GameGui` registers itself as an observer to be notified whenever the game state changes.

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.



In Java, `Observer` is an interface providing an `update()` method. Any object can be an observer by implementing this simple interface.

`Observable`, on the other hand, is a class providing implementations of several useful methods (`addObserver()`, `hasChanged()`, `setChanged()`, `notifyObservers()` etc.). To be an observable, a class should extend this existing class and call `setChanged()` and `notifyObservers()` at appropriate points in its own code (i.e., when its state changes, and when it is done with any possibly mutating public 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);
    }
}
...
```

View = Observer : here the BoardGame (Observable/Subject) informs the view that the state of the game has changed by invoking its update method. This then causes the GUI to reflect the change of state.

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));
   }
}
```

Note how the observable object separates notification into two separate actions: first it calls `setChanged()` at specific points in its code where it is certain that its state has changed, and second, it calls `notifyObservers()` when it is done with a *possibly* mutating action. Notification will only take place if the state has truly changed.

By separating these two actions we (i) ensure that *unnecessary notifications are not issued*, and (ii) we avoid the need for the object itself to implement a way to check if its state has *really* changed.

Handy way of Communicating changes

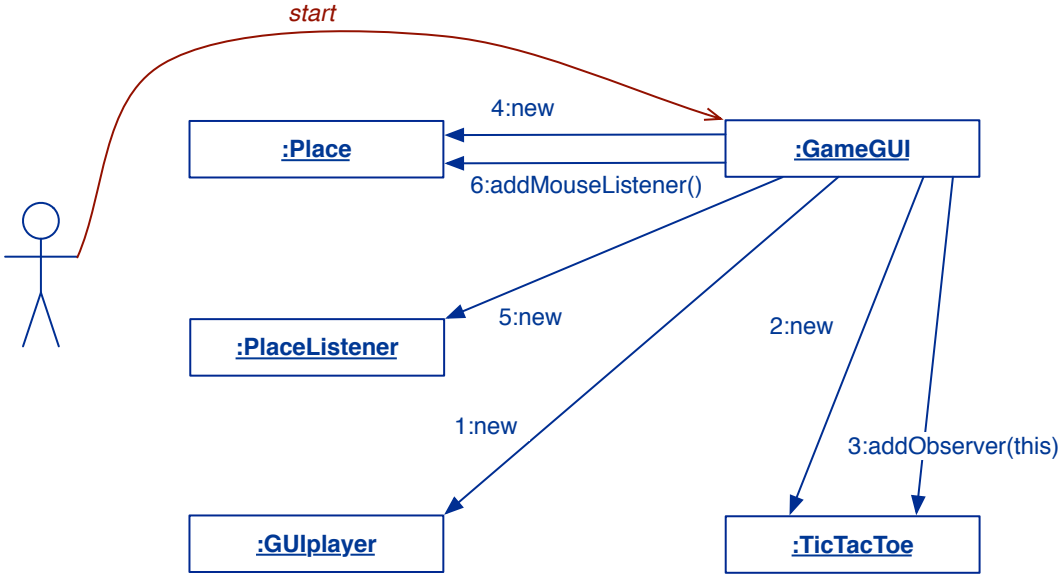
A `Move` instance bundles together information about a change of state in a `BoardGame`:

```
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 + ")";
   }
}
```

`Move` is a very simple data object, but it does serve two useful purposes: (i) to bundle together the state change information, and (ii) to provide a printable representation for feedback.

Setting up the connections

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



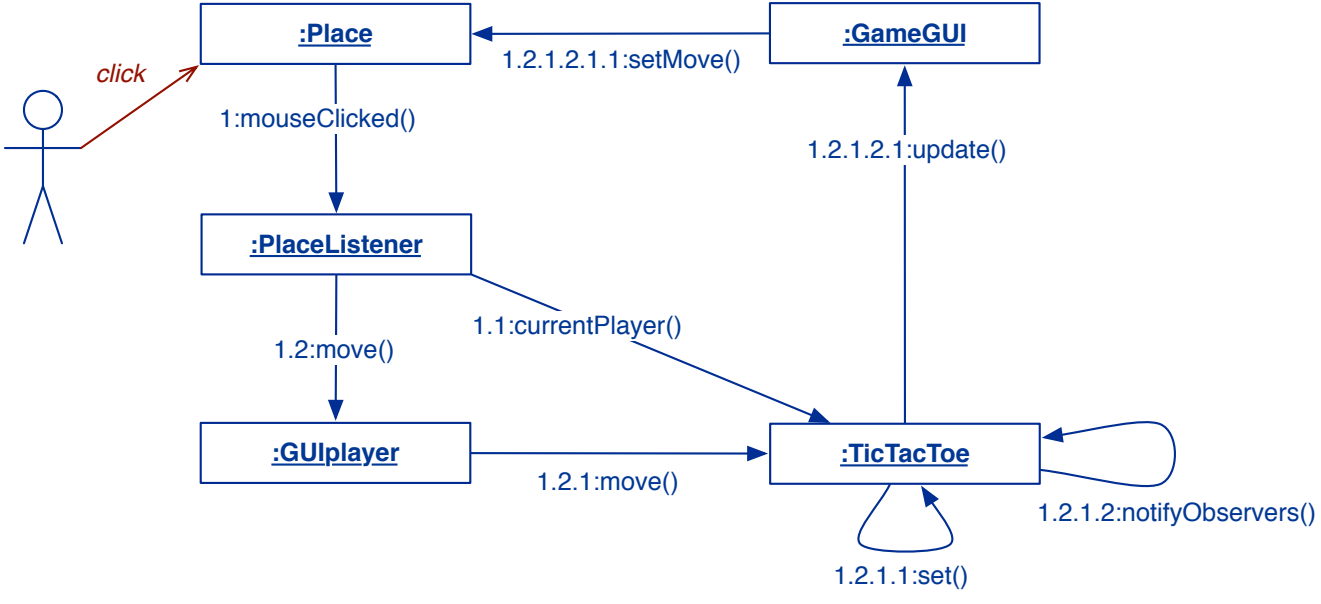
The `GameGUI` *subscribes itself as an* `Observer` to the game (observable), and subscribes a `PlaceListener` to `MouseEvents` for each `Place` on the view of the `BoardGame`.

In the setup phase, we just create all the objects that play a role in the game, and we configure the publish/subscribe relationships.

Note that `Place` is a listener (subscriber) to `GameGUI` events, while the `GameGUI` is an observer (subscriber) to changes in the game state. The publish/subscribe design pattern is therefore used in two different ways to avoid entangling GUI code and game logic: the GUI is completely unaware of game logic, and the game is completely unaware of GUI code. This simplifies the implementation of both parts and makes them both more robust to further changes in design.

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).*

Here we see how the game is played. The (human) player clicks on the GUI, generating a mouse clicked event. This is propagated to the `PlaceListener` which triggers the `GUIPlayer` to make a move on the game. This in turn generates a change in the game state, which finally causes the GUI to be updated.

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();
    checkInput(this.get(col, row).isNobody(),
                      "That square is occupied!");
    ...
}

protected void checkInput(Boolean condition, String
message)
    throws InvalidMoveException
{
    if (!condition) {
        throw new InvalidMoveException(message);
    }
}
```

In Design by Contract, a failed assertion indicates an error in our code. *If our code is free of errors, the assertions should, in principle, never fail.*

But the square that the user selects to make a move is *not under the control of our code*, so we need to check it and take appropriate action if it fails. The `checkInput()` method is meant to express this. Note that we do not take any action here, except to raise an exception, which we will handle somewhere else at a suitable level in our code.

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 input!

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/)

The Java ARchive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

The mechanism for handling JAR files is a standard part of the Java platform's core API. You can click on the jar and the main class of the jar specified by the "manifest" file will be executed.

JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single HTTP transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the author can digitally sign individual entries in a JAR file to authenticate their origin.

```
jar –tvf junit.jar | more
```

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

```
<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

- 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.

Javadoc input

```
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 $
 */
public interface Player {
    /**
     * @return the char representation of this Player
     * @see AbstractBoardGame#toString
     */
    public char mark();
    …
}
```

Javadoc output

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.

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**

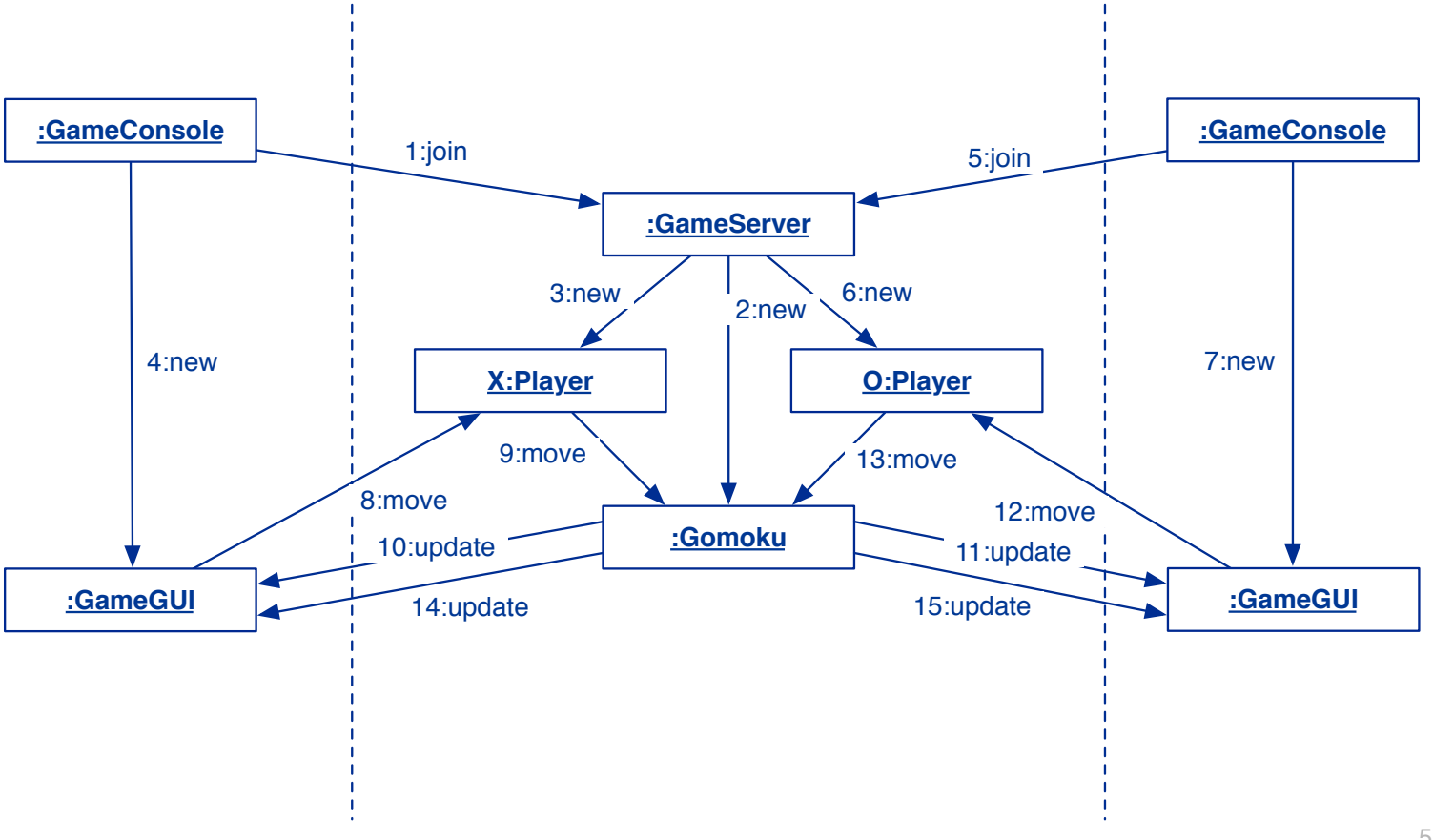
A Networked TicTacToe?

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

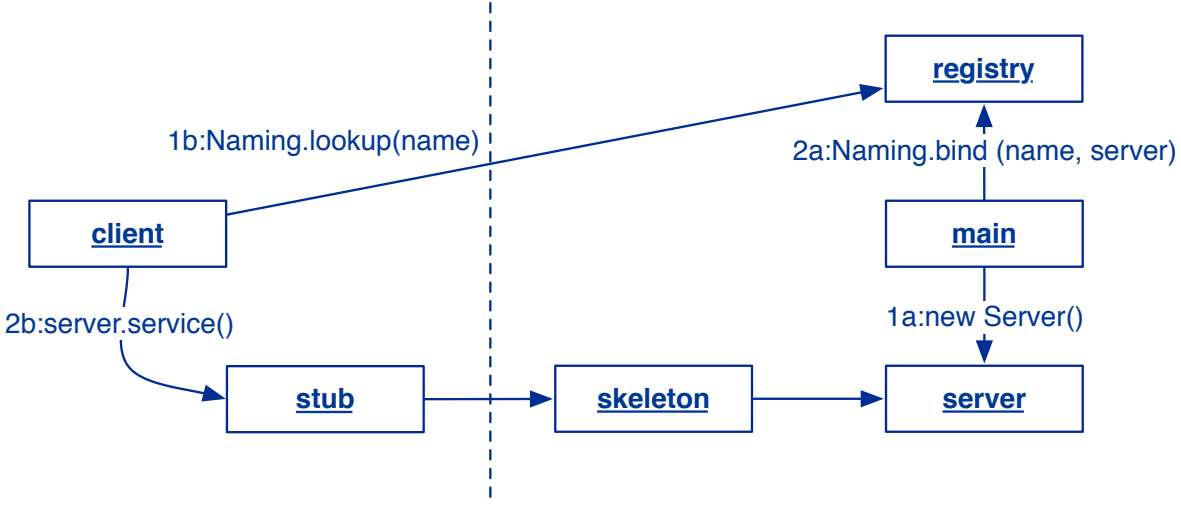

:GameConsole
:GameConsole
1:join
5:join
:GameServer
3:new
2:new
6:new
4:new
7:new
X:Player
O:Player
9:move
13:move
8:move
12:move
:Gomoku
10:update
11:update
:GameGUI
:GameGUI
14:update
15:update

In this scenario, the `GameServer` runs on a dedicated server in a local area network. A user can start up a client application running a `GameConsole` on another client machine on the same network. This causes the `GameServer` to create a new game instance with a `Player` X connected to the `GameGUI` running on the client machine. When a second player requests to join the game, the `Player` O will be created and be connected as a target of actions of that client's GUI.

When either player makes a move on their machine's GUI, the move will be propagated to the virtual player on the server. Whenever the game is updated on the server, *both* client GUIs will be updated.

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* 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).

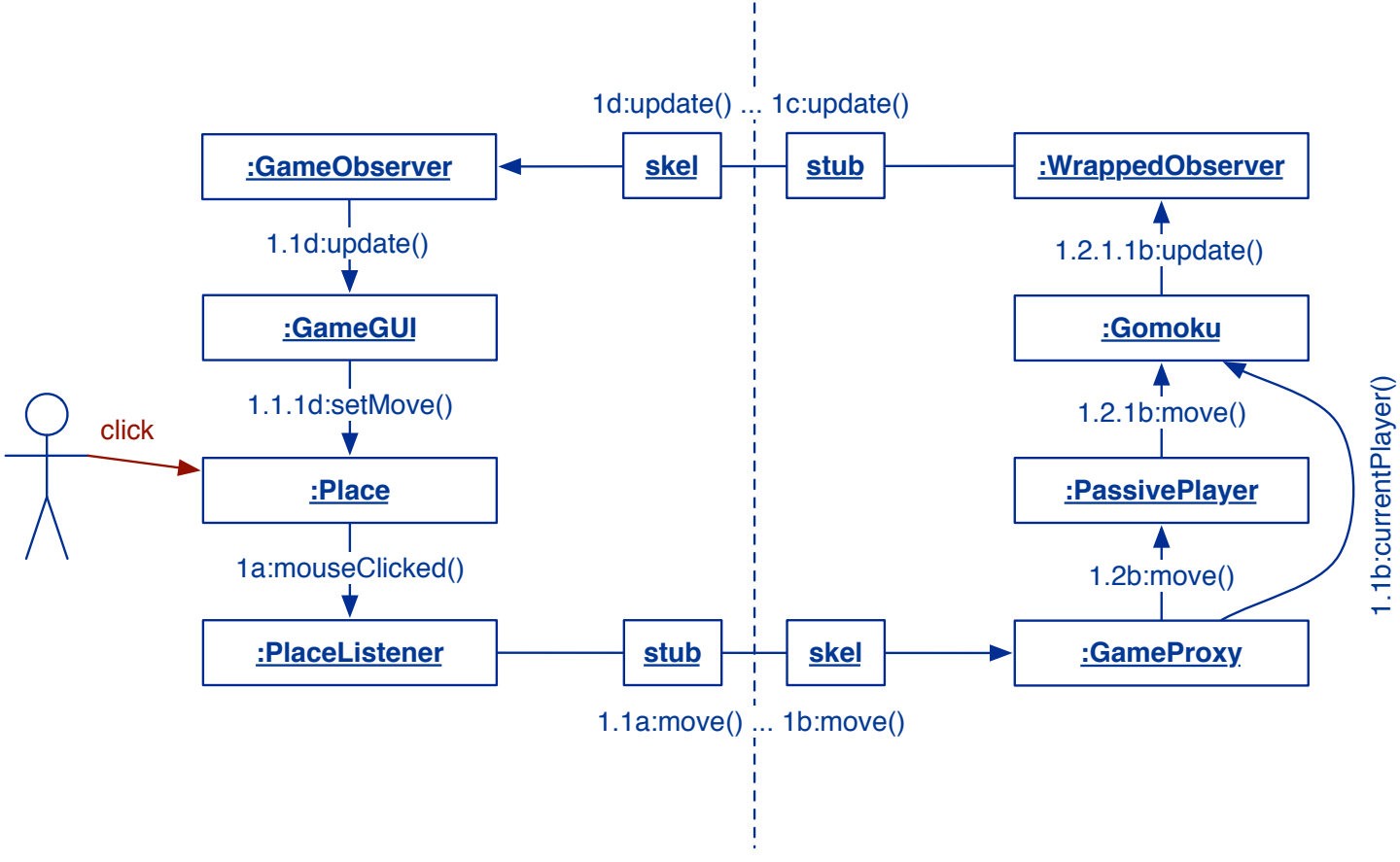
All this works with a dedicated Java API called RMI, supporting remote method invocation. On each of the machines in an RMI application, *remote objects* are represented locally by "*stub*" objects that are responsible for handling the communication.

To invoke a method on a remote object, you simply invoke a normal method on the local stub. The stub translated the method call to network communication to a remote *skeleton* object that translates the communication back to a normal call to the remote object on that machine.

The tricky part is what happens to the method arguments: all arguments are either *serialized objects* that are recreated on the remote machine, or they are themselves remote objects passed by reference (and represented by a stub at the receiving end).

In our design, the `Game`, the `GameServer` and the `Observer` are remote objects, and the `Move` objects are serialized.

Playing the game



Now we can clearly benefit from our separation of GUI and game logic. The GUI runs on client machines and the game logic runs on the server. Although translating the game to work with RMI is not trivial, the impact on our existing design is quite limited. We only have to introduce remote objects at the points where we cross the boundary between the client and the server.

Caveat: RMI only really works well on a local area network due to typical network security restrictions.

What you should know!

1. *The TicTacToe game knows nothing about the GameGUI or Places. How is this achieved? Why is this a good thing?*
2. *What are models, view and controllers?*
3. *What is a Container,Component?*
4. *What does a layout manager do?*
5. *What are events and listeners? Who publishes and who subscribes to events?*
6. *How does the Observer Pattern work?*
7. *Ant*
8. *javadoc*

1)

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

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface, into three parts:

- the *model*, responsible for the domain logic
- the *view*, responsible for the graphical display, and
- the *controller*, responsible for synchronizing the two.

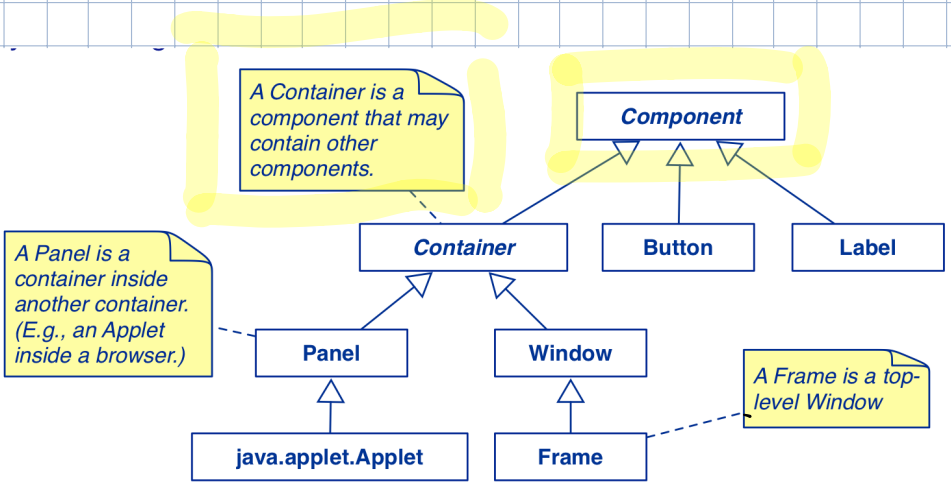
MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

Input ⟶ Processing ⟶ Output

Controller ⟶ Model ⟶ View

2)

3)



Label f.e. is a simple component with no other things inside.

4)

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

A layout manager is an object that implements the layout manager interface and determines the size and position of the components. Although the components can provide size and alignment hints, a container has the final say on the size and position of the components.

5) 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.

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`, ...)

... are handled by subscribed `Listener` *objects*

AWT Framework/
Swing Framework

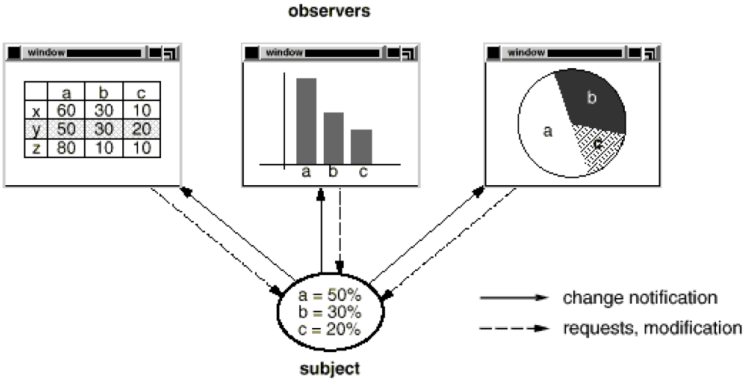
Callback methods

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

Listener: Are objects which subscribed to certain Events. Listeners want to be notified when a state of something they subscribed to changes.

AWT/Swing Components publish events and Listeners subscribe to them.

6)



> Also known as the *publish/subscribe* design pattern — to observe the state of an object in a program.
> *Observers* registered to *observe* state changes in an observable object, known as the *subject*.
> The *subject* maintains a set of *observers* to notify whenever its state changes.

7)

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.)

8)

Javadoc

- 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.

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?*