Objektorientierte Programmierung

Studiengang Medieninformatik

Hans-Werner Lang

Hochschule Flensburg

Vorlesung 10
17.05.2017

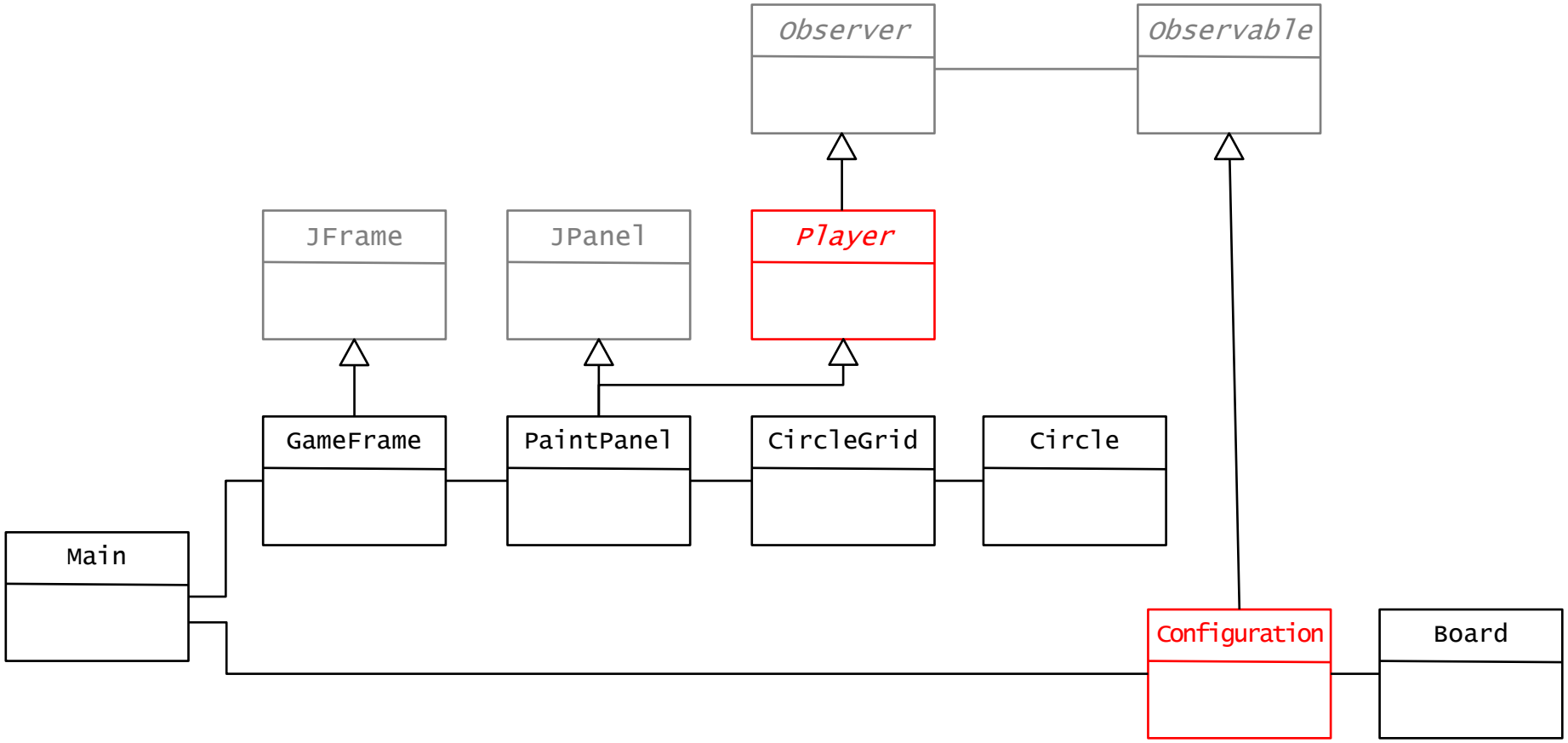
Hochschule Flensburg
University of Applied Sciences

Heute:

- Spielen mit zwei Spielern
- Gewinnstellung erkennen
- Computerspieler *SimplePlayer*

Klassendiagramm *Vier Gewinnt*

Hochschule Flensburg
University of Applied Sciences



Interface *Player*

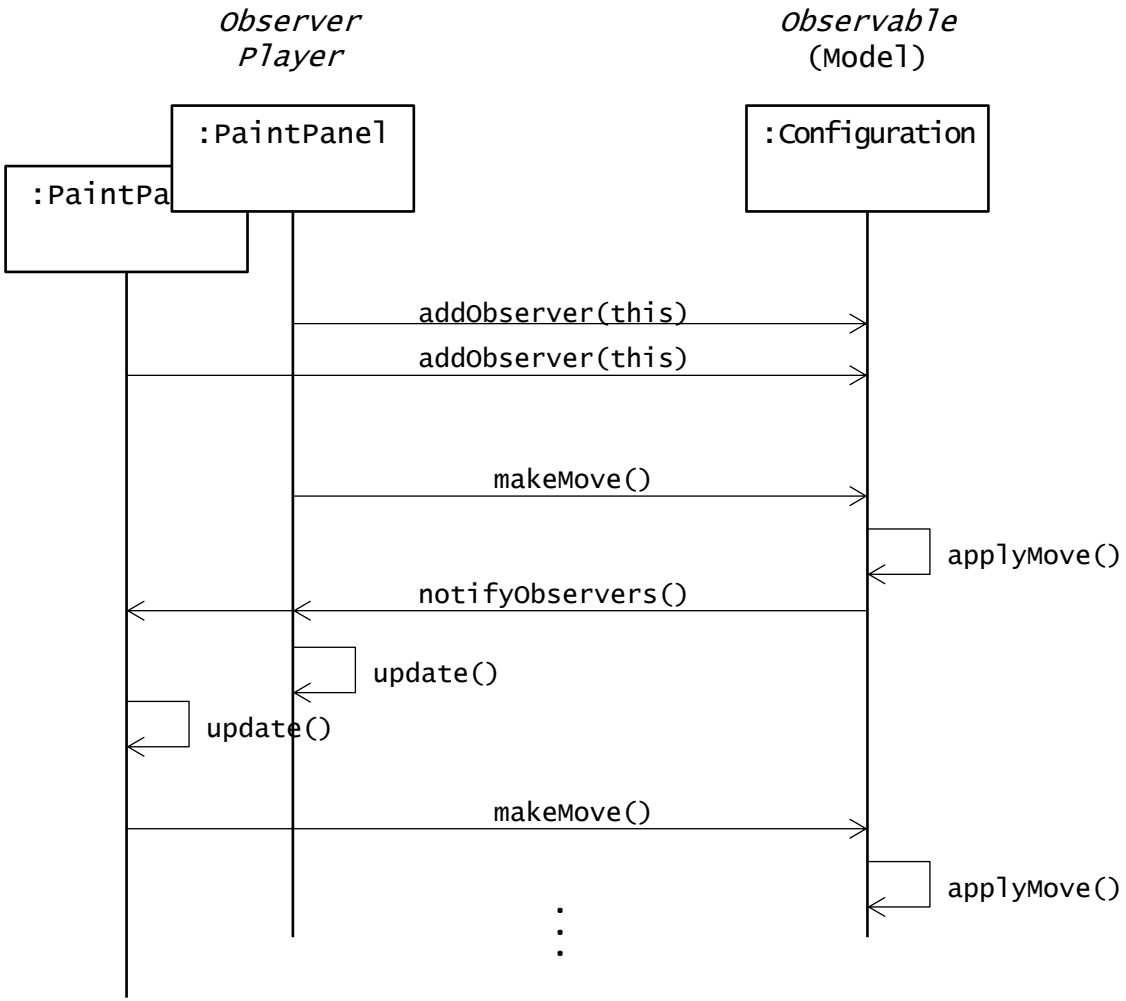
Hochschule Flensburg
University of Applied Sciences

```
public interface Player extends Observer
{
    // Farbe des Spielers festlegen (0 oder 1):
    public void setColor(int k);

    // die Spielstellung festlegen, in der gespielt wird:
    public void setConfiguration(Configuration config);

    // einen Spielzug ausführen:
    public void makeMove(Move move);
}
```

Sequenzdiagramm



Spielen mit zwei Spielern...

Hochschule
Flensburg
University of
Applied Sciences

Gewinnstellung erkennen (1)

Hochschule Flensburg
University of Applied Sciences

Vorgehensweise: Vom zuletzt gesetzten Spielstein ausgehen und prüfen, ob hierdurch ein Vierer der entsprechenden Farbe zustande gekommen ist.

D.h. die Anzahl gleichfarbiger benachbarter Steine in Nord- und Südrichtung bestimmen, in West- und Ostrichtung, in Nordwest- und Südostrichtung, sowie in Nordost- und Südwestrichtung.

Gewinnstellung erkennen (2)

Hilfreich ist eine Methode

```
public Position add(Position other)
```

in der Klasse *Position*. Mit

```
Position direction=new Position(0, 1);
```

ergibt

```
this.add(direction)
```

den Nachbarn in Ostrichtung. Hilfreich ist ferner eine Methode

```
public Position neg()
```

in der Klasse *Position*. Mit

```
Position opposite_direction=direction.neg();
```

erhalten wir die zu *direction* entgegengesetzte Richtung.

Hilfreich ist ferner eine Methode

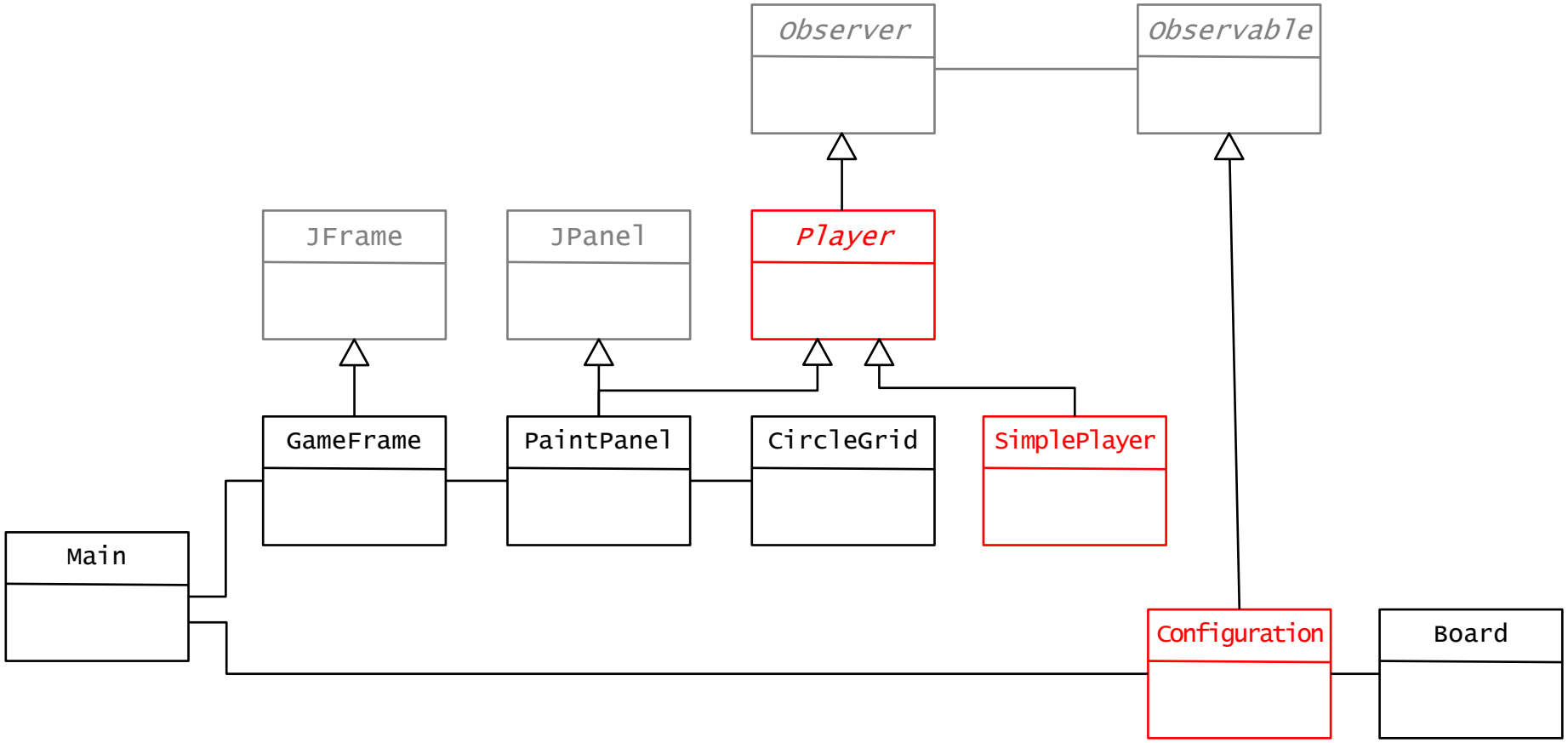
```
public boolean isValidPosition(Position p)
```

in der Klasse *Board*, um zu verhindern, dass wir über die Grenzen des Spielbretts hinausgehen.

Klassendiagramm *Vier Gewinnt*

(erweitert um *SimplePlayer*)

Hochschule Flensburg
University of Applied Sciences



Klasse *SimplePlayer* (1)

Hochschule Flensburg
University of Applied Sciences

Die Klasse *SimplePlayer* muss im wesentlichen die abstrakten Methoden des Interfaces *Player* und des übergeordneten Interfaces *Observer* implementieren:

Methoden aus *Player*:

```
public void setColor(int k);
public void setConfiguration(Configuration config);
public void makeMove(Move move);
```

Methode aus *Observer*:

```
public void update(Observable o, Object arg);
```

Klasse *SimplePlayer* (2)

Hochschule Flensburg
University of Applied Sciences

Und die Klasse *SimplePlayer* muss eine Methode *findMove* enthalten, die nach einer bestimmten Strategie ihren nächsten Spielzug bestimmt und an *makeMove* übergibt.

Eine einfache Spielstrategie besteht darin, den nächsten Spielzug zufällig zu wählen:

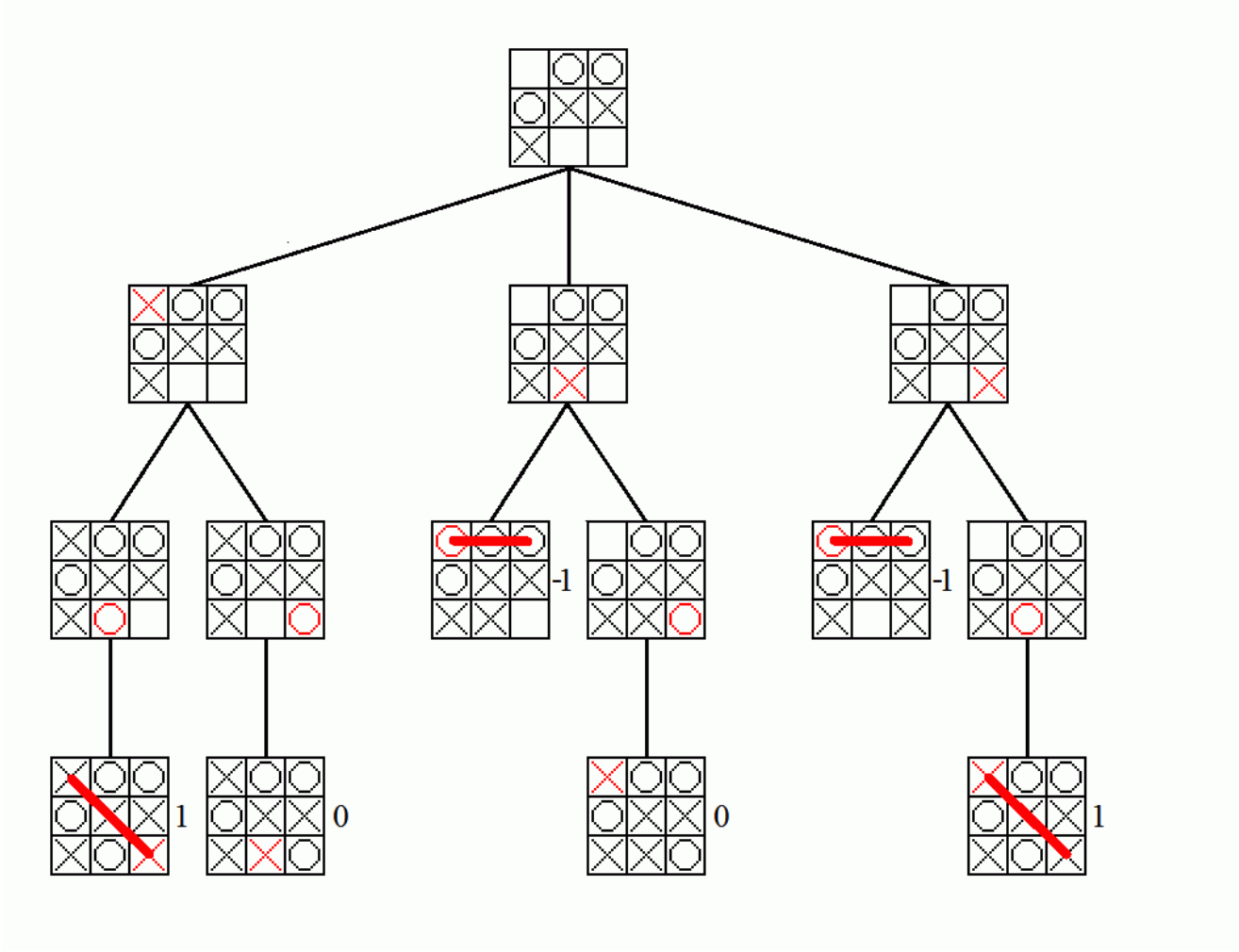
```
// simple Spielstrategie: irgendwohin setzen
public Move findMove()
{
    int j;
    Move move=null;
    while (!config.board.isValidMove(move))
    {
        j=(int)(Math.random()*Board.COLS);  // zufällige Spalte
        move=new Move(j, mycolor);
    }
    return move;
}
```

Hochschule Flensburg
University of Applied Sciences

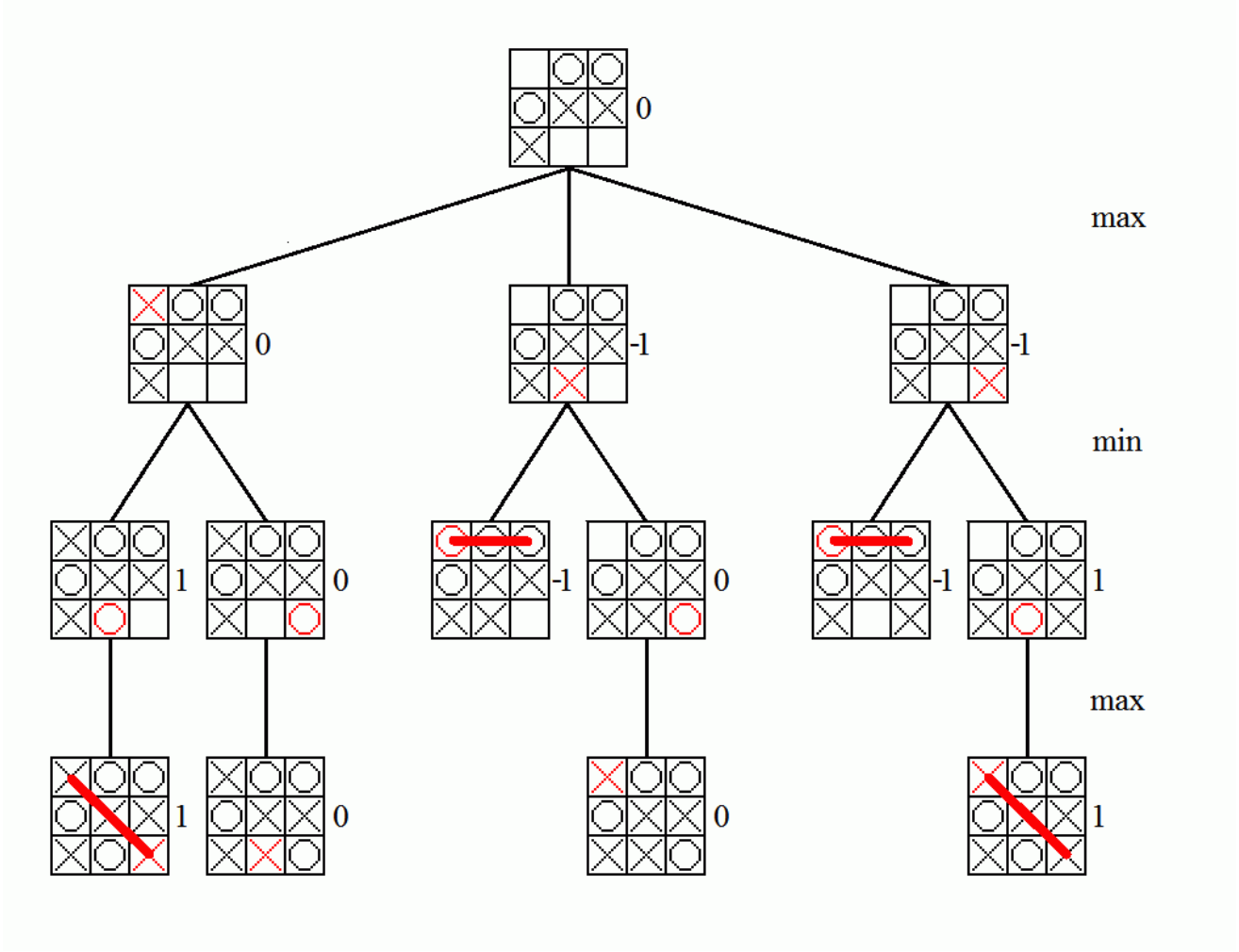
Beim nächsten Mal:

- Bessere Spielstrategie: Spielbaum auswerten, d.h. eine bestimmte Anzahl von Spielzügen vorausrechnen

Vorschau Spielbaum-Auswertung (1)



Vorschau Spielbaum-Auswertung (2)



Vorschau Programm Spielbaum-Auswertung

Hochschule Flensburg
University of Applied Sciences

```
private double eval(Configuration s, int d)
{
    if (d<=0 || s.gameOver())
        return rate(s);

    Iterator<Move> it=s.nextMoveIterator();
    Maximizer<Move> max=new Maximizer<Move>();
    Move m;
    Configuration t;
    while (it.hasNext())
    {
        m=it.next();
        t=s.copy();
        t.applyMove(m);
        max.add(-eval(t, d-1), m);
    }
    return max.getMaxVal();
}
```