- Das Interface `PlayConstants` fasst durchgängig benutzte Konstanten zusammen.
- `YOU` $<$ `NONE` $<$ `ME` repräsentieren die drei möglichen Ausgänge eines Spiels bzw. Belegungen eines Felds des Spielbretts (-1, 0 und 2).
- Die Klasse `PlayGround` repräsentiert Spielbretter.
- Das aktuelle Spielbrett wird aus demjenigen des Vater-Knotens und dem von `ME` gewählten Zug gestimmt.
- `MyChoice answers[choice]` enthält den Teilbaum für Zug `choice`.
- Das Objekt `PossibleMoves moves` enumeriert alle aktuell freien Positionen für Steine (mindestens eine :-)

  Gewinnt der Zug `choice`, wird `value = YOU` gesetzt.
  Andernfalls wird `value` auf das Minimum des Werts des Unterbaums `answers[choice]` und des alten Werts gesetzt.

```
   ...
   public int value() { return value;}
   public MyChoice answer(int place) {
      return answer[place];
   }
}  // end of class YourChoice
```

- `public int value();` liefert den Spielwert des aktuellen Spielbaums.
- `public MyChoice answer(int place);` liefert den Spiel-Teilbaum für den Zug `place`.

```
public class MyChoice implements PlayConstants {
   private PlayGround ground;
   private YourChoice yours;
   private int acceptableChoice, value;
   public MyChoice(PlayGround g, int place) {
      ground = new PlayGround(g,place,YOU);
      if (ground.won(place,YOU)) { value = YOU; return; }
      acceptableChoice = ground.force(ME);
      if (acceptableChoice != -1) { value = ME; return; }
      acceptableChoice = ground.force(YOU);
      if (acceptableChoice != -1) {
         yours = new YourChoice(ground,acceptableChoice);
         value = yours.value(); return;
      }
      ...
```

- Die Objekt-Methode `int force(int who);` liefert einen Zug, mit dem `who` unmittelbar gewinnt – sofern ein solcher existiert :-), andernfalls `-1`.
- Der Aufruf `ground.force(ME)` ermittelt einen solchen Gewinnzug für das Programm.
- Falls kein Gewinnzug existiert, liefert der Aufruf `ground.force(YOU)` eine Position, mit der der Gegner in einem Zug gewinnen könnte (was verhindert werden muss :-)
- Nur wenn keiner dieser Fälle auftritt, überprüfen wir sämtliche Zug-Möglichkeiten ...

```
      ...
      PossibleMoves moves = new PossibleMoves(ground);
      int tmp = moves.next();
      if (tmp == -1) { value = NONE; return;}
      value = YOU; do { acceptableChoice = tmp;
         yours = new YourChoice(ground,acceptableChoice);
         if (yours.value() > YOU) { value = yours.value(); return; }
         tmp=moves.next();
      } while (tmp != -1);
   }
   public int value() { return value;}
   public int acceptableChoice() { return acceptableChoice; }
   public MyChoice select(int place) {
      return yours.answer(place);
   }
}  // end of class MyChoice
```

- Bei der Iteration über alle Zug-Möglichkeiten geben wir uns bereits mit einem akzeptablen Zug zufrieden, genauer gesagt: dem ersten, der mindestens unentschieden liefert.

Der Grund:

als nachziehender können wir (bei Tic-Tac-Toe und gegen einen optimalen Gegner) nichts besseres erreichen :-(

- `public int value();` liefert wieder den Spiel-Wert des Spielbaums.
- `public int acceptableChoice()` liefert einen akzeptablen Zug und
- `public MyChoice select(int place)` liefert bei gegebenem Spielzug für den akzeptablen Zug und die Antwort `place` des Gegners den nächsten Teilbaum.

22.3 Die Klasse `Game`

Die Klasse `Game` sammelt notwendige Datenstrukturen und Methoden zur Durchführung eines Spiels.

```
public class Game implements PlayConstants {
   public PlayGround ground = new PlayGround();
   private int count = 0;
   private MyChoice gameTree;
   public int nextMove(int place) {
      if (count == 1)
           gameTree = new MyChoice(ground,place);
      else gameTree = gameTree.select(place);
      return gameTree.acceptableChoice();
   }
   ...
```

- `PlayGround ground` enthält die jeweils aktuelle Spiel-Konfiguration.
- `int count` zählt die Anzahl der bereits ausgeführten Züge.
- `MyChoice gameTree` enthält eine Repräsentation der `ME`-Strategie.
- `public int nextMove(int place)` liefert den Antwort-Zug auf den `YOU`-Zug `place`:
  - War `place` der erste `YOU`-Zug, wird ein `MyChoice`-Objekt `gameTree` für das verbleibende Spiel angelegt.
  - Bei jedem weiteren Zug wird `gameTree` auf die gemäß `place` ausgewählte Teil-Strategie gesetzt.
  - Den nächsten `ME`-Zug liefert jeweils `acceptableChoice` in `gameTree`.

```
    ...
    public boolean possibleMove(int place) {
       if (count%2 == 1) return false;
       return (ground.free(place));
    }
    public boolean finished() { return (count == 9);}
    public void move(int place,int who) {
       count++;
       ground.move(place,who);
    }
}  // end of class Game
```

Weitere nützliche Methoden:

- `boolean possibleMove(int place)` überprüft, ob der Zug `place` überhaupt möglich ist;
- `public boolean finished()` überprüft, ob bereits 9 Züge gespielt wurden;
- `public void move(int place,int who)` erhöht den Zug-Zähler und führt den Zug auf dem Spielfeld `ground` aus.

22.4 Die graphische Benutzer-Oberfläche

Idee:

- Unbesetzte Felder repräsentieren wir durch `Button`-Komponenten.
- Die Knöpfe werden in einem (3×3)-Gitter angeordnet.
- Wird ein Zug ausgeführt, wird der entsprechende Knopf durch eine (bemalte) `Canvas`-Fläche ersetzt.

```
import java.awt.*;
import java.awt.event.*;
class MyButton extends Button {
   private int number;
   public MyButton(int i) {
      number = i;
      setBackground(Color.white);
   }
   public int getNumber() { return number;}
}  // end of class MyButton
```

- `MyButton` erweitert die Klasse `Button` um die Objekt-Variable `int number`.
- `number` wird vom Konstruktor gesetzt und mit `int getNumber();` abgefragt.

```
public class Arena extends Frame {
   private MoveObserver moveObserver;
   public Arena() {
      setBackground(Color.blue);
      setSize(300,300); setLocation(400,400);
      setLayout(new GridLayout(3,3,3,3)); start();
   }
   public void start() {
      removeAll();
      moveObserver = new MoveObserver(this);
      for (int i=0; i<9;i++) {
         MyButton b = new MyButton(i);
         b.addActionListener(moveObserver); add(b);
      }
   }
```

- `Arena` ist die Unterklasse von `Frame`, in der wir das Spielfeld anlegen.
- Dazu wählen wir das `GridLayout` als Layout-Manager.
- `public GridLayout(int row, int col, int cs, int rs);` konstruiert ein `GridLayout`-Objekt mit `row` Zeilen, `col` Spalten sowie den Abständen `cs` und `rs` zwischen den Spalten bzw. Zeilen.
- Zu der Arena fügen wir neun (durch-numerierte) `MyButton`-Komponenten hinzu.
- Gemeinsamer `ActionListener` für sämtliche Knöpfe ist das `MoveObserver`-Objekt `moveObserver`.
- `public void removeAll();` beseitigt sämtliche Komponenten eines Containers.

```
    ...
    public void move(int place, Canvas canvas) {
       remove(place); add(canvas, place);
    }
    public static void main(String[] args) {
       Arena arena = new Arena(); arena.setVisible(true);
    }
}   //end of class Arena
```

- `public void move(int place, Canvas canvas);` ersetzt die Komponente an der Stelle `place` durch die `Canvas`-Komponente `canvas`.
- `public void remove(int i);` beseitigt die `i`-te Komponente eines `Container`-Objekts.
- `public void add(Component c, int i);` fügt die Komponente `c` als `i`-te Komponente ein.

22.5 Beantwortung eines Zugs

Beachte:

- Ziehen der Tic-Tac-Toe-Spielerin heißt Drücken eines Knopfs.
- Drücken eines Knopfs löst einen `ActionEvent` aus.
- Die Klasse `MoveObserver` verarbeitet die Ereignisse.
- Aufgabe eines `MoveObserver`-Objekts ist es, das Spielfeld zu modifizieren und einen Antwort-Zug zu liefern, d.h. zu finden und dann anzuzeigen.

```
import java.awt.*;
import java.awt.event.*;
public class MoveObserver implements
                            ActionListener, PlayConstants {
   private Game game; private Arena frame;
   public MoveObserver(Arena a) {
      game = new Game(); frame = a;
   }
   public void actionPerformed(ActionEvent e) {
      MyButton button = (MyButton) e.getSource();
      int place = button.getNumber();
      if (!game.possibleMove(place)) return;
      frame.move(place,new Cross());
      game.move(place,YOU);
      frame.setVisible(true);
      ...
```

- Wurde das Feld `place` ermittelt, das die Gegnerin setzte, wird der Knopf an dieser Stelle durch ein neues `Cross`-Element ersetzt.
- Dann wird der Zug auch auf der internen Repräsentation des Spielbretts auseführt.
- `frame.setVisible(true);` macht diese Änderung auf dem Bildschirm sichtbar.

```
...
if (game.ground.won(place,YOU)) {
   game.ground.clear();
   (new MyDialog(frame,"You won ...")).setVisible(true);
   return;
}
if (game.finished()) {
   (new MyDialog(frame,"No winner ...")).setVisible(true);
   return;
}
place = game.nextMove(place);
frame.move(place,new Circle());
game.move(place,ME);
frame.setVisible(true);
...
```

- Die Objekt-Methode `public void clear();` der Klasse `PlayGround` verhindert weitere Züge, indem sämtliche Felder auf `3` gesetzt werden (Hack – sorry!).
- Falls die Spielerin `YOU` gewonnen hat, beenden wir das Spiel und erzeugen ein Dialog-Fenster der Klasse `MyDialog`.
- Ähnlich verfahren wir, wenn das Spiel nicht verloren, aber zuende ist ...
- Ist das Spiel weder verloren noch zuende, bestimmen wir unseren nächsten Zug.
- Diesen führen wir auf unserer Intern-Darstellung des Spielbretts wie in dem `Arena`-Frame aus.

```
        ...
        if (game.ground.won(place,ME)) {
           game.ground.clear();
           (new MyDialog(frame,"I won ...")).setVisible(true);
           return;
        }
     }
}  // end of class MoveObserver
```

- Falls unser Zug zum Gewinn führte, beenden wir das Spiel und erzeugen ein `MyDialog`-Fenster.