

18 Tic-Tac-Toe: GUI

Idee:

- ▶ Unbesetzte Felder werden durch **Button**-Komponenten repräsentiert.
- ▶ Die Knöpfe werden in einem (3×3) -Gitter angeordnet.
- ▶ Wird ein Zug ausgeführt, wird der entsprechende Knopf durch eine (bemale) **Canvas**-Fläche ersetzt (**Kreis** oder **Kreuz**).

GUI: Model – View – Controller

Modell (Model):

Repräsentiert das Spiel, den aktuellen Spielzustand, und die Spiellogik.

Ansicht (View)

Die externe graphische(?) Benutzeroberfläche mit der die Benutzerin interagiert.

Steuerung (Controller)

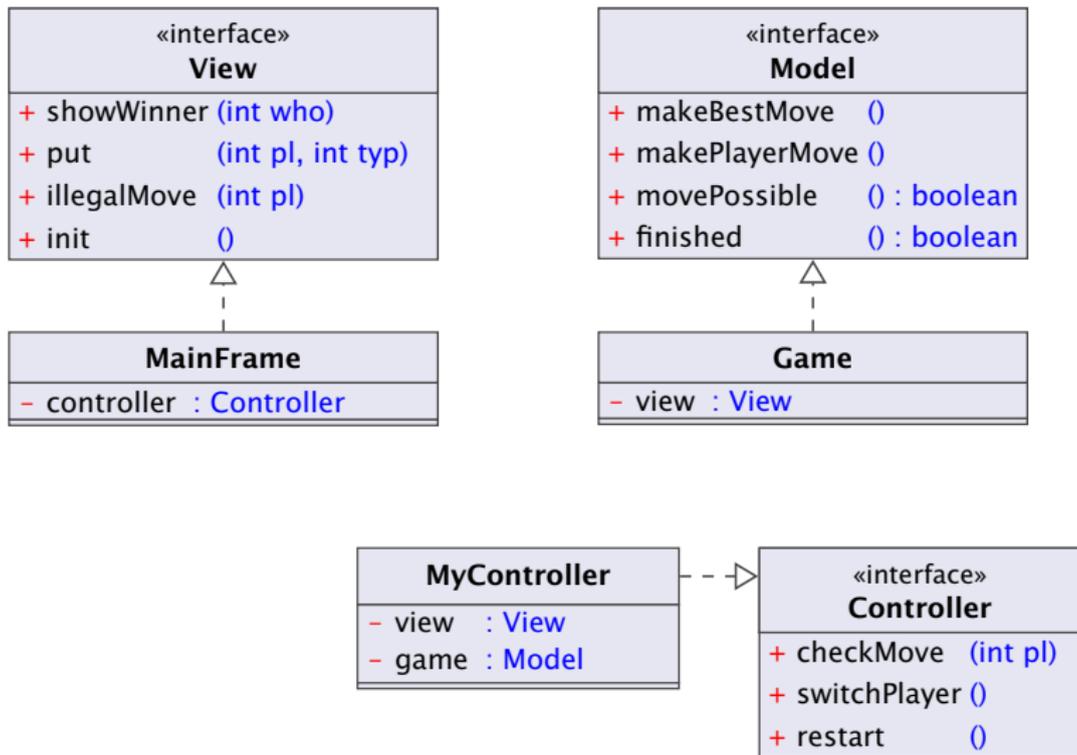
Kontrollschicht, die Aktionen der Nutzerin and die Spiellogik weiterleitet, und Reaktionen sichtbar macht.

Typisch für viele interaktive Systeme. Es gibt viele Varianten (Model-View-Presenter, Model-View-Adapter, etc.).

GUI: Model – View – Controller

- ▶ Es gibt viele solcher Standardvorgehensweisen, für das Strukturieren, bzw. Schreiben von großen Programmen (**Design Patterns**, ↑**Softwaretechnik**).
- ▶ Es gibt auch **Anti Patterns**, d.h., Dinge, die man normalerweise nicht tun sollte (die aber trotzdem häufig vorkommen).

TicTacToe - GUI



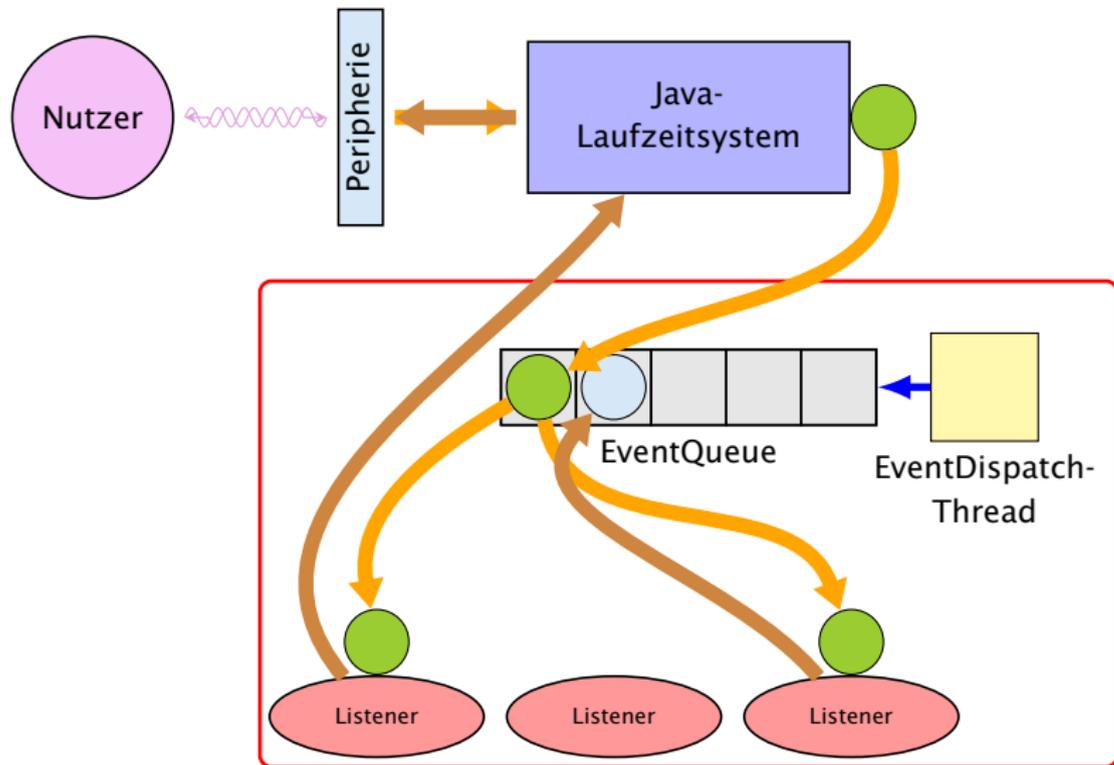
18.1 Hintergrund – GUIs

Eine graphische Benutzer-Oberfläche (**GUI**) ist i.A. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich) **intuitiven Dialog** mit der Benutzerin ermöglichen sollen.

Idee:

- ▶ Einzelne Komponenten bieten der Benutzerin Aktionen an.
- ▶ Ausführen der Aktionen erzeugt **Ereignisse**.
- ▶ Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht **Ereignis-basiertes Programmieren**.

Überblick – Events



Ereignisse

- ▶ Maus-Bewegungen und -Klicks, Tastatureingaben etc. werden von der Peripherie registriert und an das ↑Betriebssystem weitergeleitet.
- ▶ Das Java-Laufzeitsystem nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür AWTEvent-Objekte.
- ▶ Diese Objekte werden in eine AWTEventQueue eingetragen Producer!
- ▶ Die Ereignisschlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen. . .
- ▶ Der AWTEvent-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet Consumer!

Ereignisse

- ▶ Abarbeiten eines Ereignisses bedeutet:
 1. Weiterleiten des **AWTEvent**-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse **angemeldet** wurde;
 2. Aufrufen einer speziellen Methode des Listener-Objekts.
- ▶ Die Objekt-Methode des Listener-Objekts hat für die Reaktion des GUI zu sorgen.

Abarbeitung von Events erfolgt sequentiell.

Ein Button

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.swing.border.*;
5
6 public class MyButton extends JButton {
7     private int number;
8     public MyButton(int i) {
9         number = i;
10        setBackground(Color.WHITE);
11        setBorder(new LineBorder(Color.WHITE, 0));
12    }
13    public int getNumber() { return number; }
14 }
```

- ▶ `MyButton` erweitert die Klasse `JButton` um die Objektvariable `int number`
- ▶ `number` wird vom Konstruktor gesetzt und mit `int getNumber()`; abgefragt.

Der View

```
8 public class MainFrame extends JFrame implements
9                               PlayConstants,View {
10     private Controller controller;
11     private JDialog dia;
12     private Container c;
13
14     public MainFrame(Controller con) {
15         controller = con;
16         c = getContentPane();
17         setVisible(true);
18         init();
19     }
```

- ▶ Wenn man ein Objekt dieser Klasse anlegt öffnet sich ein Fenster, da die Klasse von `JFrame` abgeleitet ist.
- ▶ `c` ist der Hauptcontainer für graphische Elemente.
- ▶ `init` initialisiert die graphischen Komponenten für ein (neues) Spiel.

Der View

`pack()` computes the size of the frame, depending on its contents and the `LayoutManager`.

```
20     public void init() { invokeLater(()->{
21         c.removeAll();
22         c.setBackground(Color.BLACK);
23         c.setLayout(new GridLayout(3,3,3,3));
24         c.setPreferredSize(new Dimension(600,600));
25         for (int i=0; i<9;i++) {
26             MyButton b = new MyButton(i);
27             b.addActionListener( this::buttonAction );
28             c.add(b); }
29         pack(); } });}
```

- ▶ Die Funktion `void invokeLater(Runnable r)` speichert ein Objekt `r` in der `EventQueue`, das später vom `EventDispatchThread` ausgeführt wird.
- ▶ Jede Operation, die die GUI verändert, muß vom `EventDispatchThread` ausgeführt werden.

Der View

- ▶ `removeAll()` entfernt alle graphischen Elemente des Containers;
- ▶ Wir wählen `GridLayout` als `Layout-Manager`.
- ▶ `GridLayout(int row, int col, int cs, int rs);` konstruiert ein `GridLayout`-Objekt mit `row` Zeilen, `col` Spalten sowie Abständen `cs` und `rs` zwischen Spalten bzw. Zeilen.
- ▶ Wir fügen 9 durchnummerierte `MyButton`-Elemente ein.

Lambda-Ausdrücke

Ein **funktionales Interface** ist ein Interface, das **genau** eine Methode enthält.

```
interface Runnable {  
    void run();  
}
```

Ein **Lambda-Ausdruck** ist das Literal eines Objektes, das ein funktionales Interface implementiert. Z.B.:

Syntax:

- ▶ allgemein

```
(%Parameterliste) -> {...}
```

- ▶ nur **return**-statement/eine Anweisung (bei **void**-Funktion)

```
(%Parameterliste) -> %Ausdruck
```

- ▶ nur **genau** ein Parameter

```
a -> {...}
```

Die formalen Parameter eines Lambda-Ausdrucks, müssen keine Typbezeichner enthalten. Der notwendige Typ wird durch den Kontext bei der Verwendung des Lambda-Ausdrucks ermittelt.

Beispiele

Bei Namenskollisionen verhalten sich die beiden Codestücke leicht unterschiedlich. Insbesondere falls der Name `Foo` schon verwendet wird...

```
Runnable r = () -> {System.out.println("Hello!");};
```

ist (im Prinzip) äquivalent zu

```
class Foo implements Runnable {  
    void run() {  
        System.out.println("Hello!");  
    }  
}  
Runnable r = new Foo();
```

Methodenreferenzen

An der Stelle, an der ein Lambda-Ausdruck möglich ist, kann man auch eine **Methodenreferenz** einer passenden Methode angeben.

Beispiel:

- ▶ Klasse `ClassA` verfügt über statische Methode `boolean less(int a, int b)`.
- ▶ Das **Funktionsinterface** `Iface` verlangt die Implementierung einer Funktion, die zwei `ints` nach `boolean` abbildet.
- ▶ Außerdem existiert Funktion `sort(int[] a, Iface x)`.
- ▶ Dann sortiert der Aufruf:

```
int[] arr = {5,8,7,2,11};  
sort(arr, ClassA::less);
```

gemäß der durch `less` vorgegebenen Ordnung.

Wenn man eine Objektmethode benutzen möchte müsste man `x::less` schreiben, wobei `x` ein Objekt der Klasse `ClassA` ist.

Vorausgesetzt die Funktion `sort` ist so implementiert, dass sie tatsächlich sortiert...

Button – Events

- ▶ Wenn ein Knopf gedrückt wird, wird ein `ActionEvent` ausgelöst, d.h., ein Objekt dieser Klasse in die Queue eingefügt.
- ▶ Ein Objekt, das auf solch ein Ereignis reagieren soll muss das Interface `ActionListener` implementieren.

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

- ▶ Die Methode `actionPerformed()` erhält das Objekt zu dem auslösenden Ereignis als Parameter.
- ▶ Wir konstruieren das Listenerobjekt über eine Methodenreferenz.

Button - Events

```
54     public void buttonAction(ActionEvent e) {
55         MyButton button = (MyButton) e.getSource();
56         int place = button.getNumber();
57         controller.checkMove(place);
58     }
```

- ▶ Falls ein Knopf gedrückt wird, überprüft der `controller` ob der entsprechende Zug möglich ist.
- ▶ Die Spiellogik (überprüfen des Zuges) ist von der Visualisierung bzw. der Userinteraktion getrennt.

View – Interfacemethoden

```
30 public void put(int place, int type) {
31     invokeLater(()->{
32         JPanel canvas;
33         if (type == MIN) canvas = new Cross();
34         else canvas = new Circle();
35         c.remove(place);
36         c.add(canvas, place);
37         revalidate();
38         repaint();
39     });
40 }
41 public void illegalMove(int place) {
42     System.out.println("Illegal move: "+place);
43 }
```

"MainFrame.java"

View – Interfacemethoden

effectively final bedeutet, dass man den Parameter als **final** deklarieren könnte, ohne dass es zu einem Compilerfehler führt.

```
44 public void showWinner(int who) {
45     String str = "";
46     switch(who) {
47         case -1: str = "Kreuz gewinnt!"; break;
48         case 0: str = "Unentschieden!"; break;
49         case 1: str = "Kreis gewinnt!"; break;
50     }
51     final String s = str;
52     invokeLater(()->dia = new MyDialog(this,s));
53 }
```

- ▶ Lokale, innere Klassen dürfen auf Parameter der umgebenden Funktion zugreifen, aber nur wenn diese **effectively final** sind.
- ▶ Durch den Lambdaausdruck wird implizit eine lokale, innere Klasse erzeugt.

Ein Dialog

```
1 public class MyDialog extends JDialog {
2     JButton repeat, kill;
3     public MyDialog(MainFrame frame, String str) {
4         setSize(300,70);
5
6         setLayout(new FlowLayout());
7         add(new JLabel(str));
8         repeat = new JButton("new");
9         repeat.addActionListener(frame::dialogAction);
10        add(repeat);
11        kill = new JButton("kill");
12        kill.addActionListener(frame::dialogAction);
13        add(kill);
14        setVisible(true);
15    }
16 }
```

Ein `JLabel`-Objekt wird genutzt um den Text anzuzeigen.

Der Konstruktor erhält einen anzuzeigenden Text (`str`) und eine Referenz auf das Objekt in dem die `ActionListener`-Methoden definiert sind.

Wir fügen zwei Knöpfe hinzu, die beide die gleiche `ActionListener`-Funktion aufrufen.

ActionListener des Dialogs

```
59     public void dialogAction(ActionEvent e) {
60         JButton b = (JButton) e.getSource();
61         if (e.getActionCommand().equals("kill")) {
62             System.exit(0);
63         } else {
64             controller.restart();
65             dia.dispose();
66         }
67     }
```

"MainFrame.java"

- ▶ `getActionCommand()` gibt per default den Text des zugehörigen `JButtons` zurück.
- ▶ `dispose()` löscht das Dialogfenster.

Controller - Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MyController implements PlayConstants,
5                                     Controller {
6     private Model game;
7     private View view;
8     public void setup(Model m, View v) {
9         game = m; view = v;
10    }
```

"MyController.java"

Controller – Methoden

```
11     public void checkMove(int place) {
12         if (game.movePossible(place)) {
13             game.makePlayerMove(place);
14         }
15         else view.illegalMove(place);
16     }
17     public void switchPlayer() {
18         if (game.finished()) return;
19         game.makeBestMove();
20     }
21     public void restart() {
22         view.init();
23         game = new Game(view);
24 } }
```

"MyController.java"

Main

```
68     public static void main(String[] args) {
69         invokeLater()->{
70             Controller c = new MyController();
71             View v = new MainFrame(c);
72             Model m = new Game(v);
73             c.setup(m,v);
74         });
75     }
```

"MainFrame.java"

Was ist hier falsch?

Was passiert wenn wir einen sehr grossen Spielbaum berechnen?

Die eingefügten Befehle simulieren eine Baumberechnung die ziemlich lange dauert. Dies verdeutlicht das Problem.

```
16     private void initTree() {
17         try {Thread.sleep(4000);}
18         catch(InterruptedException e) {}
19         g.nodeCount = 0;
20         g = new GameTreeNode(p);
21         System.out.println("generate tree... (" +
22                             g.nodeCount + " nodes)");
23     }
```

"GameWithDelay.java"

Die GUI reagiert nicht mehr, da wir die gesamte Berechnung im **Event Dispatch Thread** ausführen.

View - ActionListener

```
54     public void buttonAction(ActionEvent e) {
55         MyButton button = (MyButton) e.getSource();
56         int place = button.getNumber();
57         ctrl.exec(()->ctrl.checkMove(place));
58     }
59     public void dialogAction(ActionEvent e) {
60         JButton b = (JButton) e.getSource();
61         if (e.getActionCommand().equals("kill")) {
62             System.exit(0);
63         } else {
64             ctrl.exec(()->ctrl.restart());
65             dia.dispose();
66         }
67     }
```

"MainFrameNew.java"

Die neue Methode `exec` im Controller startet ein übergebenes `Runnable`-Objekt in einem speziellen Controllerthread. Falls dieser Thread gerade aktiv ist (z.B. mit einer Spielbaumberechnung) passiert gar nichts.

Controller – Attributes

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.util.concurrent.locks.*;
4
5 public class MyController
6     extends Thread
7     implements PlayConstants,
8         Controller {
9     private Model game;
10    private View view;
11    final Lock lock = new ReentrantLock();
12    final Condition cond = lock.newCondition();
```

"MyControllerNew.java"

Die geänderte Klasse `MyController` repräsentiert jetzt einen eigenen `Thread` und enthält Variablen für die Koordination.

Controller – Methoden

`exec` versucht durch `lock.trylock()` zu erhalten. Falls dieses nicht gelingt blockiert diese Funktion nicht, sondern kehrt einfach zum Aufrufer zurück.

```
14     Runnable r = null;
15     public void exec(Runnable r) {
16         if (lock.tryLock()) {
17             if (r != null) this.r = r;
18             cond.signal();
19             lock.unlock();
20         } }
21     public void run() {
22         lock.lock(); try {
23             while (true) {
24                 while (r == null)
25                     cond.await();
26                 r.run();
27                 r = null;
28             }}
29     catch (InterruptedException e) {}
30     finally { lock.unlock(); }
31 }
```

Die `run()`-Methode des Threads läuft in einer Endlosschleife. Wenn dem `runnable`-Objekt etwas zugewiesen wird, wird die zugehörige Methode ausgeführt.

Controller – Methoden

```
32     public void setup(Model m, View v) {  
33         game = m; view = v;  
34         start();  
35     }
```