GUI construction

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

Source

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

http://docs.oracle.com/javase/8/javase-clienttechnologies.htm

Roadmap

1. Model-View-Controller (MVC)
2. JavaFX Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. AWT, Swing, SWT
6. Jar files

Roadmap

1. **Model-View-Controller (MVC)**
2. JavaFX Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. AWT, Swing, SWT
6. Jar files

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:

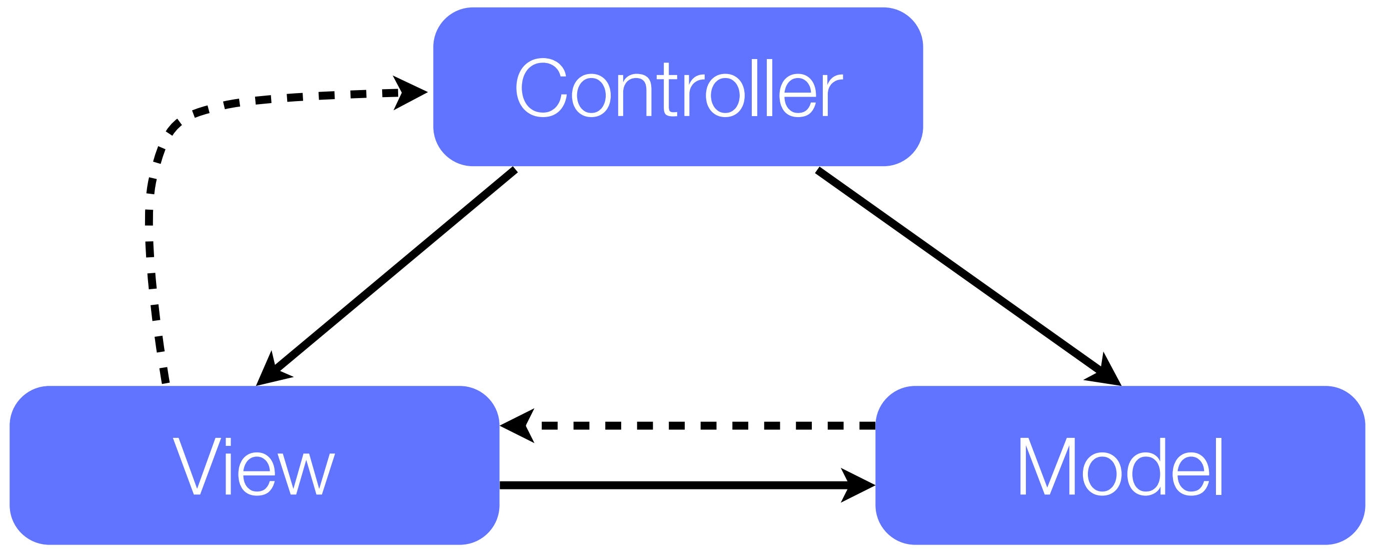
- running the game with graphical display and mouse input

Model-View-Controller

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

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

Model-View-Controller

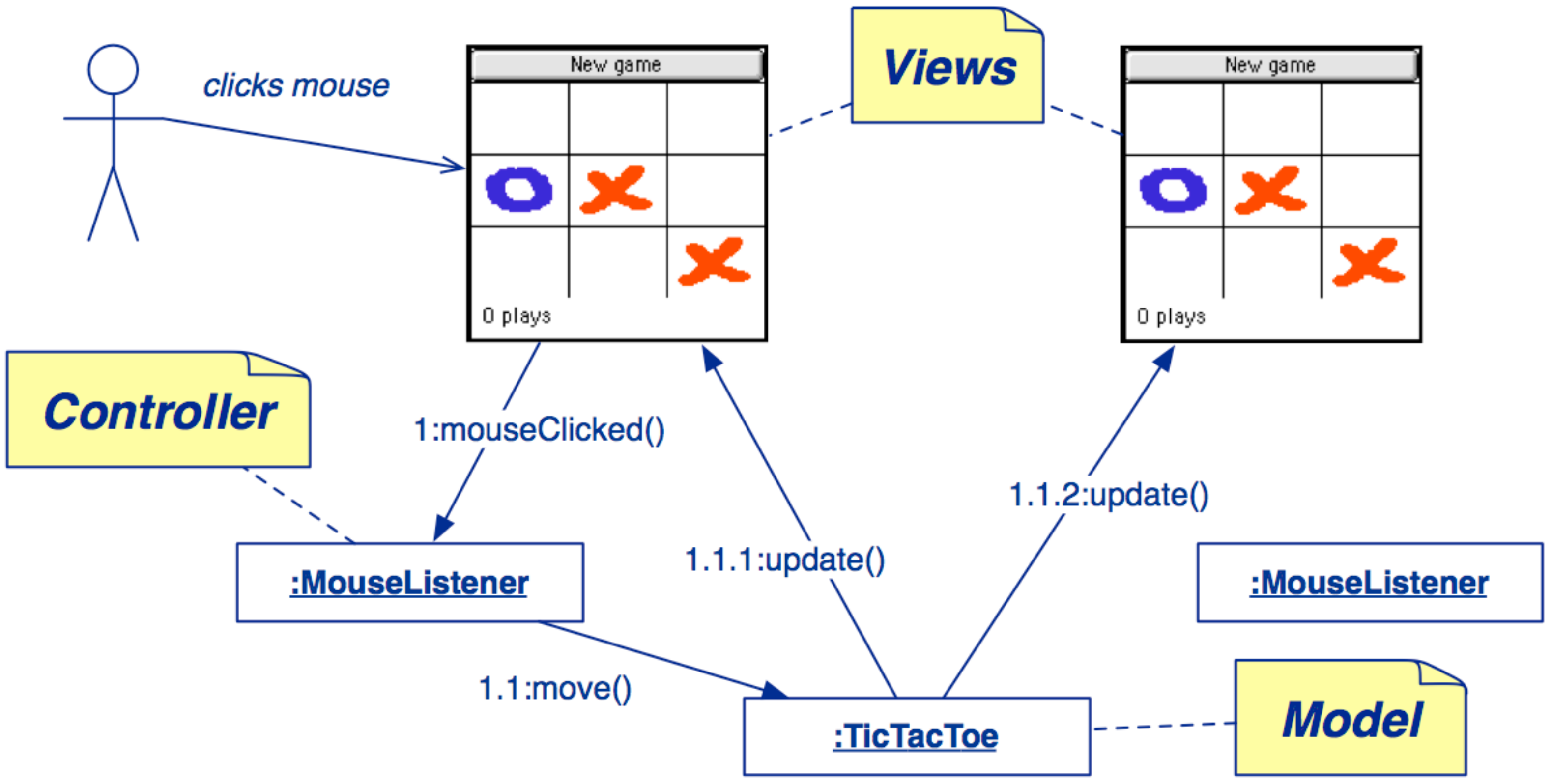


→ direct association

⇢ 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

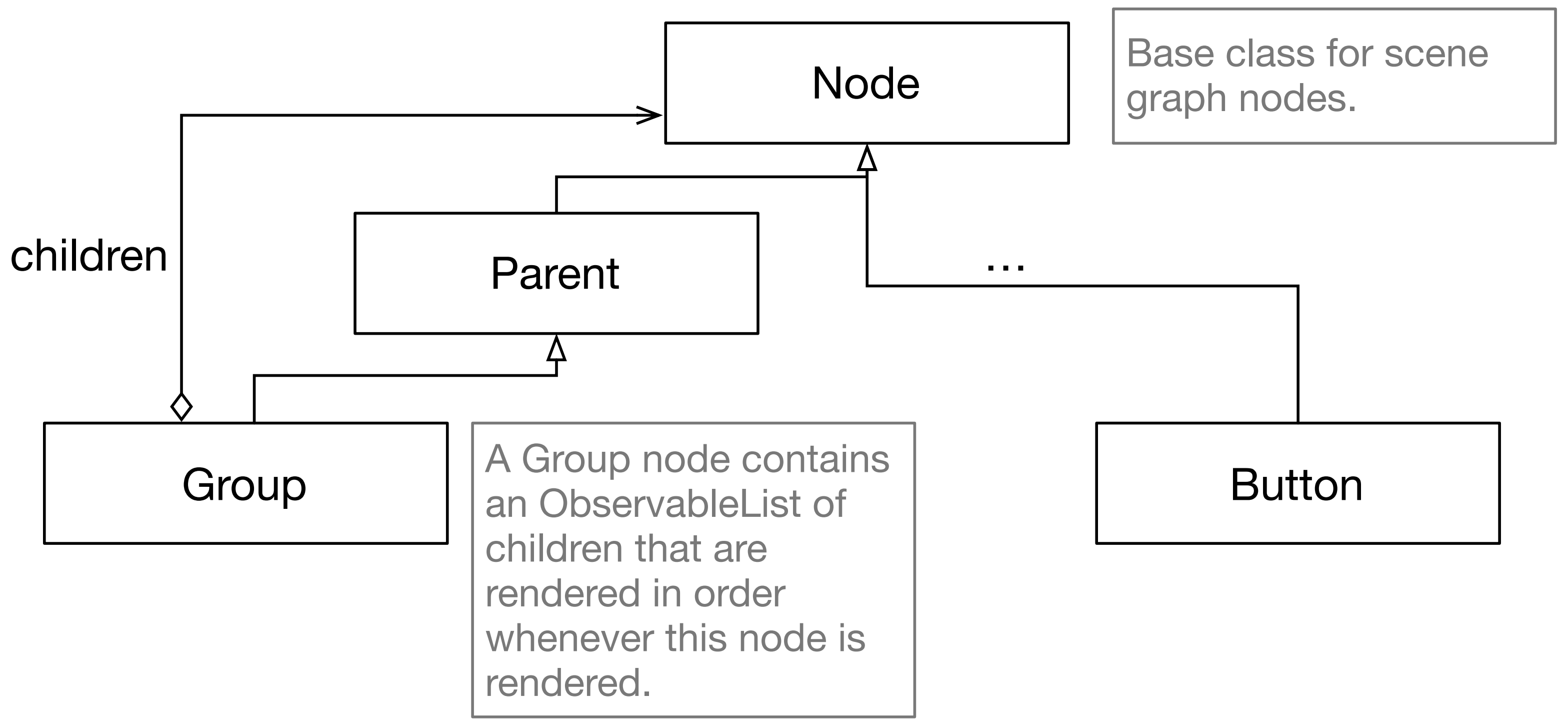


Roadmap

1. Model-View-Controller (MVC)
2. **JavaFX Components, Containers and Layout Managers**
3. Events and Listeners
4. Observers and Observables
5. AWT, Swing, SWT
6. Jar files

JavaFX Components and Containers

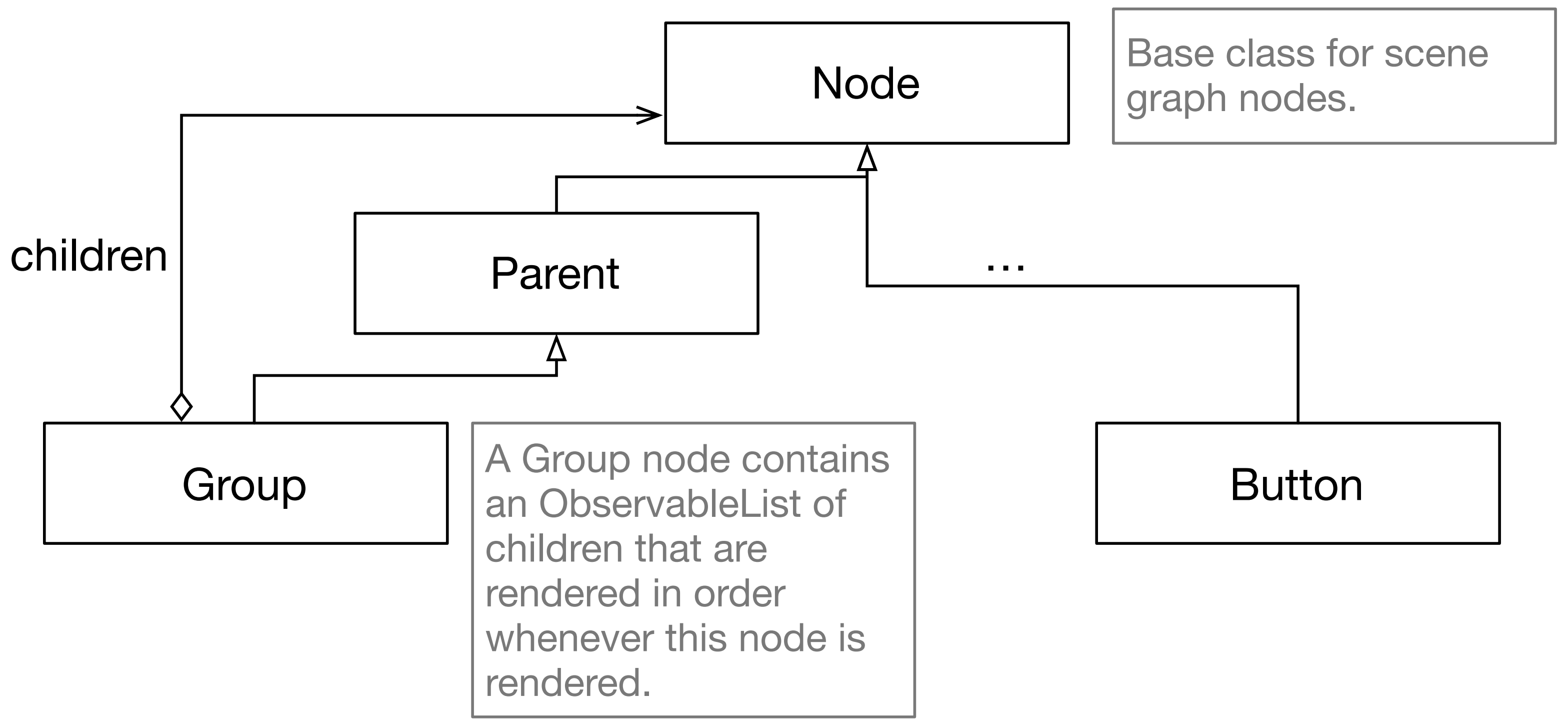
The javafx package defines GUI components, containers, widgets


Node
Base class for scene graph nodes.
children
Parent
…
Group
A Group node contains an ObservableList of children that are rendered in order whenever this node is rendered.
Button

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

JavaFX Components and Containers

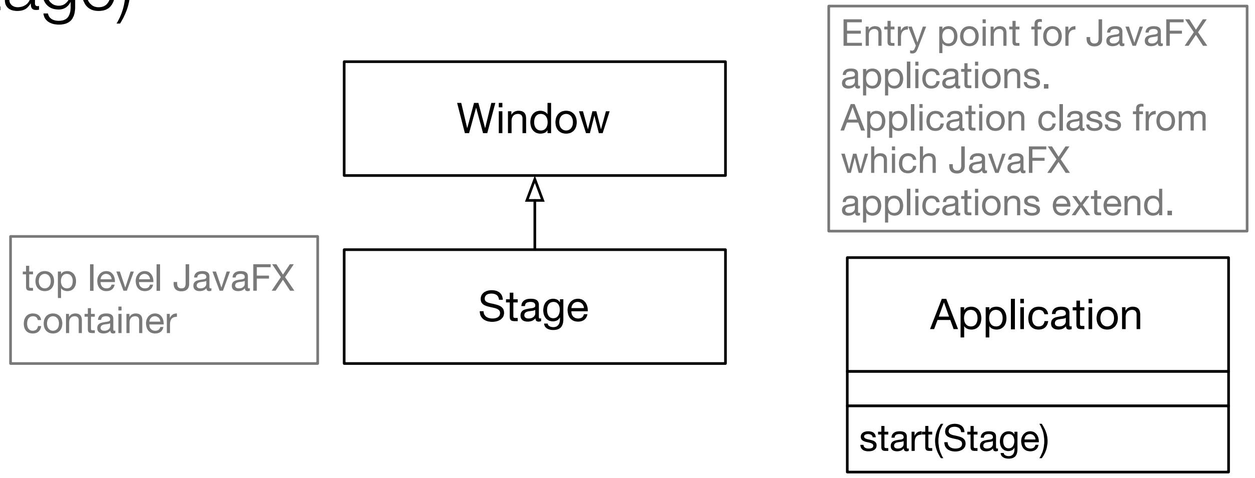
The javafx package defines GUI components, containers, widgets


Node
Base class for scene graph nodes.
children
Parent
...
Group
A Group node contains an ObservableList of children that are rendered in order whenever this node is rendered.
Button

```
Button button = new Button();
button.setText("Say 'Hello World'");
Group root = new Group();
root.getChildren().add(button);
```

JavaFX top level container & window

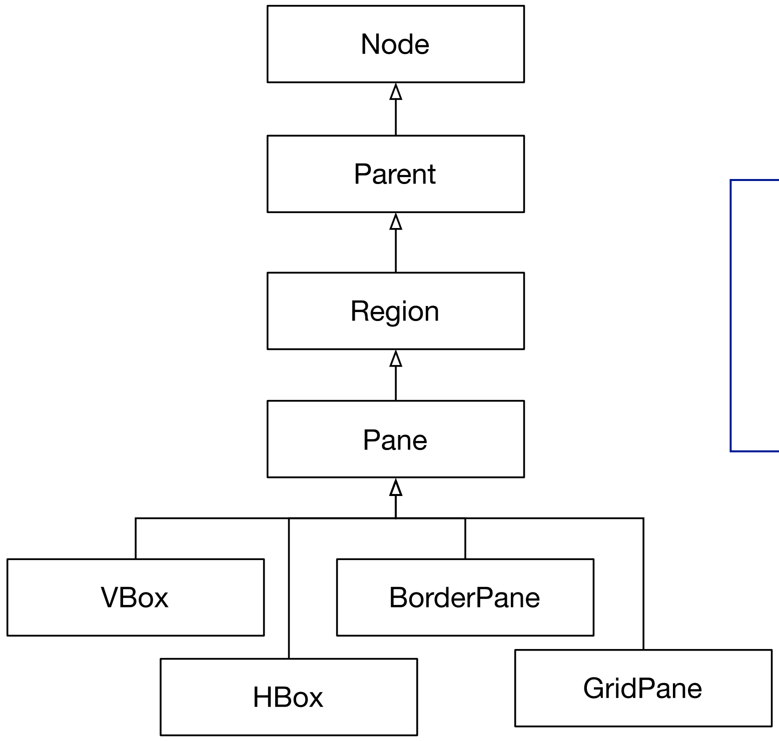
A JavaFX UI has to subclass Application and override start(Stage)


Window
Stage
top level JavaFX container
Entry point for JavaFX applications. Application class from which JavaFX applications extend.
Application
start(Stage)

```java
public class HelloWorld extends Application {
    @Override
    public void start(Stage stage) {
       stage.setTitle("Hello World!");
       . . .
       stage.setScene(new Scene(root, 300, 250));
       stage.show();
    }
}
```

Layout Management

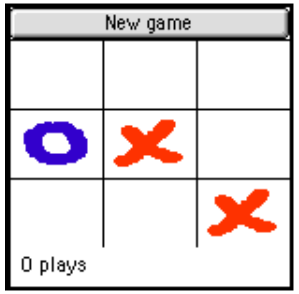
The *Pane* defines how the nodes are arranged in a container (size and position)

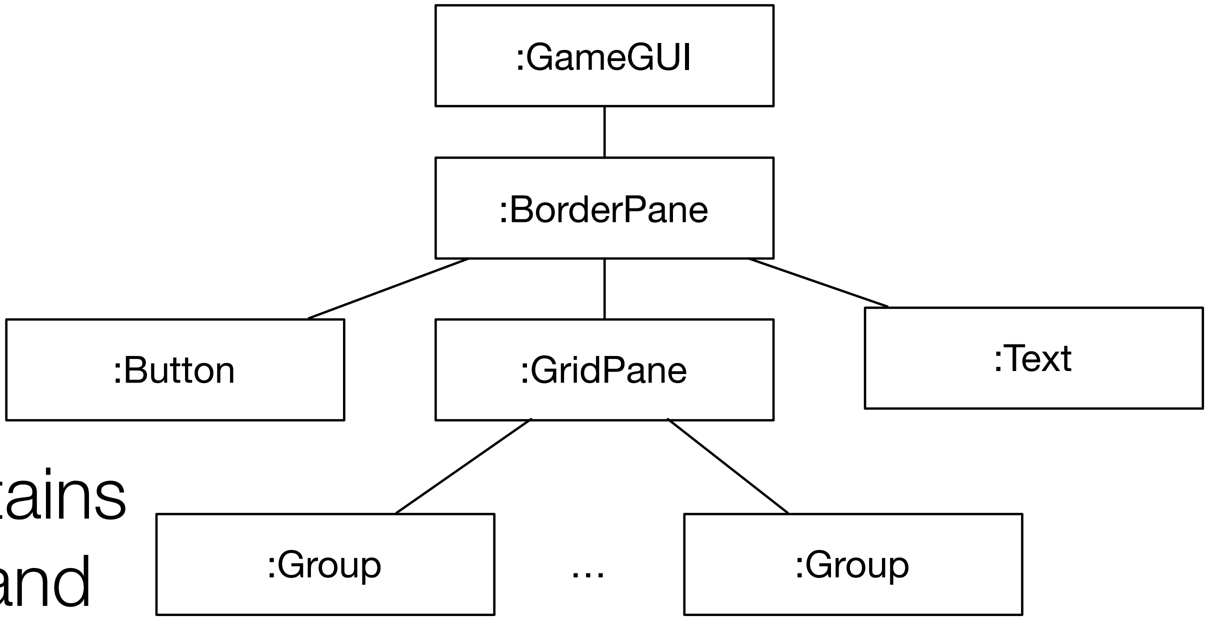


```
GridPane grid = new GridPane();
grid.setHgap(5); //horizontal gap between nodes
grid.setVgap(5); //vertical gap
grid.add(new Button("Hello"), 0, 1);
grid.add(new Button("World"), 0, 2);
```

The GameGUI

The GameGUI is an *Application* using a *BorderPane* (with a centre and up to four border components), and containing a *Button* ("top"), a *GridPane* ("center") and a *Text* ("bottom").





The central Panel itself contains a grid of squares (Group) and uses a GridPane.

Laying out the GameGUI

```
public class GameGUI extends Application implements Observer {
    public void start(Stage stage) {
        stage.setTitle("Tic Tac Toe");

        game = makeGame();
        BorderPane borderpane = new BorderPane();
        borderpane.setTop(makeControls());
        borderpane.setCenter(makeGrid());
        statusbar = new Text();
        borderpane.setBottom(statusbar);

        // Create a JavaFX scene
        Scene scene = new Scene(borderpane, … , … ));
        stage.setScene(scene);
        stage.show();
    }
    …
}
```

Helper methods

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

```
private Node makeControls() {
    Button again = new Button("New game");

    again.setPrefSize(CELL_SIZE*game.getCols(),20);
    again.setOnAction( . . . );
    return again;
}
```

Roadmap

1. Model-View-Controller (MVC)
2. JavaFX Components, Containers and Layout Managers
3. **Events and Listeners**
4. Observers and Observables
5. AWT, Swing, SWT
6. Jar files

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 provided by the package javafx.event

Concurrency and JavaFX

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

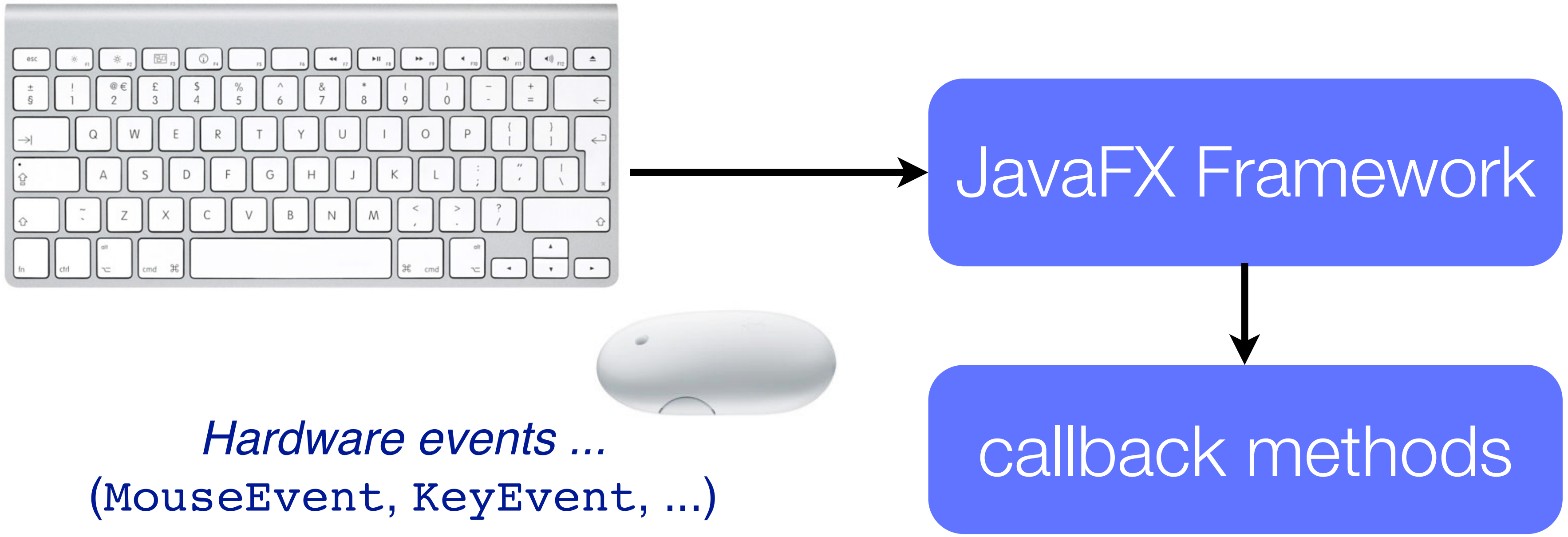
The runtime of the JavaFX framework creates threads

- 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:



Nodes *publish* events and (possibly multiple) Listeners *subscribe* interest in them

Events and Listeners (II)

Every Node publishes a variety of different events with associated Listener interfaces

User Action	Event Type	Class
Key on the keyboard is pressed.	`KeyEvent`	`Node`, `Scene`
Mouse is moved or a button on the mouse is pressed.	`MouseEvent`	`Node`, `Scene`
Full mouse press-drag-release action is performed.	`MouseDragEvent`	`Node`, `Scene`
Swipe gesture is performed on an object	`SwipeEvent`	`Node`, `Scene`
Context menu is requested	`ContextMenuEvent`	`Node`, `Scene`
Button is pressed, combo box is shown or hidden, or a menu item is selected.	`ActionEvent`	`ButtonBase`, `ComboBoxBase`, `ContextMenu`, `MenuItem`, `TextField`
Window is closed, shown, or hidden.	`WindowEvent`	`Window`

Events and Listeners (III)

Convenience methods for registering event handlers have the following format:

```
setOnEvent-type(EventHandler<? super event-class> value)
```

`Event-type` is the type of event that the handler processes, for example, `setOnKeyTyped` for KEY_TYPED events or `setOnMouseClicked` for MOUSE_CLICKED events. event-class is the class that defines the event type, for example, `KeyEvent` for events related to keyboard input or `MouseEvent` for events related to mouse input.

Listening for Button events

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

```
    private Node makeControls() {
        Button again = new Button("New game");

        again.setPrefSize(horizontalSize(), 20);
        again.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                showFeedBack("starting new game ...");
                newGame();
            }
        });
        return again;
    }
```

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

Listening for Button events

When we create the “New game” Button, we *attach an ActionListener* with the Butto

method:

Instance an unnamed subclass of EventHandler()

```
    private Node makeControls() {
        Button again = new Button("N

        again.setPrefSize(horizontalSize(), 20);
        again.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                showFeedBack("starting new game ...");
                newGame();
            }
        });
        return again;
    }
```

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

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);
}
```

```java
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

Listening for mouse clicks

We also attach a MouseListener to each Place on the board

```
    private Node makeGrid() {
        int cols = game.getCols();
        int rows = game.getRows();

        GridPane grid = new GridPane();
        grid.setHgap(5);
        grid.setVgap(5);

        places = new Place[cols][rows];
        for (int row = rows - 1; row >= 0; row--) {
            for (int col = 0; col < cols; col++) {
                Place p = new Place(col, row);
                p.setOnMouseClicked(new PlaceListener(p, this));
                grid.add(p, col, row);
                places[col][row] = p;
            }
        }
        return grid;
    }
```

The PlaceListener

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

```
public class PlaceListener implements EventHandler<MouseEvent> {
    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 handle( ) method:

```
@Override
public void handle(MouseEvent event) {
    …
    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!");
    }
}
```

Roadmap

1. Model-View-Controller (MVC)
2. JavaFX Components, Containers and Layout Managers
3. Events and Listeners
4. **Observers and Observables**
5. AWT, Swing, SWT
6. Jar files

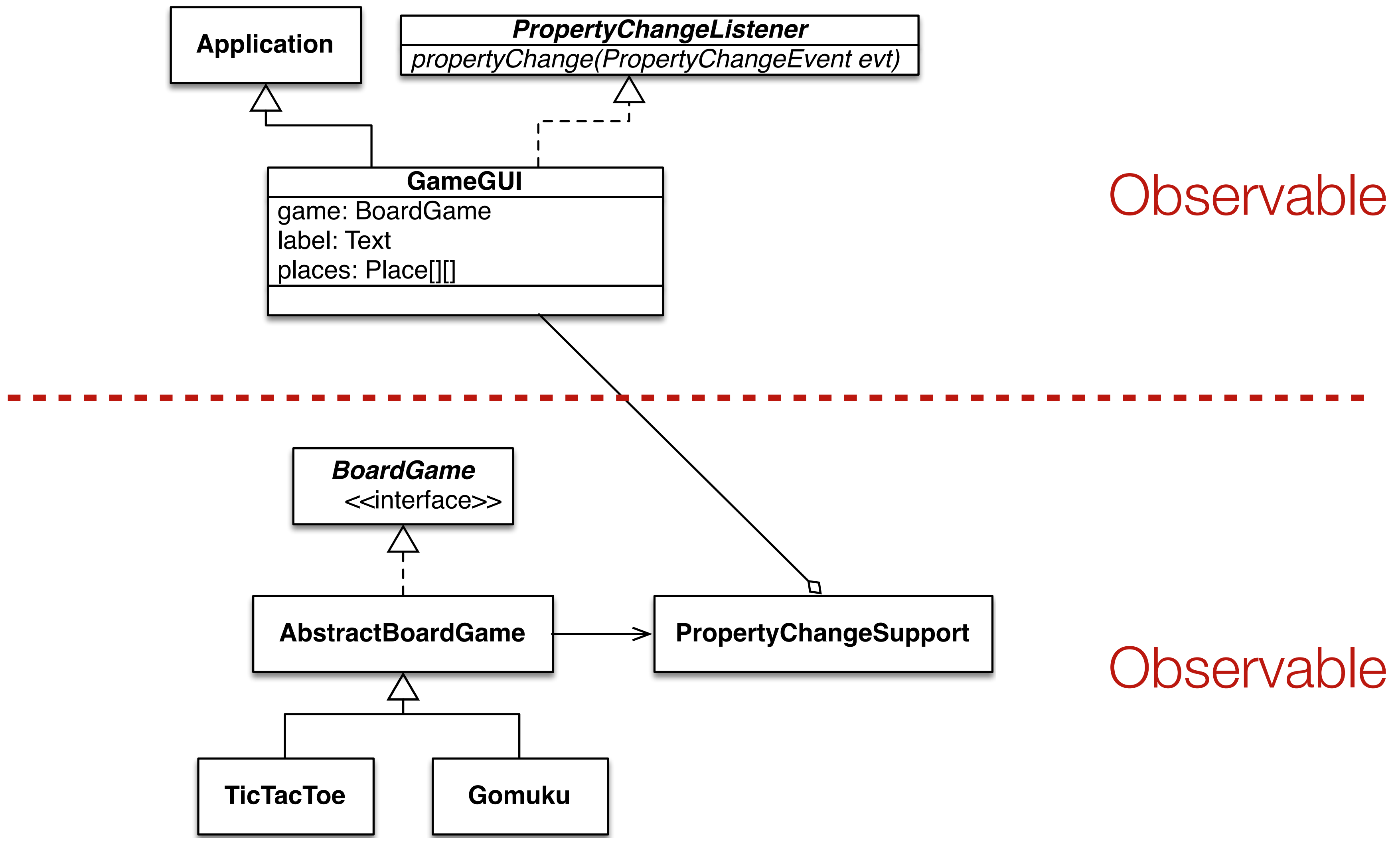
The Observer Pattern (remember?)

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

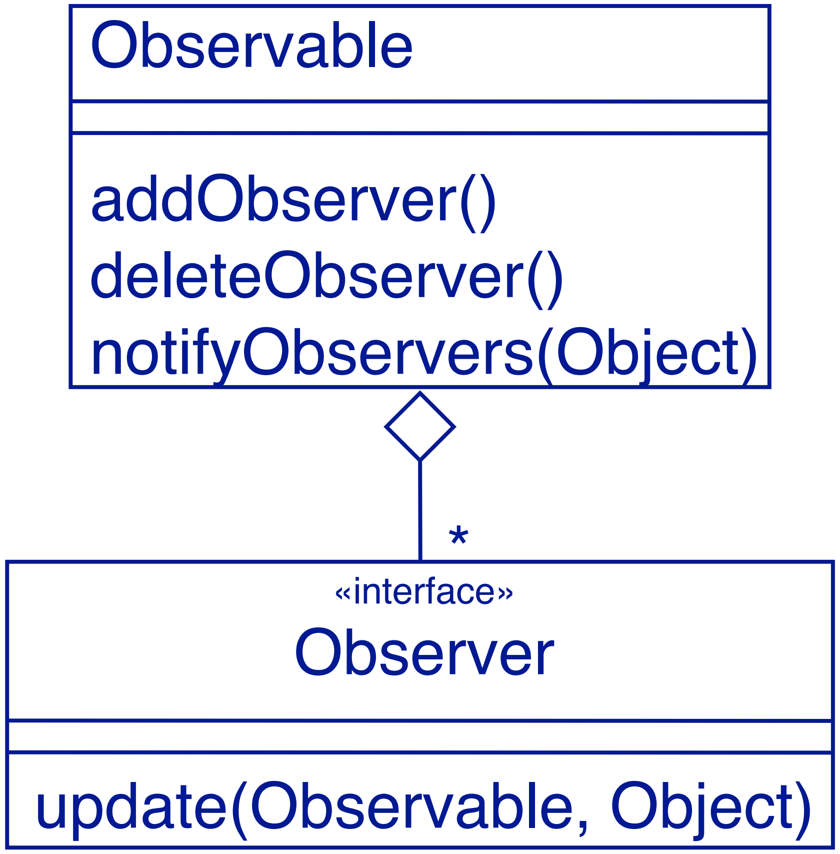


Observers and Observables (< Java 9)

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 Application implements PropertyChangeListener {
    public void start(Stage stage) {
        stage.setTitle("Tic Tac Toe");

        game = makeGame();

        // notify GameGui if change of state
        game.addObserver(this);

…
```

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 Application implements PropertyChangeListener {
    . . .
    public void propertyChange (PropertyChangeEvent evt) {
        Move move = (Move) arg;
        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 implements BoardGame
{  protected PropertyChangeSupport changes;
  public AbstractBoardGame(Player playerX, Player playerO){
  … changes = new PropertyChangeSupport(this);
  … }

  public void move(int col, int row, Player p) {
       …
       changes.firePropertyChange(new
PropertyChangeEvent(this, "update", null, new Move(col, row,
p)));
   }
}
```

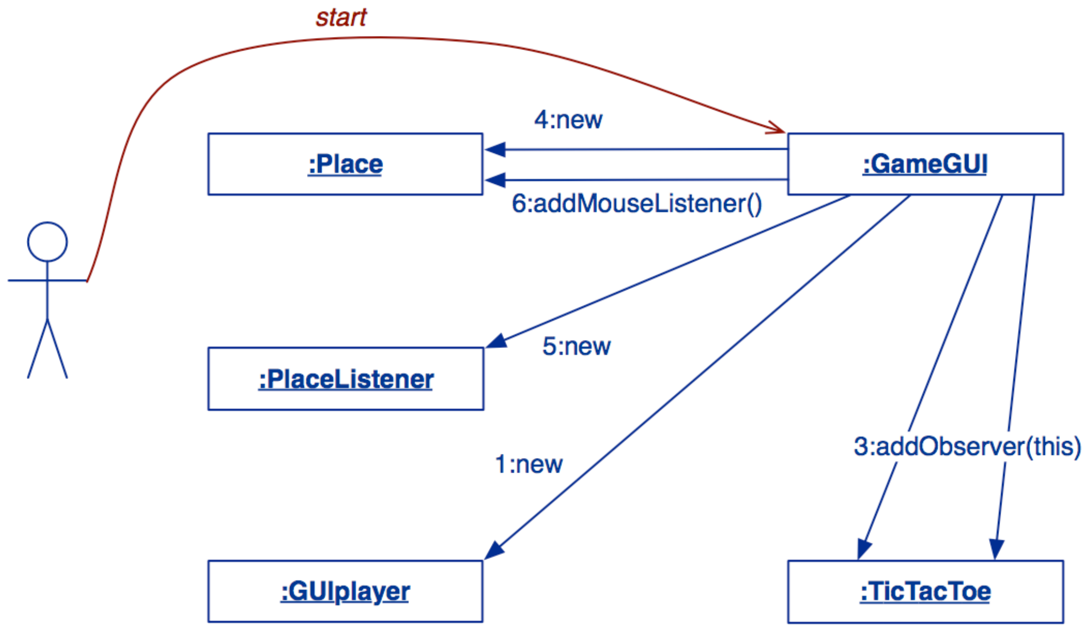
Handy way of Communicating changes

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

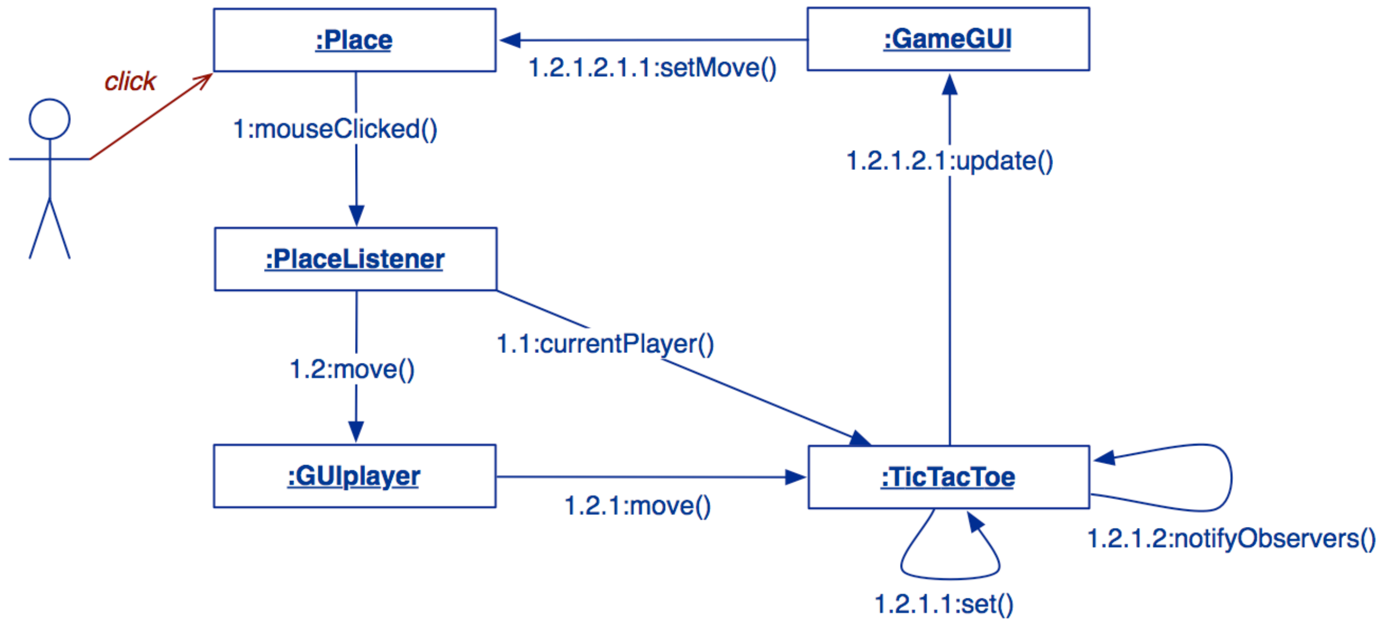
When the GameGUI is created, the *model* (*BoardGame*), *view* (*GameGui*) and *controller* (*Place*) components are instantiated



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. JavaFX Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. **AWT, Swing, SWT**
6. Jar files

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 is likely to be the new mainstream UI framework for Java

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

Roadmap

1. Model-View-Controller (MVC)
2. JavaFX Components, Containers and Layout Managers
3. Events and Listeners
4. Observers and Observables
5. AWT, Swing, SWT
6. **Jar files**

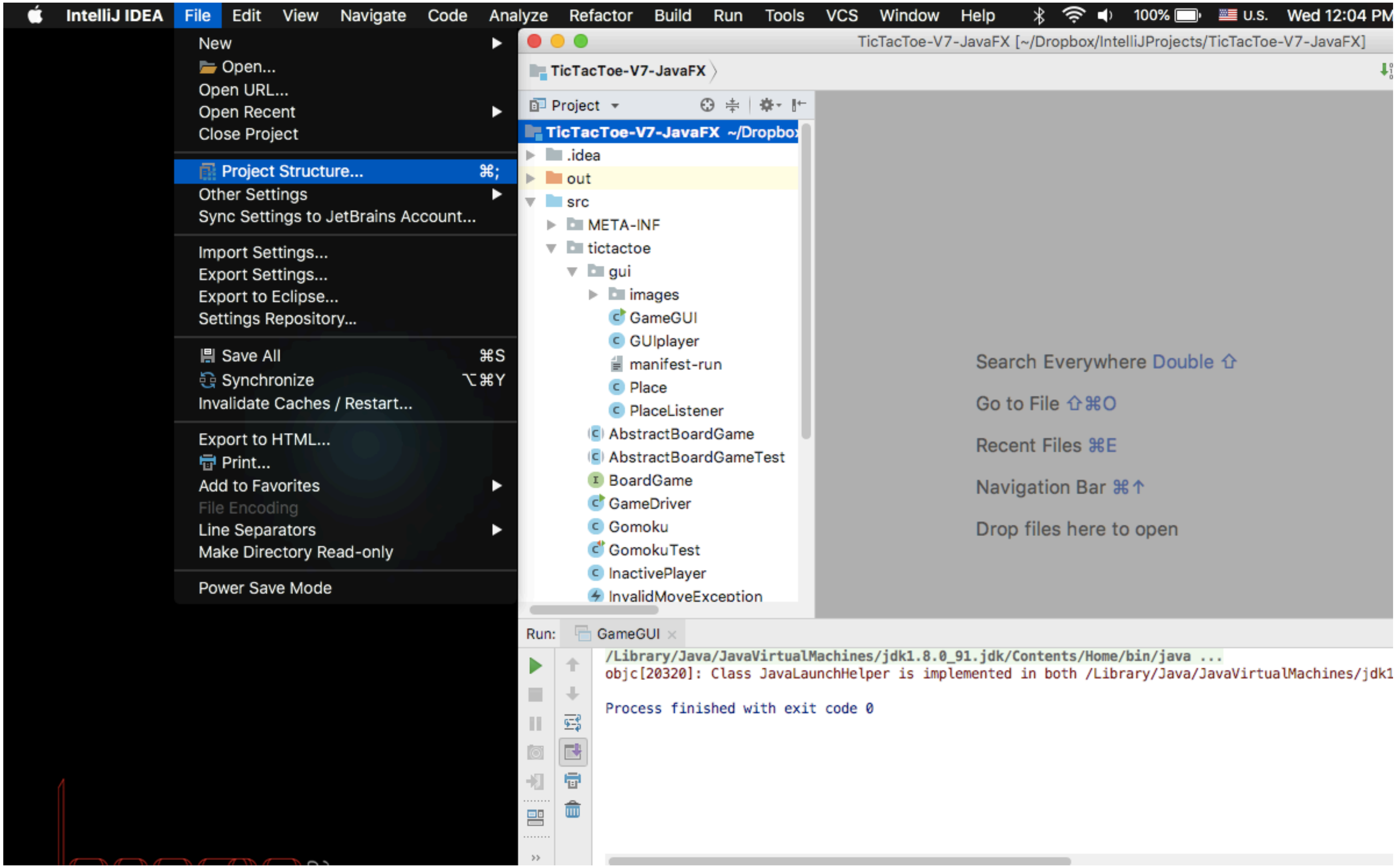
JAR File

A JAR (Java ARchive) is a package file format used to aggregate many java class files and their resources (text, images, …) into one file for distribution

A JAR file is typically used as a deployment unit

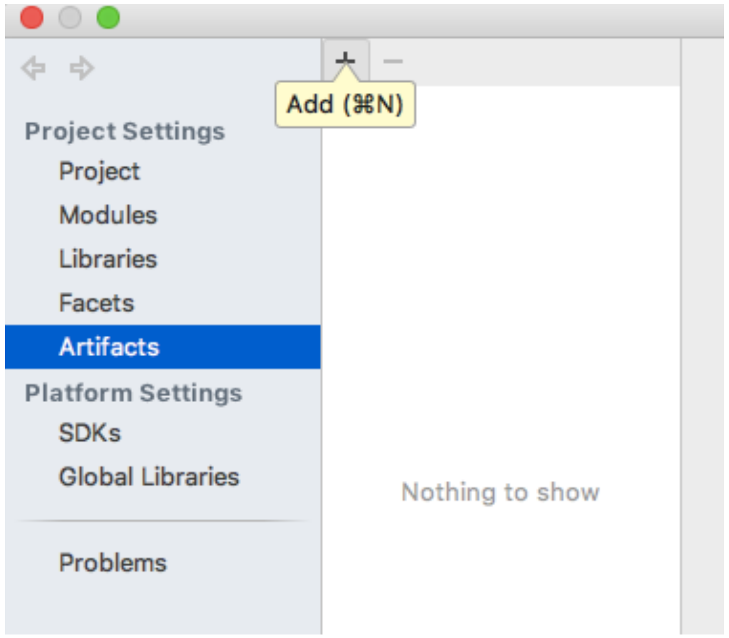
The following slides show how to create a JAR file for the TicTacToe game using IntelliJ and Eclipse

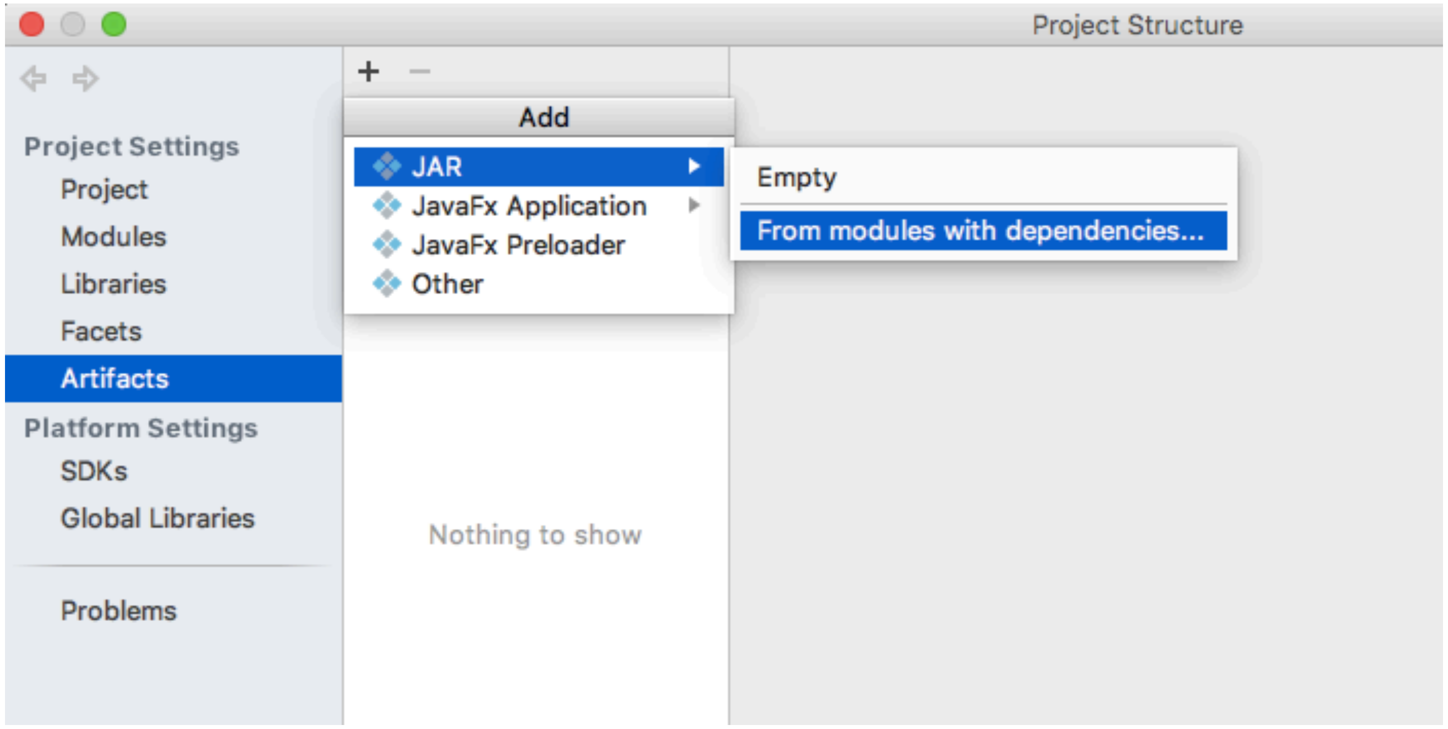
IntelliJ



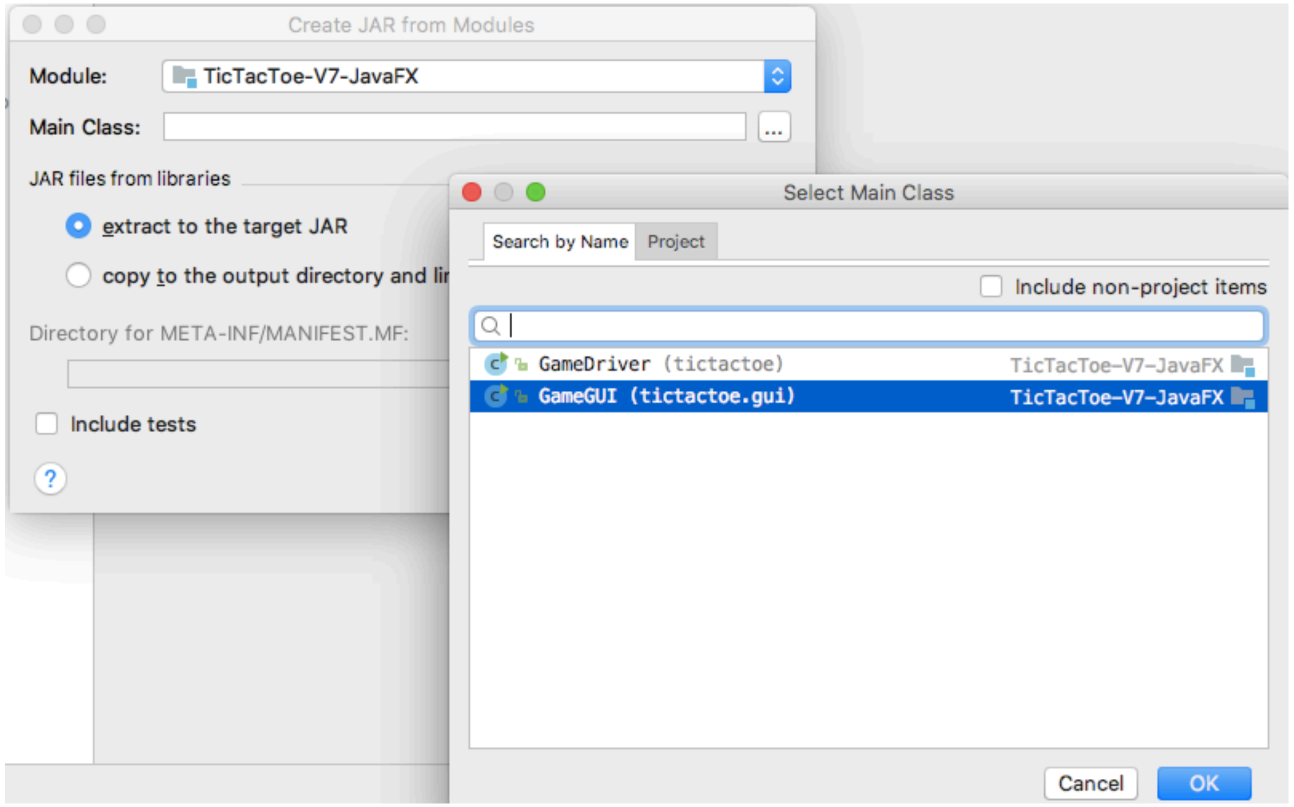
```
/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java ...
objc[20320]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachines/jdk1

Process finished with exit code 0
```

IntelliJ


Add (⌘N)
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Nothing to show


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Add
JAR
JavaFx Application
JavaFx Preloader
Other
Empty
From modules with dependencies...
Nothing to show

IntelliJ


Create JAR from Modules

Module: TicTacToe-V7-JavaFX

Main Class:

JAR files from libraries

- (●) extract to the target JAR
- ( ) copy to the output directory and lir

Directory for META-INF/MANIFEST.MF:

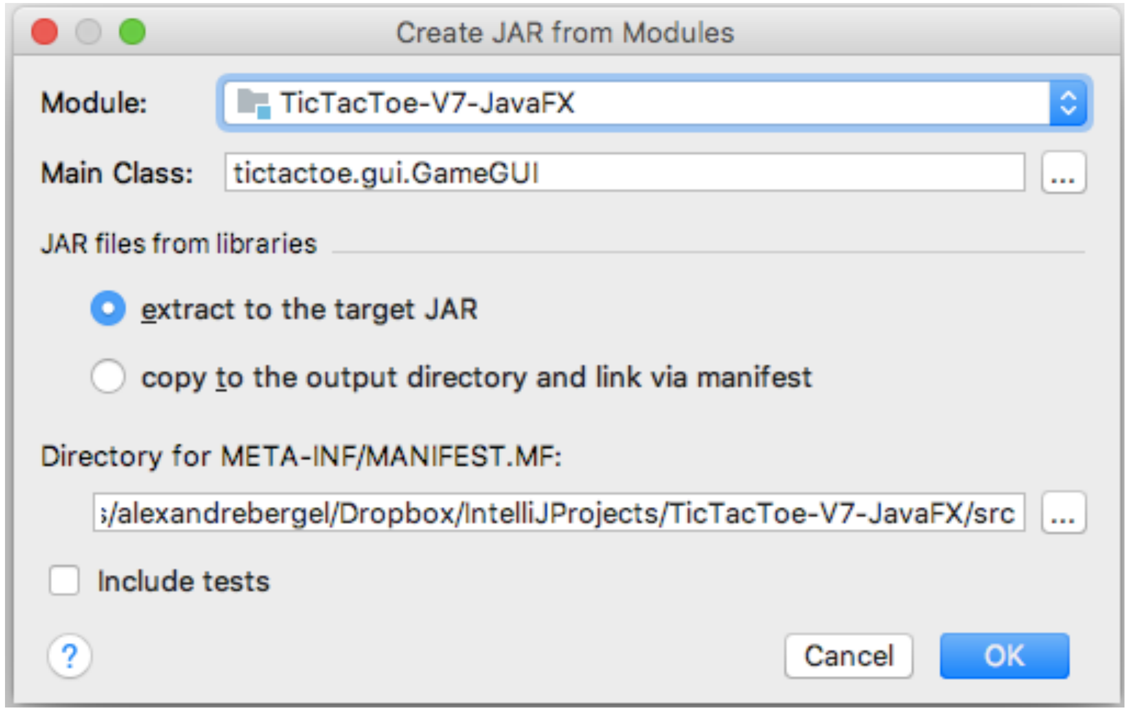
[ ] Include tests

Select Main Class

Search by Name | Project

[ ] Include non-project items

```
GameDriver (tictactoe)        TicTacToe-V7-JavaFX
GameGUI (tictactoe.gui)       TicTacToe-V7-JavaFX
```


Create JAR from Modules

Module: TicTacToe-V7-JavaFX

Main Class: tictactoe.gui.GameGUI

JAR files from libraries

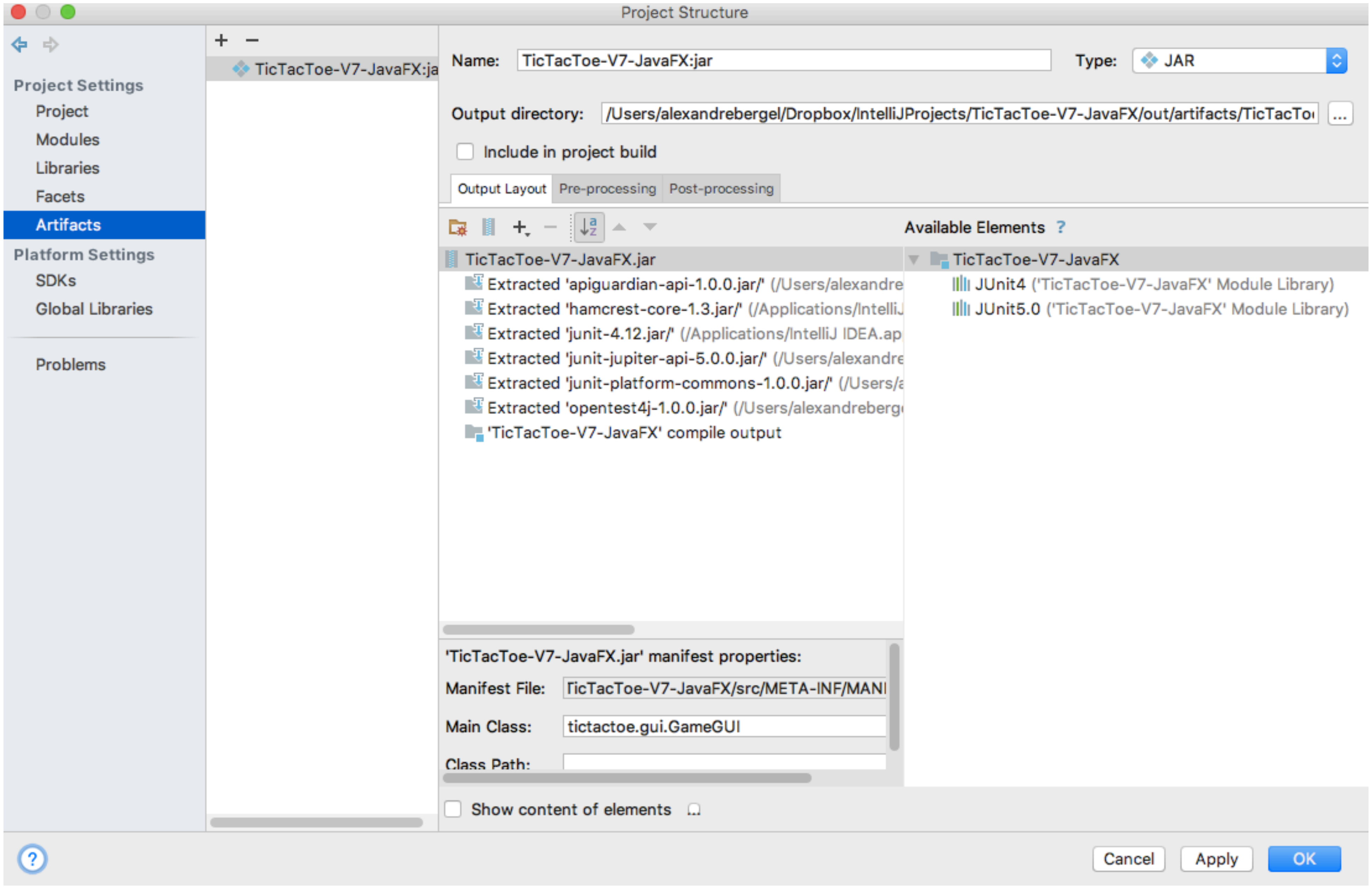
- (●) extract to the target JAR
- ( ) copy to the output directory and link via manifest

Directory for META-INF/MANIFEST.MF:

:/alexandrebergel/Dropbox/IntelliJProjects/TicTacToe-V7-JavaFX/src

[ ] Include tests

IntelliJ



Name: TicTacToe-V7-JavaFX:jar

Type: JAR

Output directory: /Users/alexandrebergel/Dropbox/IntelliJProjects/TicTacToe-V7-JavaFX/out/artifacts/TicTacTo

Include in project build

TicTacToe-V7-JavaFX.jar
- Extracted 'apiguardian-api-1.0.0.jar/' (/Users/alexandre
- Extracted 'hamcrest-core-1.3.jar/' (/Applications/IntelliJ
- Extracted 'junit-4.12.jar/' (/Applications/IntelliJ IDEA.ap
- Extracted 'junit-jupiter-api-5.0.0.jar/' (/Users/alexandre
- Extracted 'junit-platform-commons-1.0.0.jar/' (/Users/a
- Extracted 'opentest4j-1.0.0.jar/' (/Users/alexandreberg
- 'TicTacToe-V7-JavaFX' compile output

TicTacToe-V7-JavaFX
- JUnit4 ('TicTacToe-V7-JavaFX' Module Library)
- JUnit5.0 ('TicTacToe-V7-JavaFX' Module Library)

'TicTacToe-V7-JavaFX.jar' manifest properties:

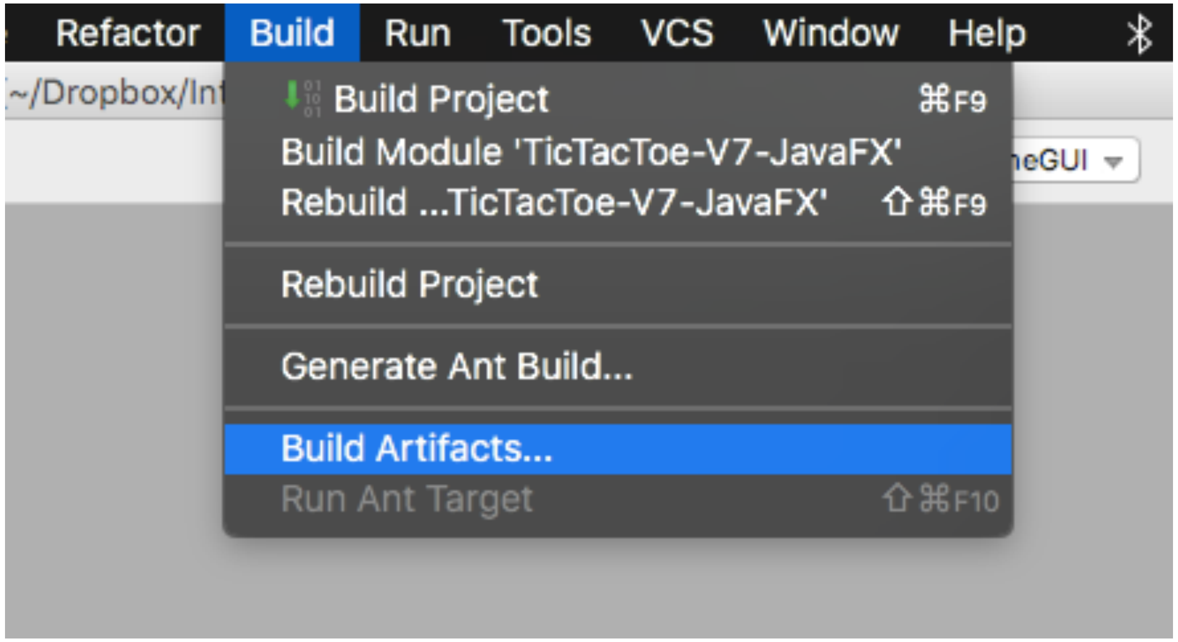
Manifest File: TicTacToe-V7-JavaFX/src/META-INF/MANI

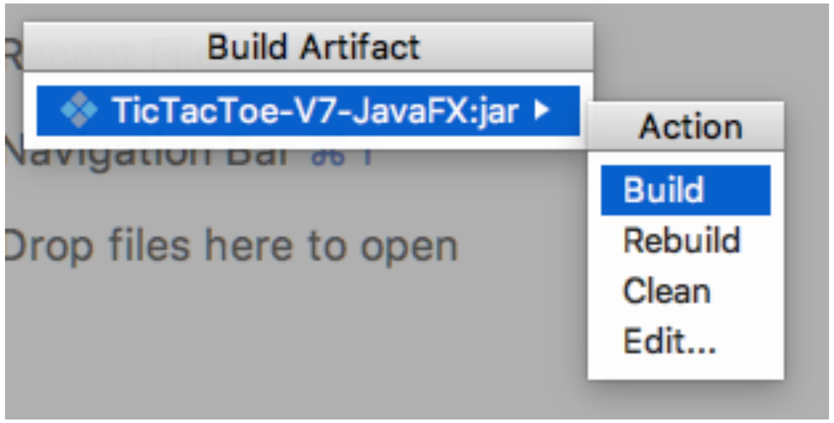
Main Class: tictactoe.gui.GameGUI

Class Path:

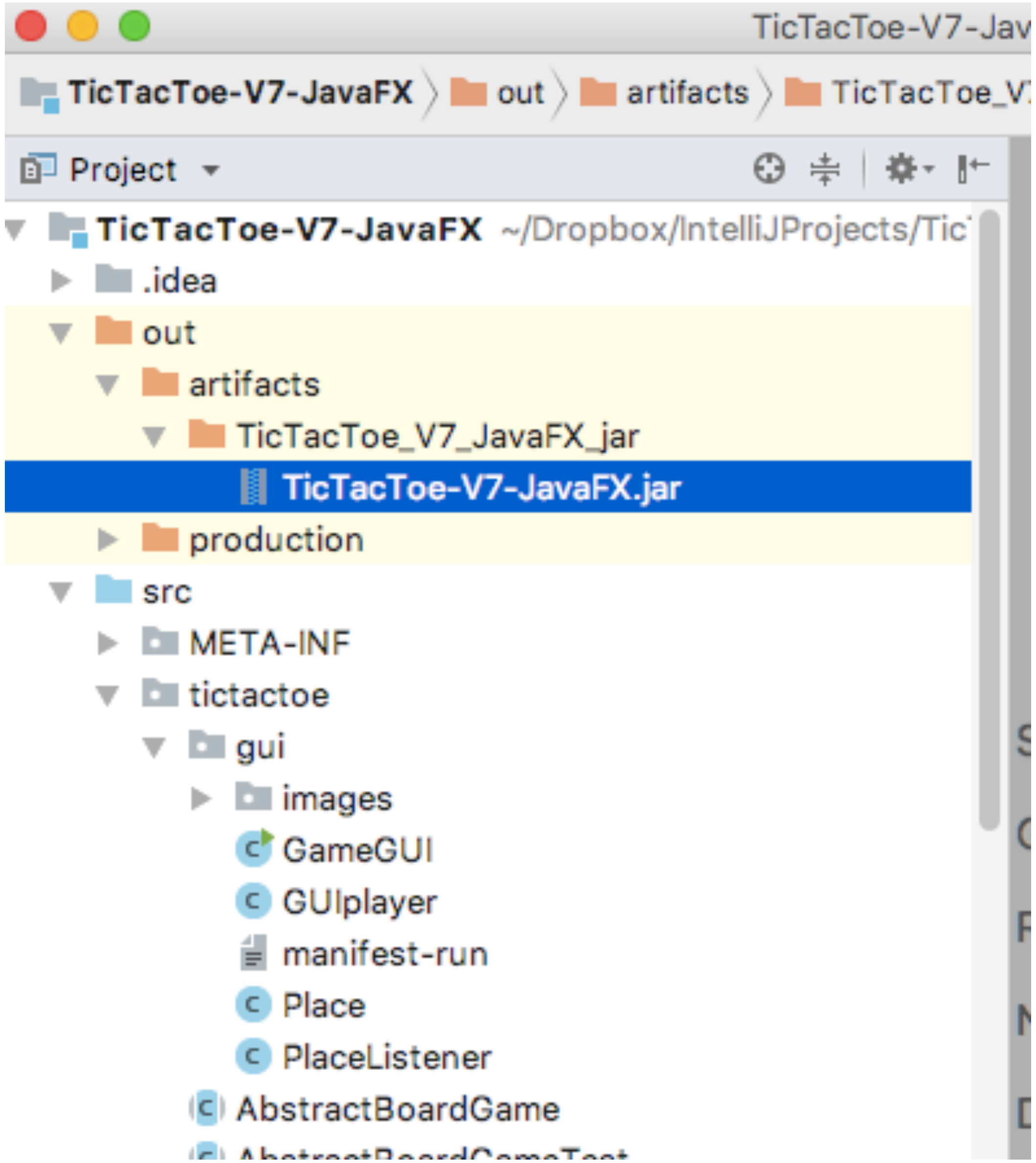
Show content of elements

IntelliJ

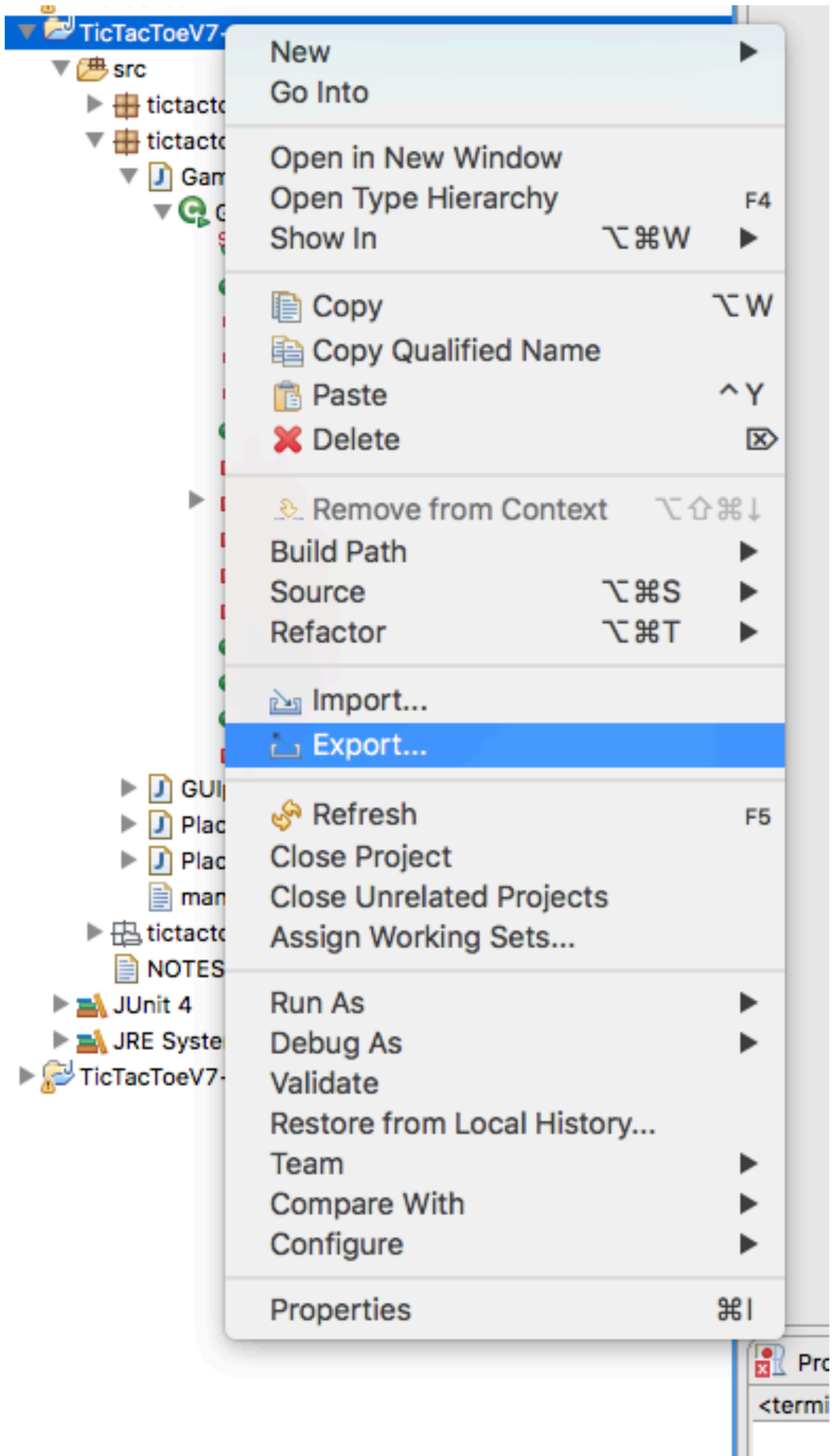

Refactor
Build
Run
Tools
VCS
Window
Help
~/Dropbox/Int
Build Project ⌘F9
Build Module 'TicTacToe-V7-JavaFX'
Rebuild ...TicTacToe-V7-JavaFX' ⇧⌘F9
Rebuild Project
Generate Ant Build...
Build Artifacts...
Run Ant Target ⇧⌘F10
neGUI


Build Artifact
TicTacToe-V7-JavaFX:jar
Action
Build
Rebuild
Clean
Edit...
Drop files here to open

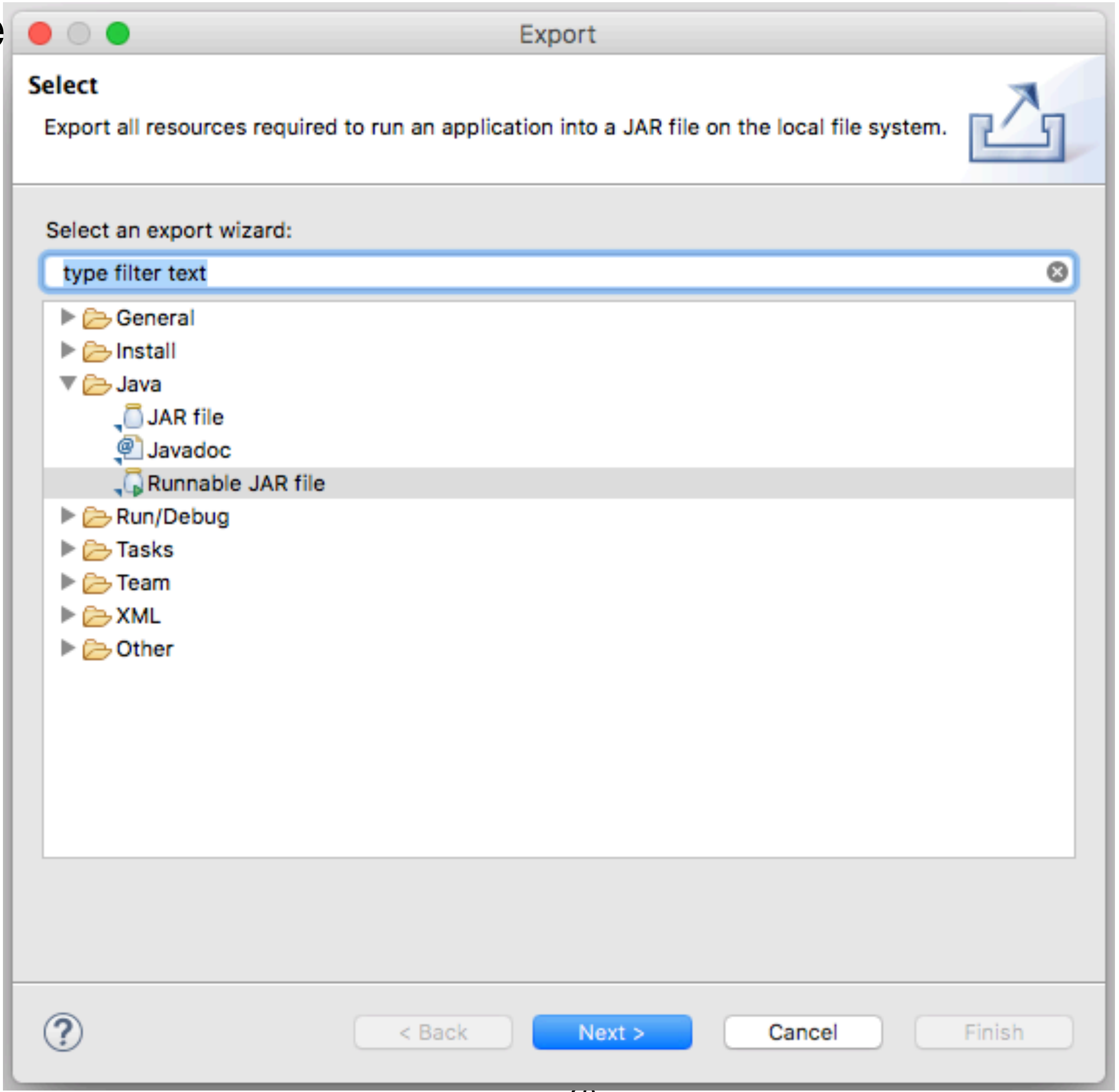
IntelliJ


TicTacToe-V7-Jav
TicTacToe-V7-JavaFX
out
artifacts
TicTacToe_V7
Project
TicTacToe-V7-JavaFX ~/Dropbox/IntelliJProjects/Tic
.idea
out
artifacts
TicTacToe_V7_JavaFX_jar
TicTacToe-V7-JavaFX.jar
production
src
META-INF
tictactoe
gui
images
GameGUI
GUIplayer
manifest-run
Place
PlaceListener
AbstractBoardGame

Eclipse


TicTacToeV7-
src
tictactc
tictactc
Gam
GUI
Plac
Plac
man
tictactc
NOTES
JUnit 4
JRE Syste
TicTacToeV7-
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌥W
Copy Qualified Name
Paste ^Y
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Validate
Restore from Local History...
Team
Compare With
Configure
Properties ⌘I
Pro
<termi

Eclipse



Export

Select

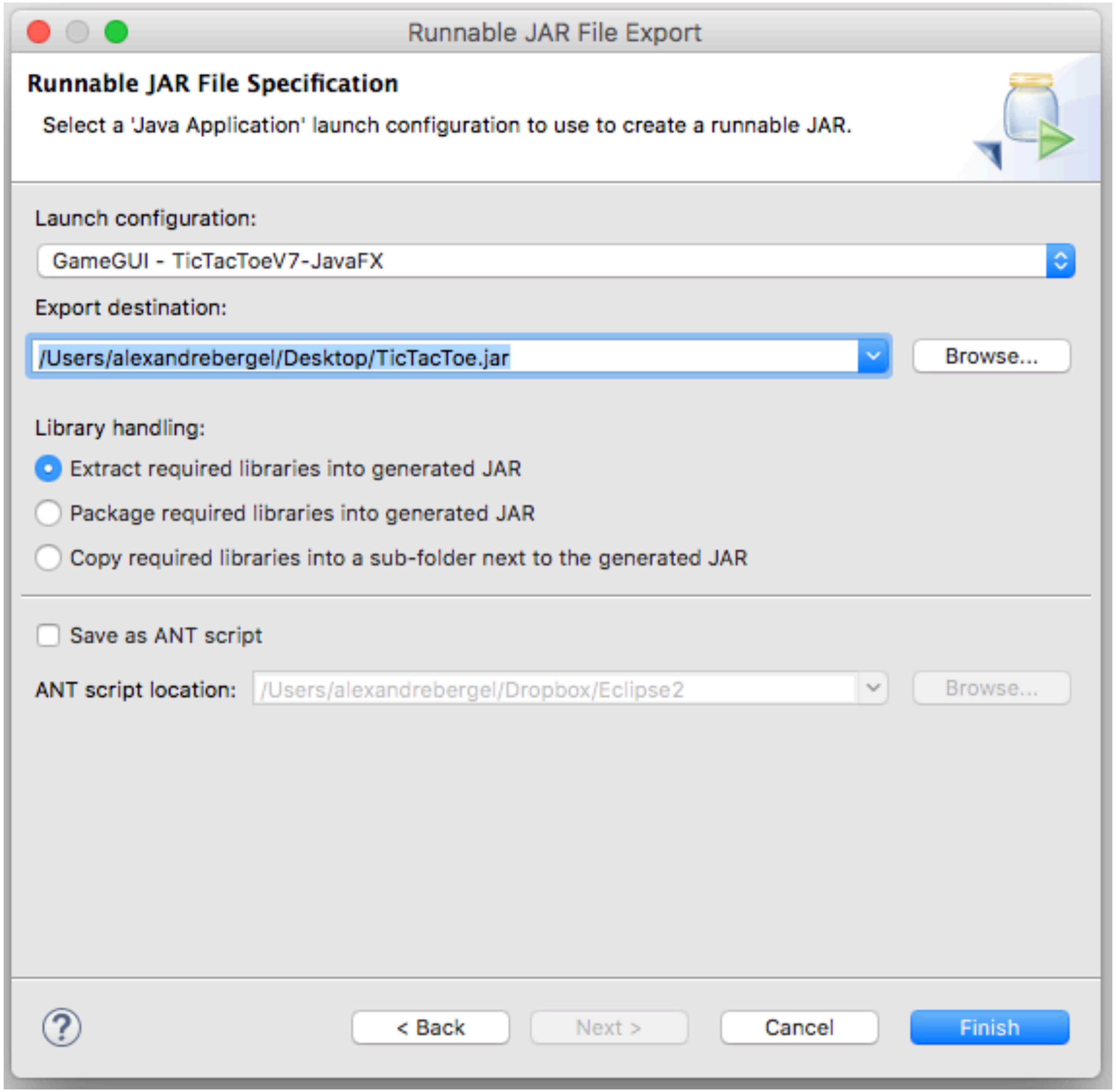
Export all resources required to run an application into a JAR file on the local file system.

Select an export wizard:

type filter text

- General
- Install
- Java
  - JAR file
  - Javadoc
  - Runnable JAR file
- Run/Debug
- Tasks
- Team
- XML
- Other

Eclipse


Runnable JAR File Export

Runnable JAR File Specification

Select a 'Java Application' launch configuration to use to create a runnable JAR.

Launch configuration: GameGUI - TicTacToeV7-JavaFX

Export destination: /Users/alexandrebergel/Desktop/TicTacToe.jar

Library handling:

- [✓] Extract required libraries into generated JAR
- [ ] Package required libraries into generated JAR
- [ ] Copy required libraries into a sub-folder next to the generated JAR

- [ ] Save as ANT script

ANT script location: /Users/alexandrebergel/Dropbox/Eclipse2

JAR file

You can now double click on the icon

No need to have IntelliJ or Eclipse to run your application

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