

- Mithilfe der Objekt-Methoden

```
void setLocation(int x, int y);  
void setSize(int width, int height);
```

... kann ein Fenster positioniert bzw. dimensioniert werden.

- Der Standard-LayoutManager der Klasse Frame ist BorderLayout. Für diesen fügt `void add(Component c)` die Komponente `c` in der Mitte ein (sind die Ränder unbesetzt, breitet sich die Mitte über die ganze Fläche aus :-)
- Die Klassen-Methode `main()` legt zwei `Basic`-Objekte an verschiedenen Stellen des Bildschirms an ...

Achtung: Um auf Schließen des Fensters adequat reagieren zu können, empfiehlt es sich, einen `WindowListener` für das Frame zu implementieren !!!

Mögliche Anordnungen von Komponenten:

Frage: was passiert, wenn das Fenster (evt.) redimensioniert wird?

Frage: Wie kann man erreichen, dass das Erscheinungsbild exakt unserer Vorstellung entspricht???

Flexible Proportionierung mithilfe eines LayoutManagers:

- Das Layout der Komponenten passt sich der Größe der zur Verfügung stehenden Fläche an :-)
- Knöpfe vergrößern sich unförmig oder wandern von einer Zeile in die nächste :-(
- Die eigene Vorstellung muss (evt. relativ mühsam) mithilfe vorgefertigter Manager realisiert werden :-{

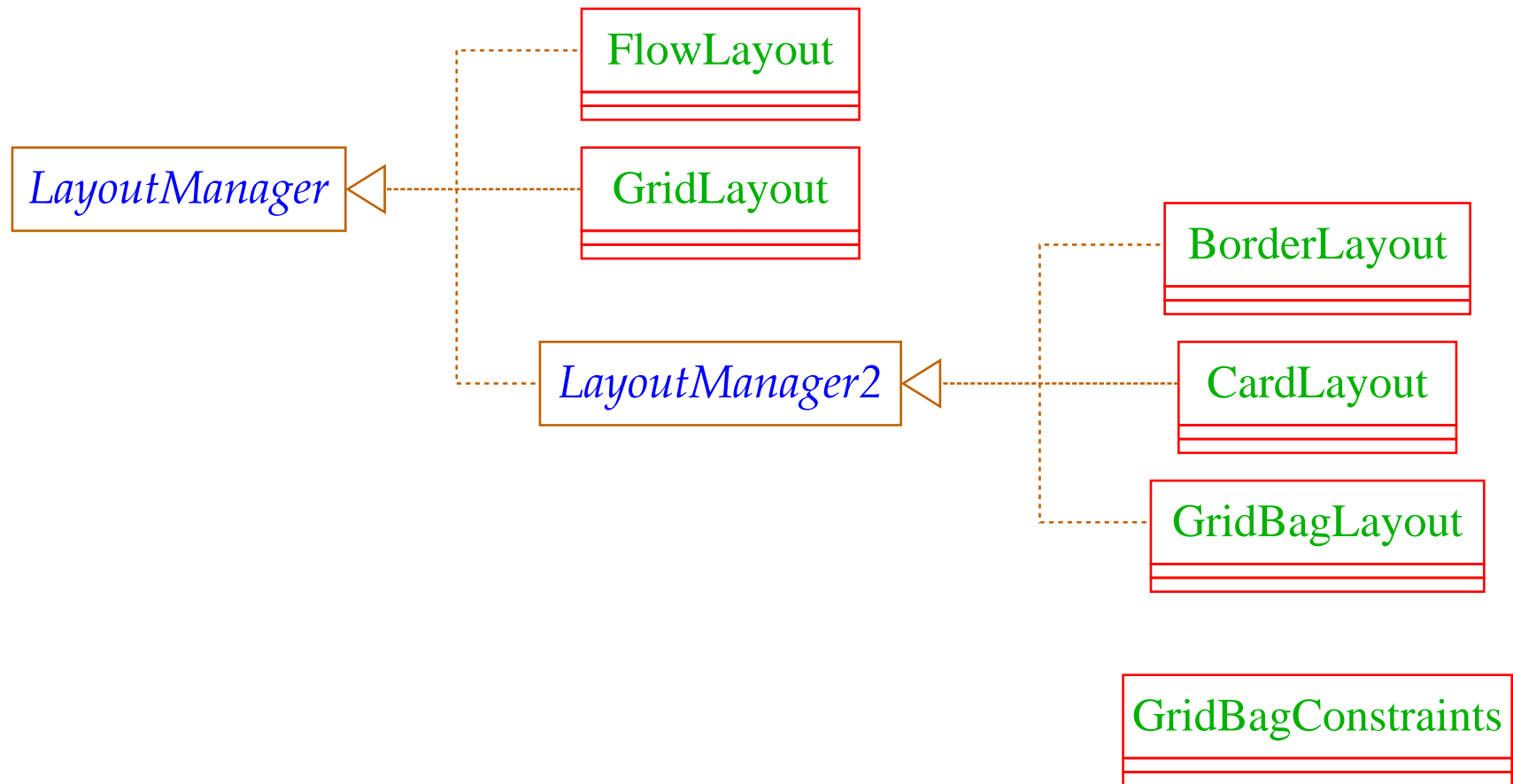
Absolute Positionierung und Dimensionierung:

- ... kennen wir bereits für Bildschirm-Fenster und graphische Elemente einer Fläche.
- Die belegte Fläche erhält bei Fenster-Deformierung einen unbenutzten Rand oder wird (teilweise) unsichtbar :-)
- Um sie für Komponenten in Container-Objekten zu ermöglichen, kann man mittels `setLayout(null);` den aktuellen Layout-Manager **ausschalten** :-)
- Sukzessive Aufrufe von `void add(Component c);` erzeugen einen **Stack** von Komponenten, deren Größe und Position mittels

```
public void setSize(int width, int height);
public void setLocation(int x, int y);
```

... modifiziert werden kann :-}

Einige vorgefertigte Layout-Manager:



FlowLayout: Das Default-Layout der Klasse `Panel`. Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

BorderLayout: Das Default-Layout der Klasse `Window`. Die Fläche wird in die fünf Regionen `North`, `South`, `West`, `East` und `Center` aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

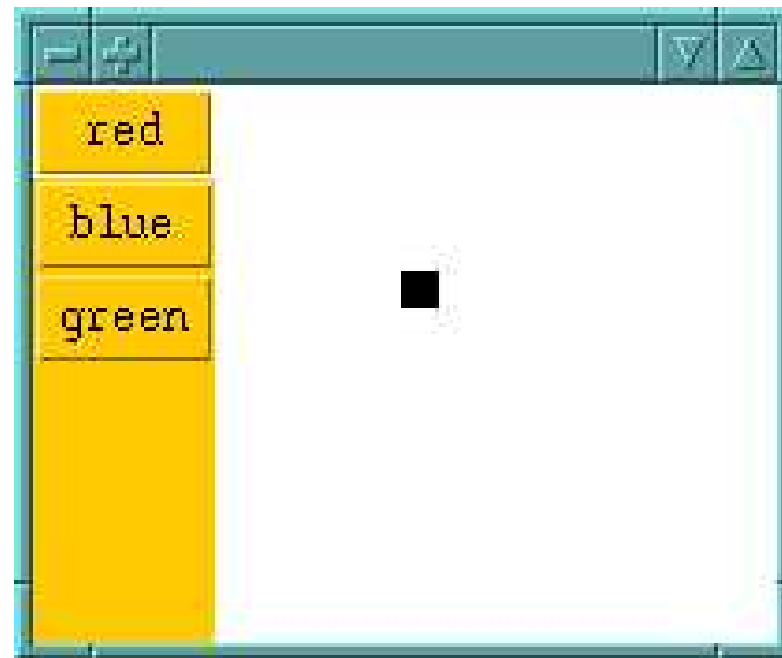
GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

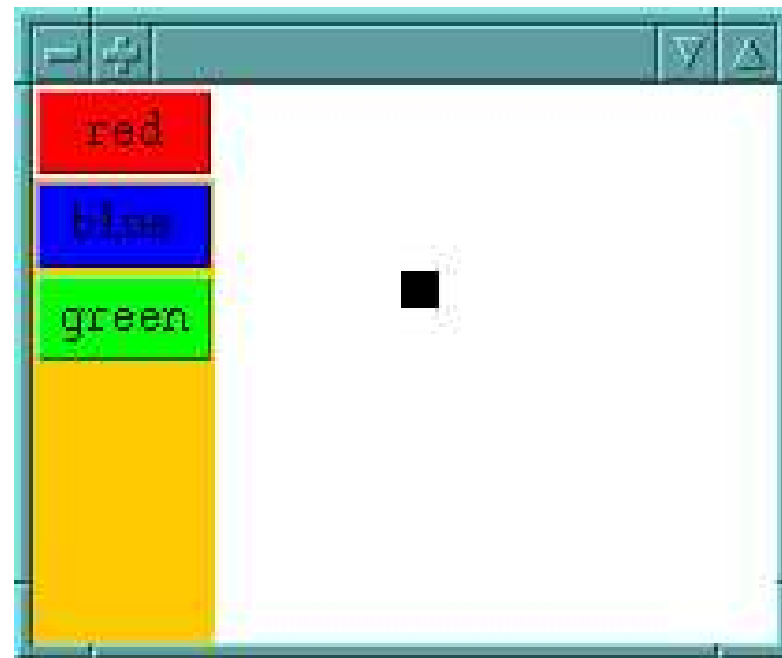
GridBagLayout: Wie `GridLayout`, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Beispiel:

Ziel:

- Eine orange Knopfleiste am linken Rand;
 - drei Knöpfe in der oberen Hälfte der Leiste;
 - auf Knopf-Druck soll die Farbe des Knopfs wechseln;
 - links daneben eine weiße Fläche mit einem schwarzen Quadrat
- ...





Implementierung:

- Die Knöpfe legen wir mittels `GridLayout` in ein `Panel`-Objekt.
- Die weiße Fläche mit schwarzem Quadrat malen wir auf ein `Canvas`-Objekt.
- Ein `Frame`-Objekt besitzt bereits das `BorderLayout`.
- Das `Panel`-Objekt legen wir im Westen des Frames ab, das `Canvas`-Objekt in der Mitte.

```
import java.awt.*;
import java.awt.event.*;
class MyPanel extends Panel implements ActionListener {
    Button b0, b1, b2;
    public MyPanel() {
        setBackground(Color.orange);
        b0 = new Button("red");
        b1 = new Button("blue");
        b2 = new Button("green");
        b0.addActionListener(this);
        b1.addActionListener(this);
        b2.addActionListener(this);
        setLayout(new GridLayout(6,1));
        add(b0); add(b1); add(b2);
    }
    ...
}
```

- Der Konstruktor `public GridLayout(int row, int col);` teilt die zur Verfügung stehende Fläche in ein Raster von gleich großen Feldern ein, die in `row` vielen Zeilen und `col` vielen Spalten angeordnet sind.
- die Felder werden sukzessive von links oben nach rechts unten aufgefüllt.
- Nicht alle Felder müssen tatsächlich belegt werden.
Im Beispiel bleibt die Hälfte frei ...
- Gemeinsamer `ActionListener` für alle drei Knöpfe ist (hier) das Panel selbst ...

```
public void actionPerformed(ActionEvent e) {
    Button b = (Button) e.getSource();
    if (b.getBackground() == Color.orange) {
        if (b == b0) b0.setBackground(Color.red);
        else if (b == b1) b1.setBackground(Color.blue);
        else b2.setBackground(Color.green);
    } else
        b.setBackground(Color.orange);
}
} // end class MyPanel
...
```

- Der Aufruf `e.getSource()`; liefert das **Objekt**, das das `ActionEvent`-Objekt erzeugte, hier ein `Button`-Objekt.
- Falls die Hintergrunds-Farbe orange ist, modifizieren wir die Farbe. Ansonsten setzen wir sie auf orange zurück.

```

public class Nest extends Frame {
    public Nest() {
        setSize(200,150); setLocation(500,500);
        add(new MyPanel(), "West");
        add(new MyCanvas(), "Center");
    }
    public static void main(String[] args) {
        (new Nest()).setVisible(true); }
} // end of class Nest

class MyCanvas extends Canvas {
    public MyCanvas() { setBackground(Color.white);}
    public void paint(Graphics page) {
        page.setColor(Color.black);
        page.fillRect(50,50,10,10);
    }
} // end of class MyCanvas

```

- Ein neues Canvas-Objekt besitzt eigentlich Breite und Höhe 0 — und ist damit **unsichtbar!**
- Im BorderLayout wird eine Komponente jedoch auf die gesamte zur Verfügung stehende Fläche ausgedehnt.
- Wie in der Klasse Applet wird zum (Neu-)Malen der Canvas-Fläche implizit die Objekt-Methode `public void paint(Graphics page);` aufgerufen.

Was es sonst noch so in Java gibt:

... an nützlichem:

innere Klassen;

... an mysteriösem:

Klassen zur Selbst-Reflektion;

... an comfortablem:

Malen mit Swing;

... an technischem:

Networking mit Sockets, RMI, Jini, Corba, ...

... an sonstigem:

... :-)

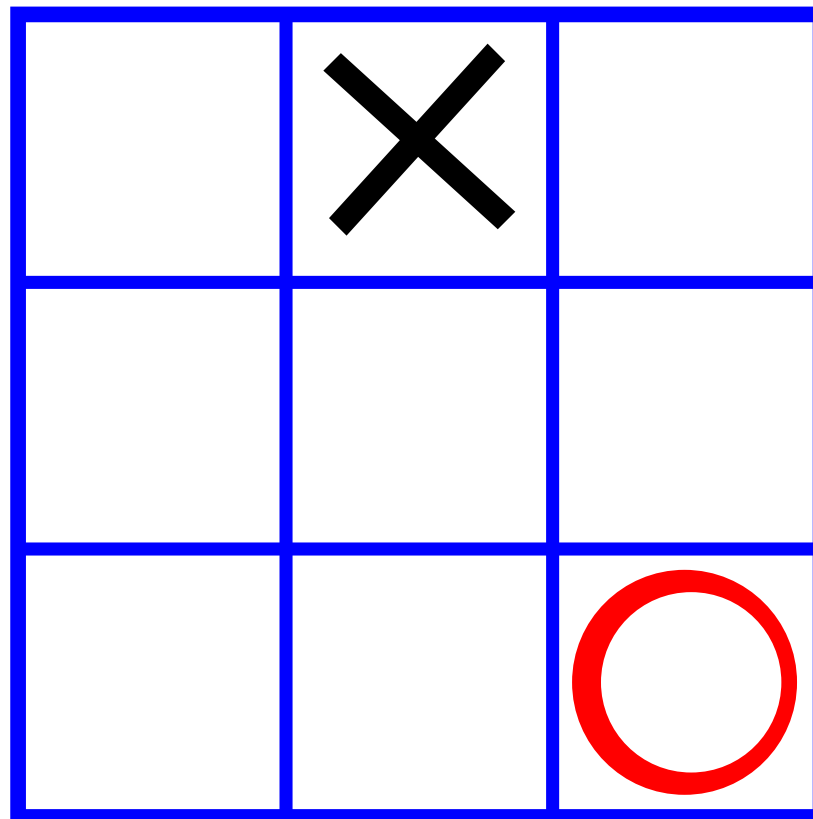
2 Tic-Tac-Toe

Regeln:

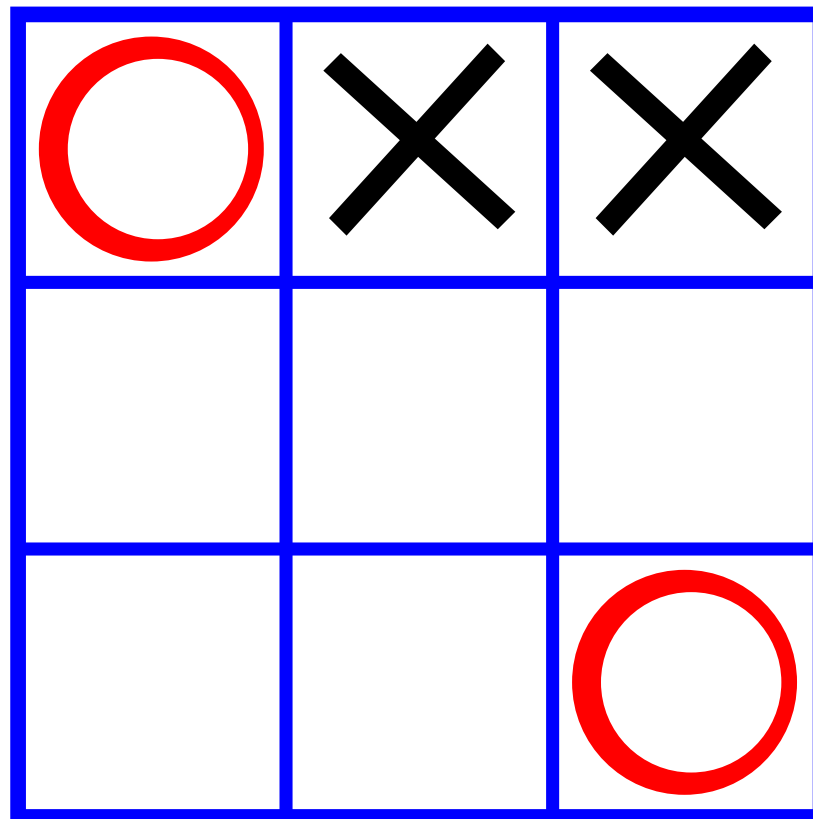
- Zwei Personen setzen abwechselnd **Steine** auf ein (3×3) -Spielfeld.
- Wer zuerst drei Steine in einer **Reihe** erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

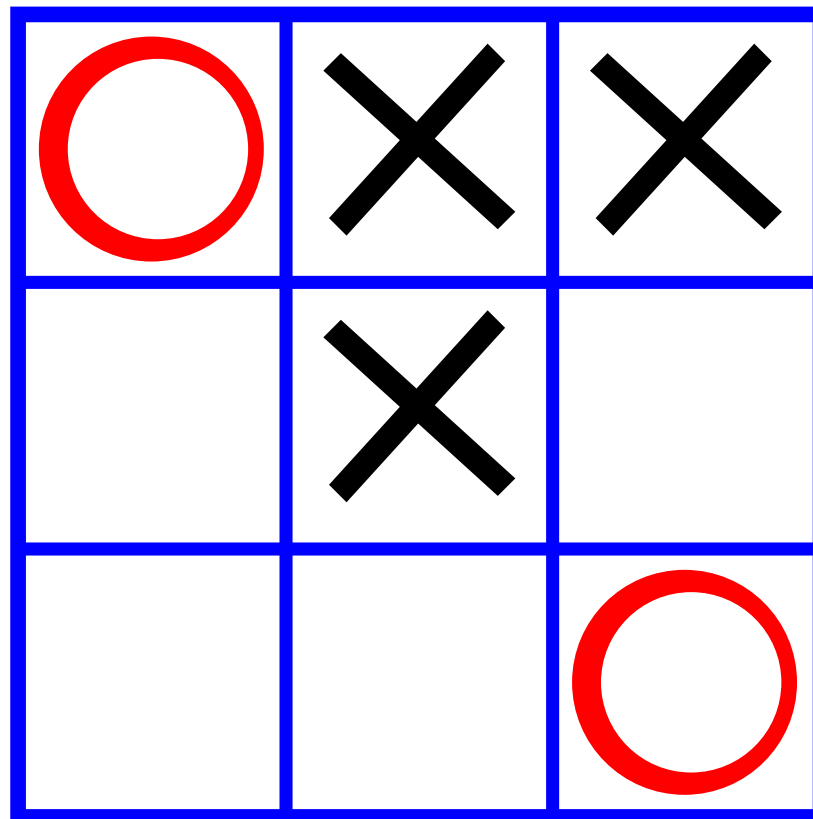
Beispiel:

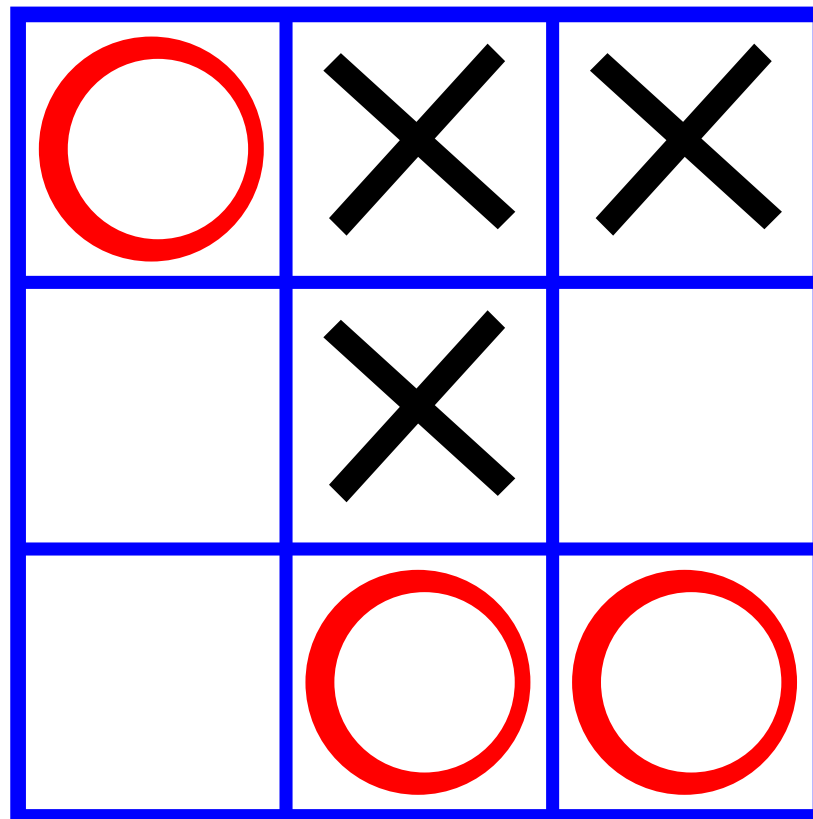
	X	

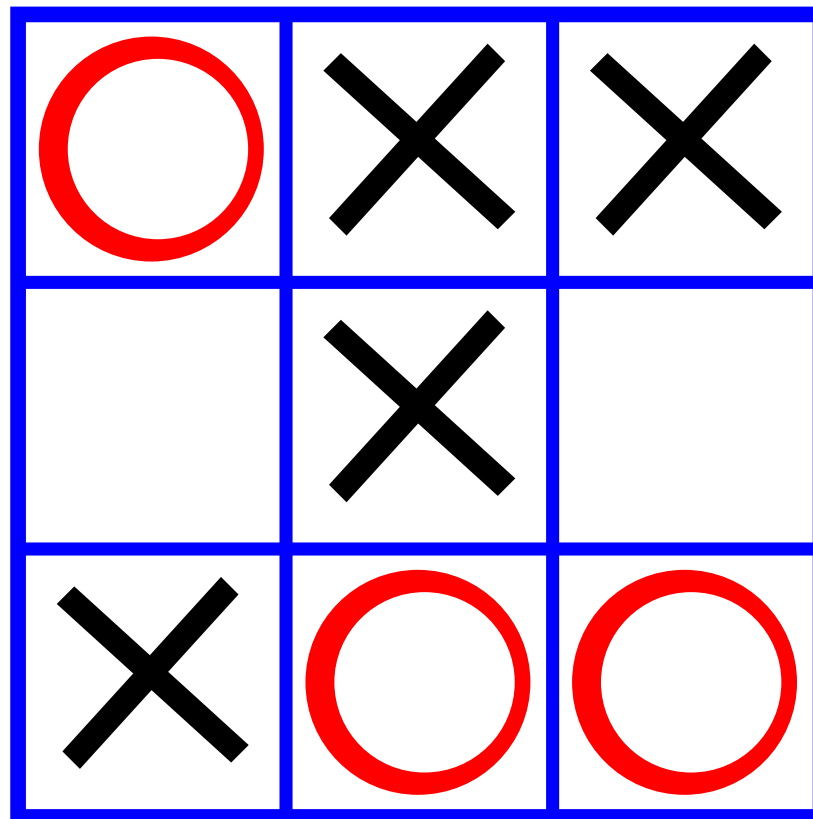


	X	X
		O









... offenbar hat die anziehende Partei gewonnen.

Fragen:

- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein **Tic-Tac-Toe**-Programm, das
 - ... möglichst oft gewinnt?
 - ... eine **ansprechende** Oberfläche bietet?

Hintergrund:

Tic-Tac-Toe ist ein endliches **Zwei-Personen-Nullsummen-Spiel**.

Das heißt:

- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von **Spiel-Konfigurationen**.
- Die Spieler ziehen abwechselnd. Ein **Zug** wechselt die Konfiguration, bis eine **End-Konfiguration** erreicht ist.
- Jede End-Konfiguration ist mit einem **Gewinn** aus \mathbb{R} bewertet.
- Person 0 hat gewonnen, wenn eine End-Konfiguration erreicht wird, deren Gewinn negativ ist, Person 1, wenn er positiv ist.

... im Beispiel:

Konfiguration:

	×	
		○

End-Konfigurationen:

○	×	×
	×	
○	×	○

Gewinn -1

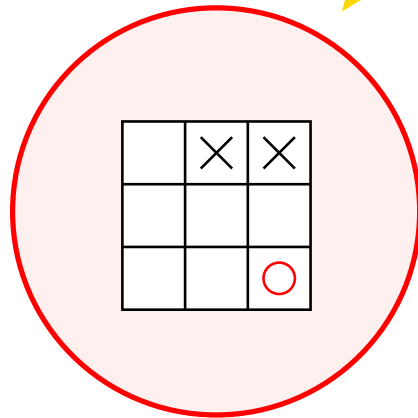
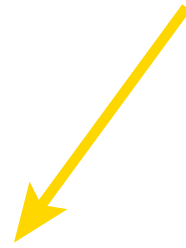
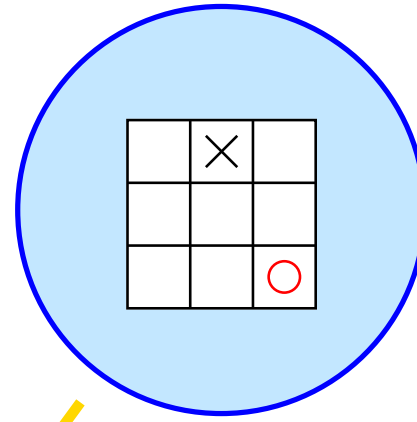
○	×	×
×	○	○
○	×	×

Gewinn 0

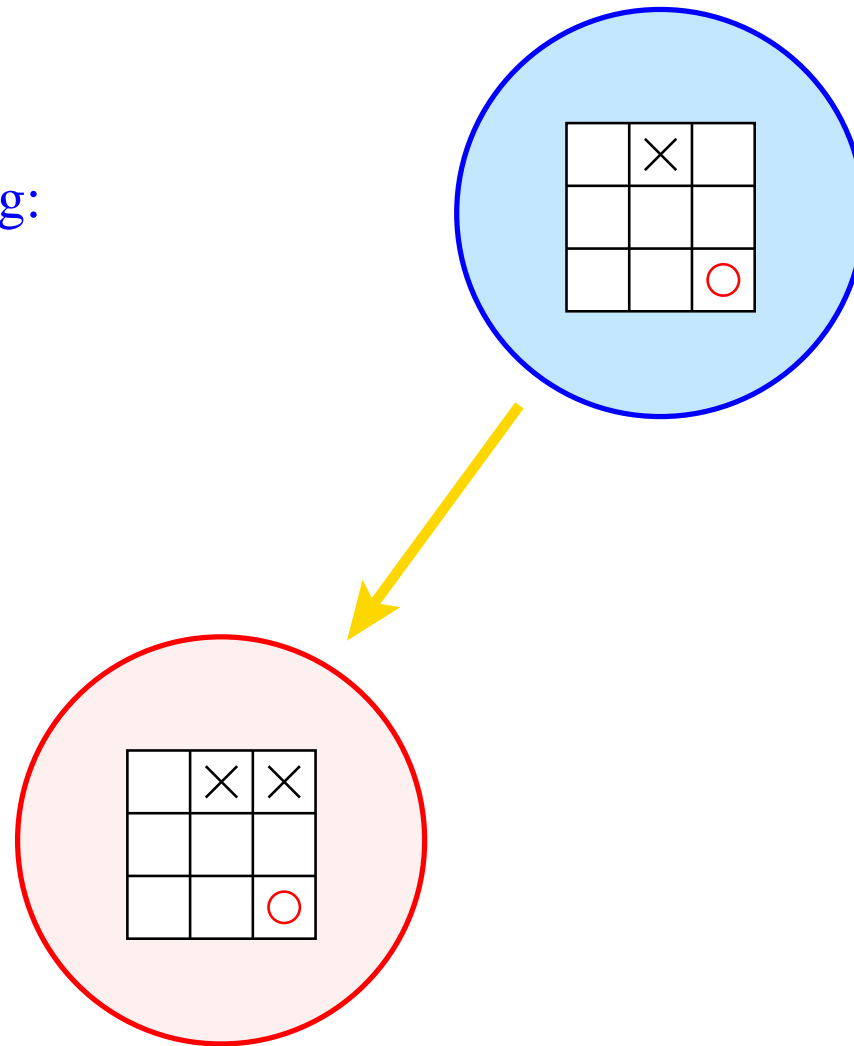
○	×	×
○	○	○
	×	×

Gewinn +2

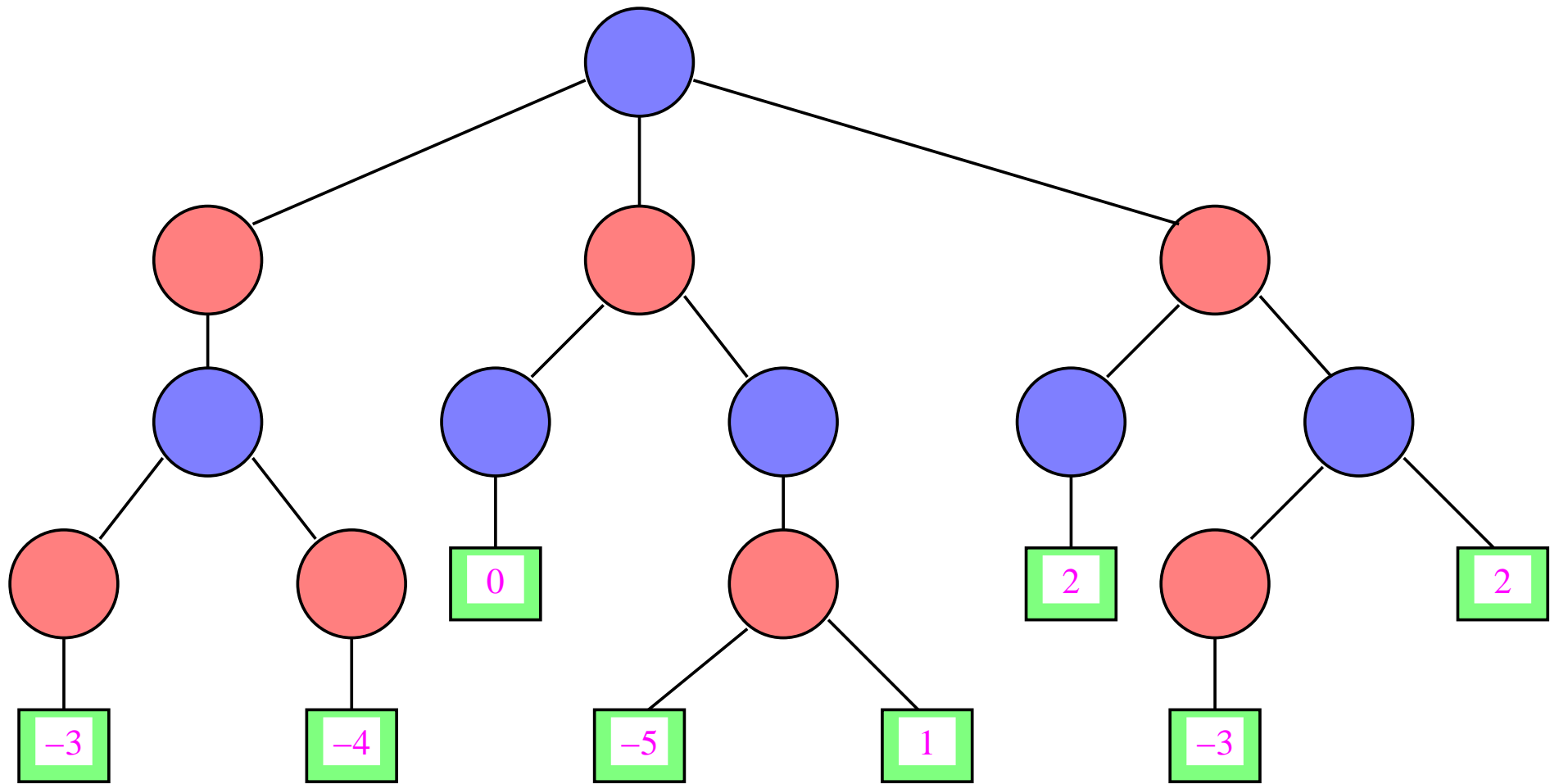
Spiel-Zug:



Spiel-Zug:



Alle möglichen Spiel-Verläufe lassen sich als ein Baum darstellen:



Knoten des Spielbaums	==	Konfigurationen
Kanten	==	Spiel-Züge
Blätter	==	End-Konfigurationen

Frage:

Wie finden wir (z.B. als **blaue** Person) uns im Spiel-Baum zurecht?

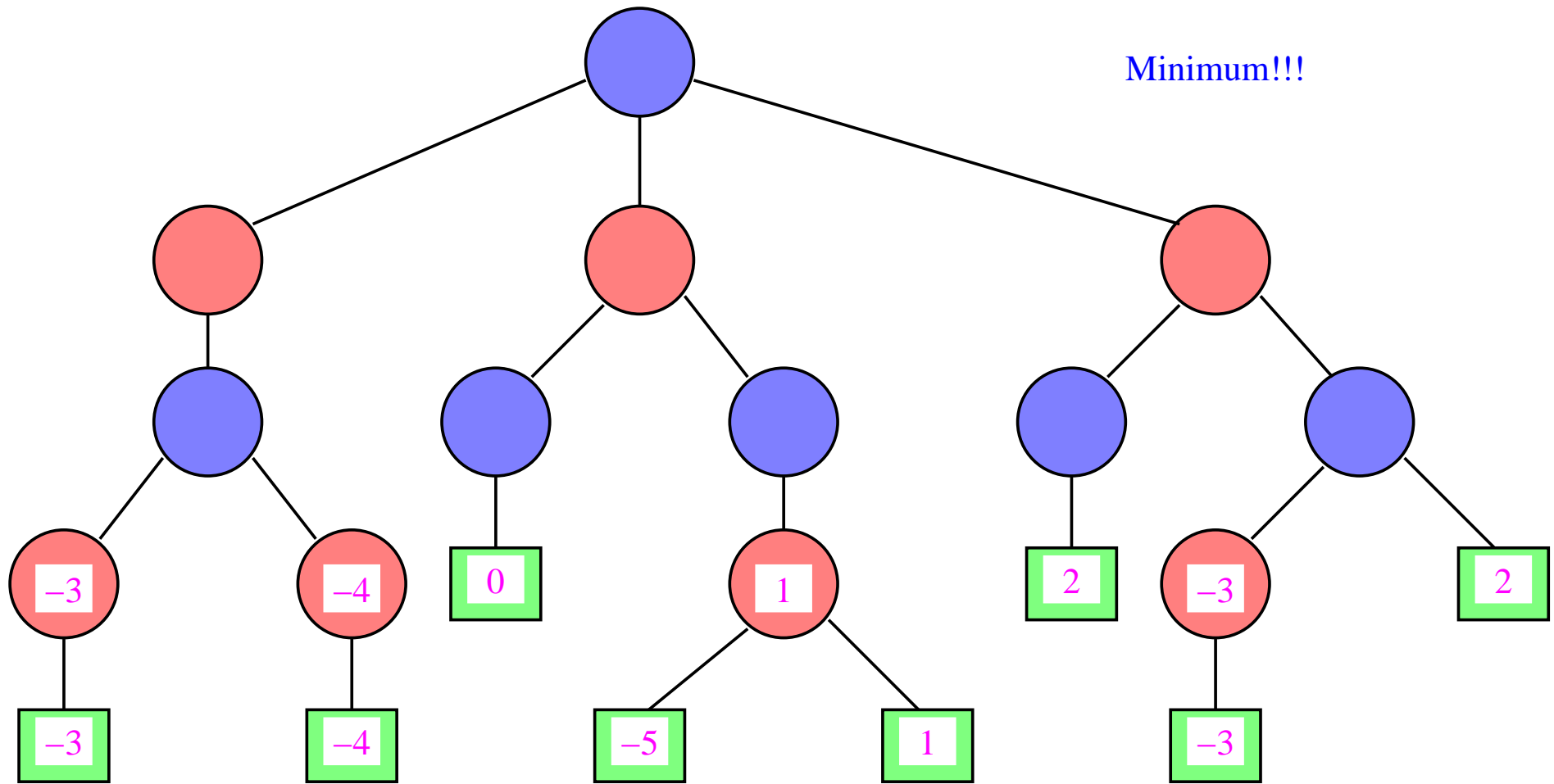
Was müssen wir tun, um **sicher** ein negatives Blatt zu erreichen?

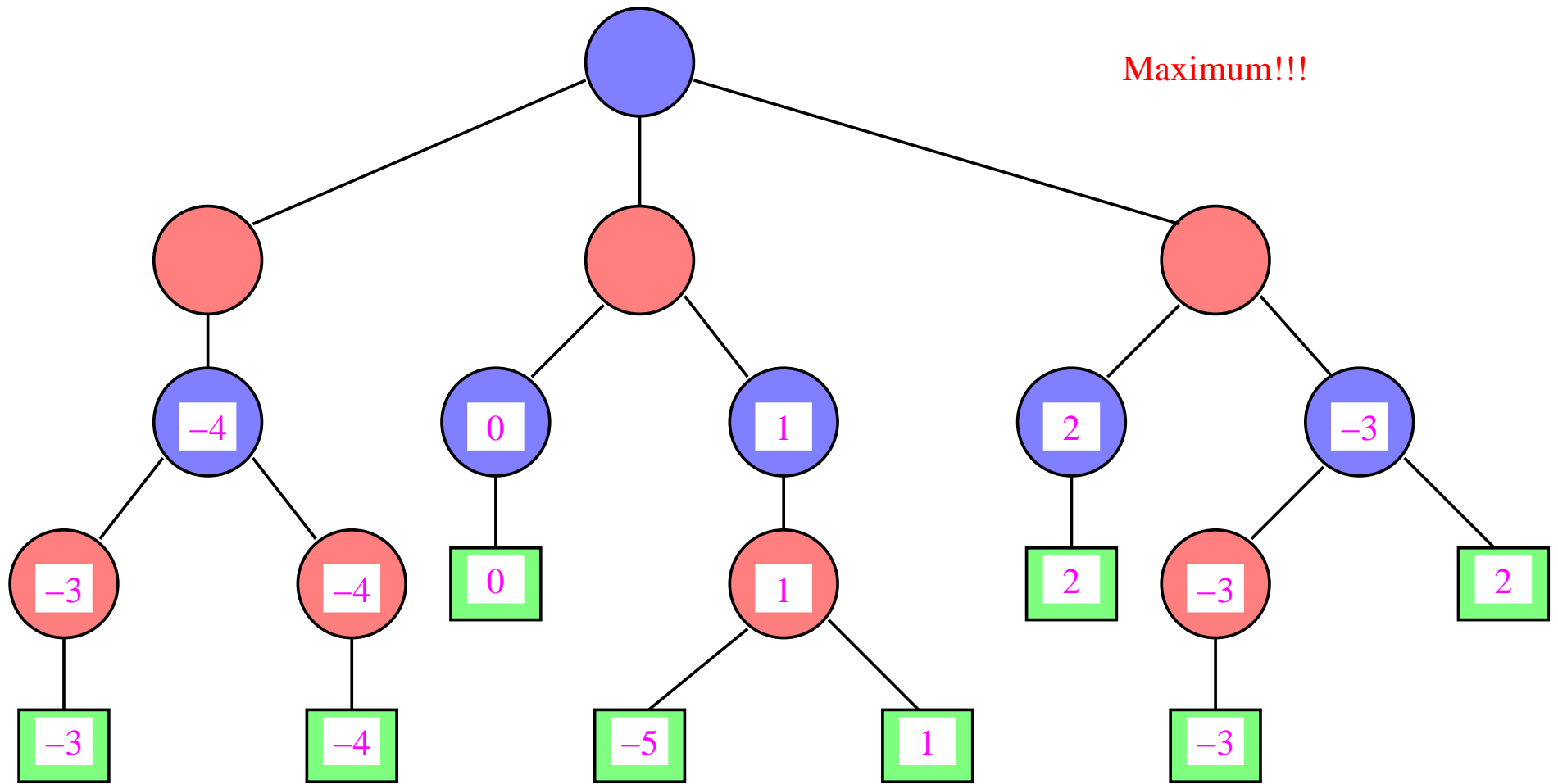
Idee:

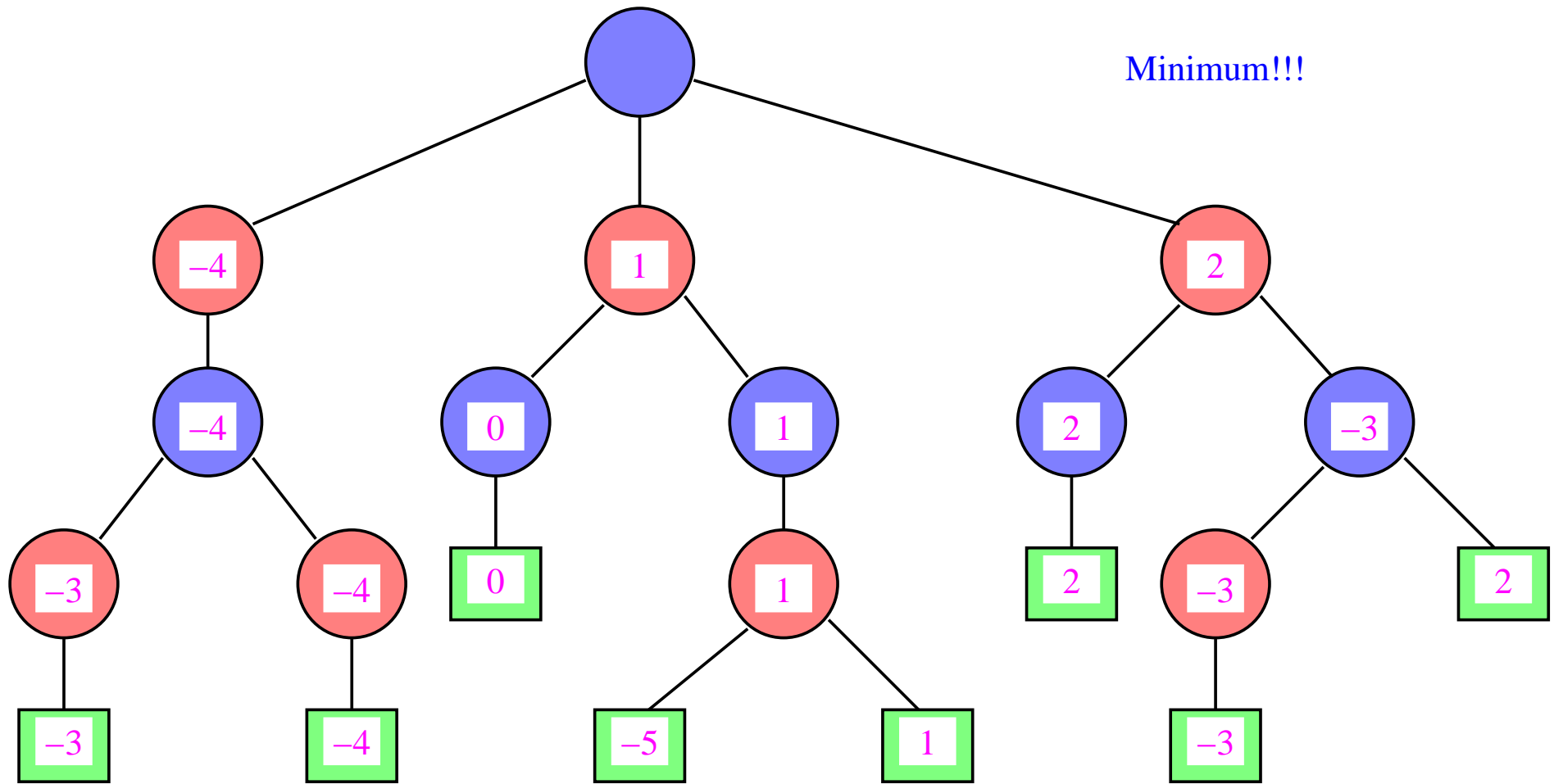
- Wir ermitteln für jede Konfiguration den jeweils **besten** zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

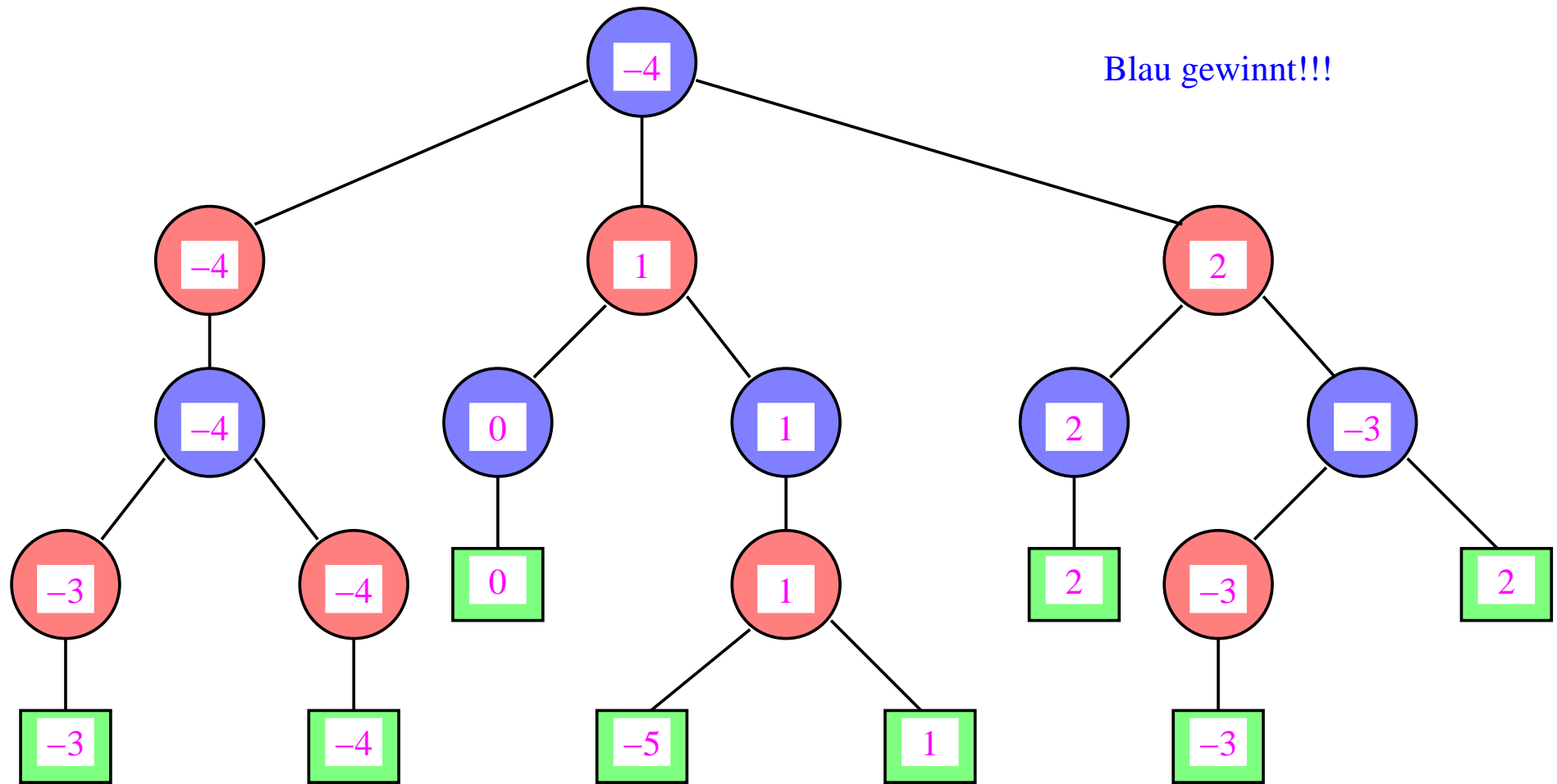
Fall 1: Die Konfiguration ist **blau**: wir sind am Zug. Wir können garantiert das **Minimum** der Gewinne der Söhne erzielen.

Fall 2: Die Konfiguration ist **rot**: der Gegner ist am Zug. Er kann garantiert das **Maximum** der Gewinne der Söhne erzielen.







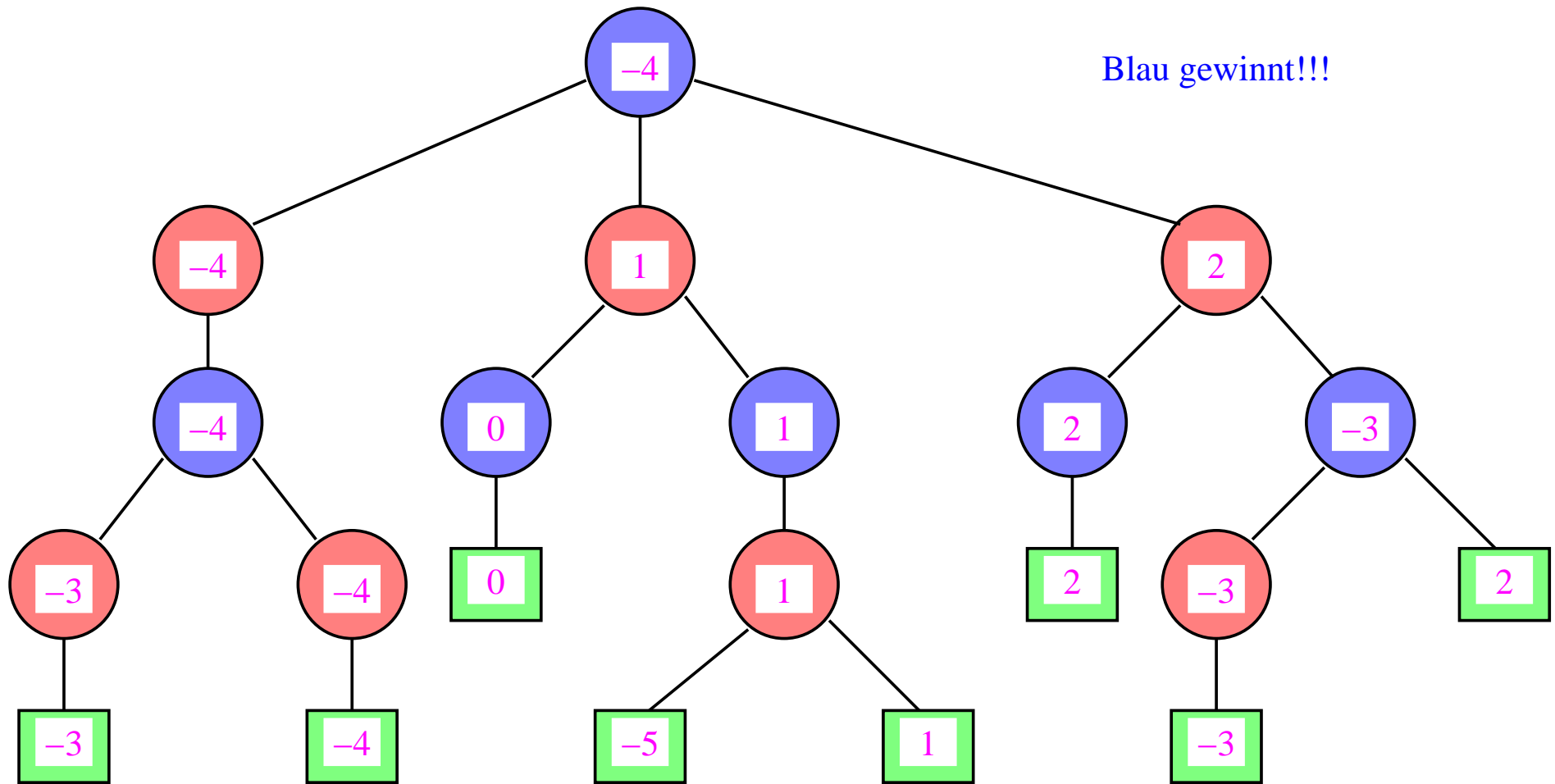


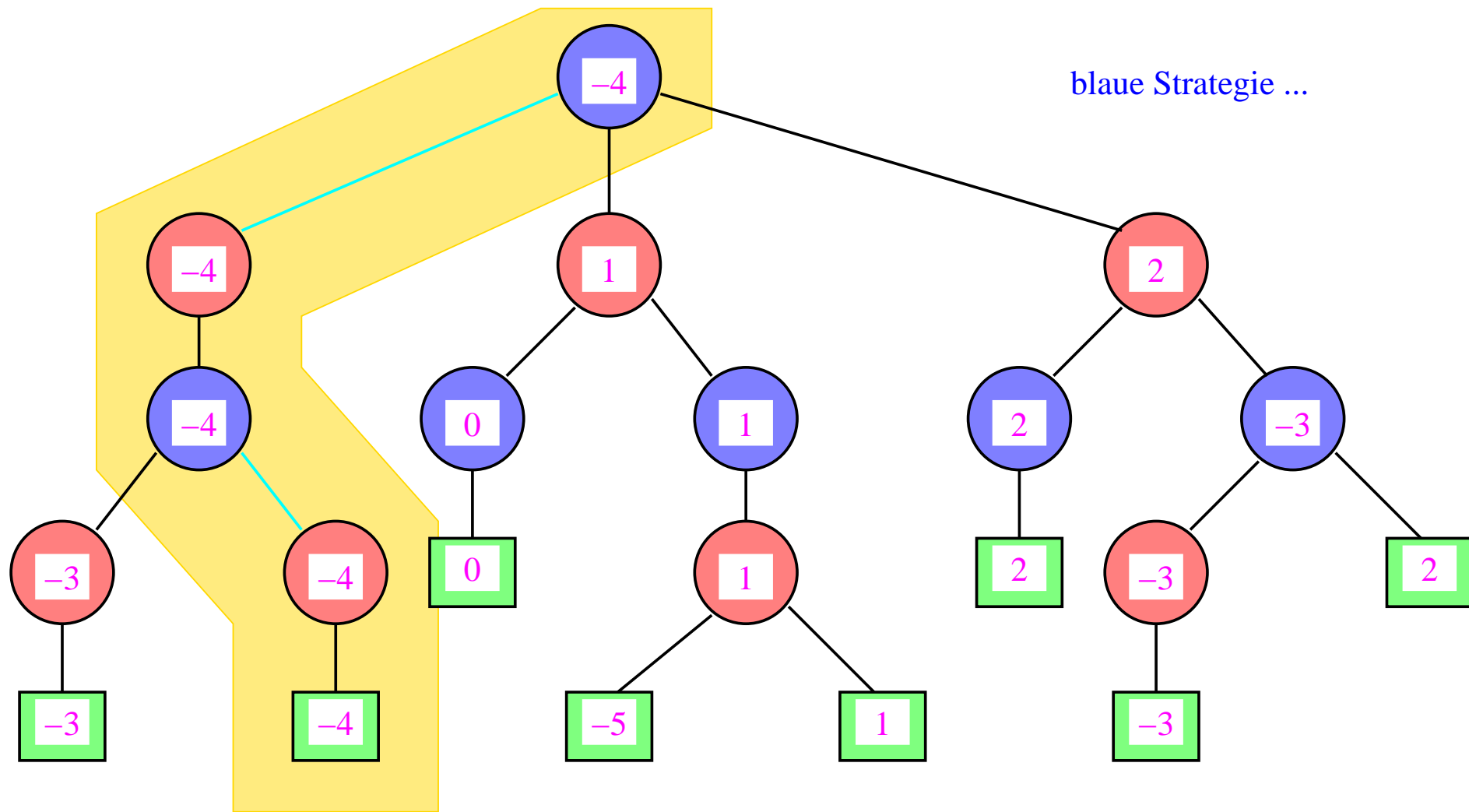
Blau gewinnt!!!

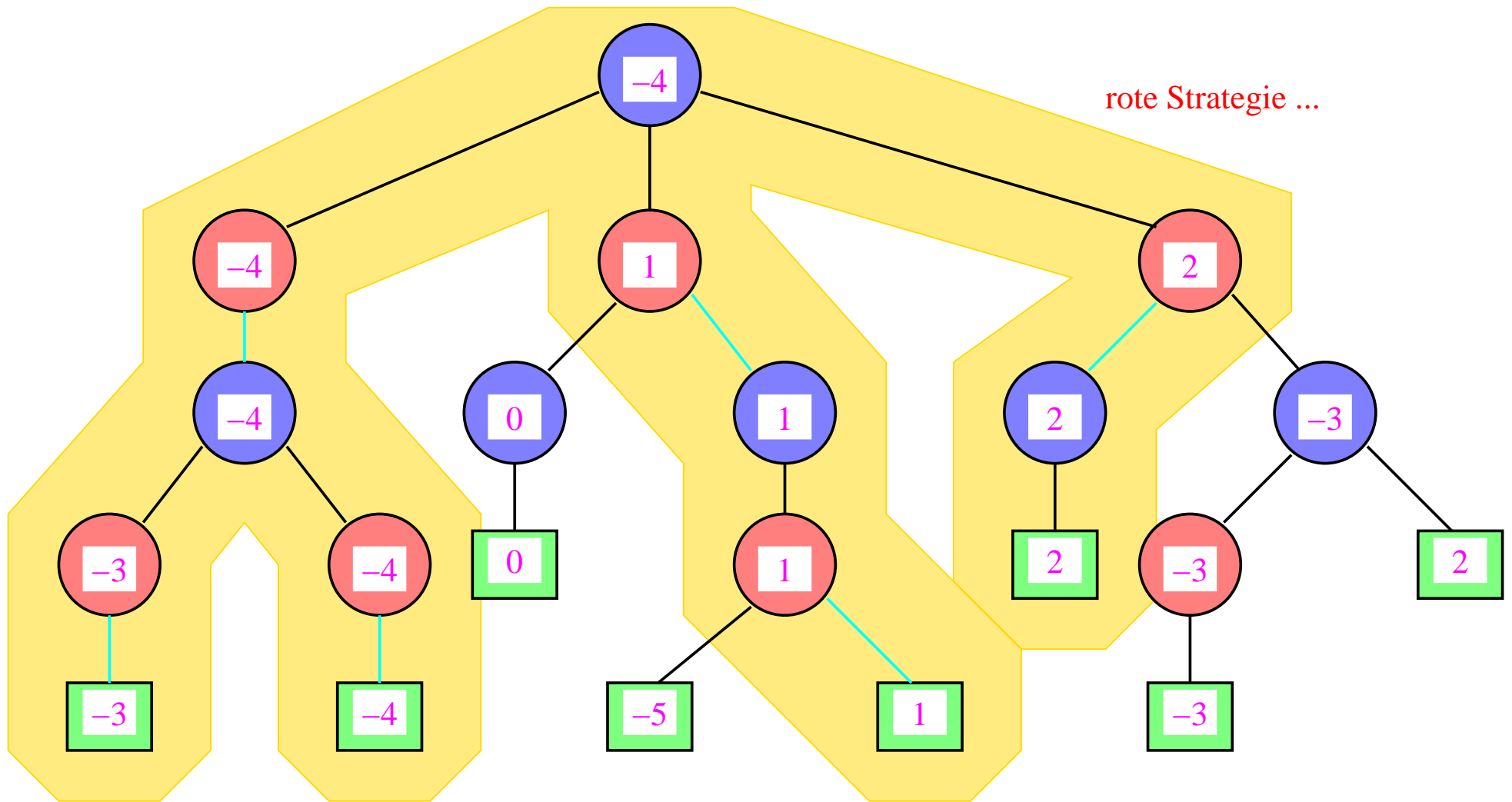
Eine **Strategie** ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine **optimale** Strategie ist eine, deren Anwendung garantiert zu einer End-Konfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

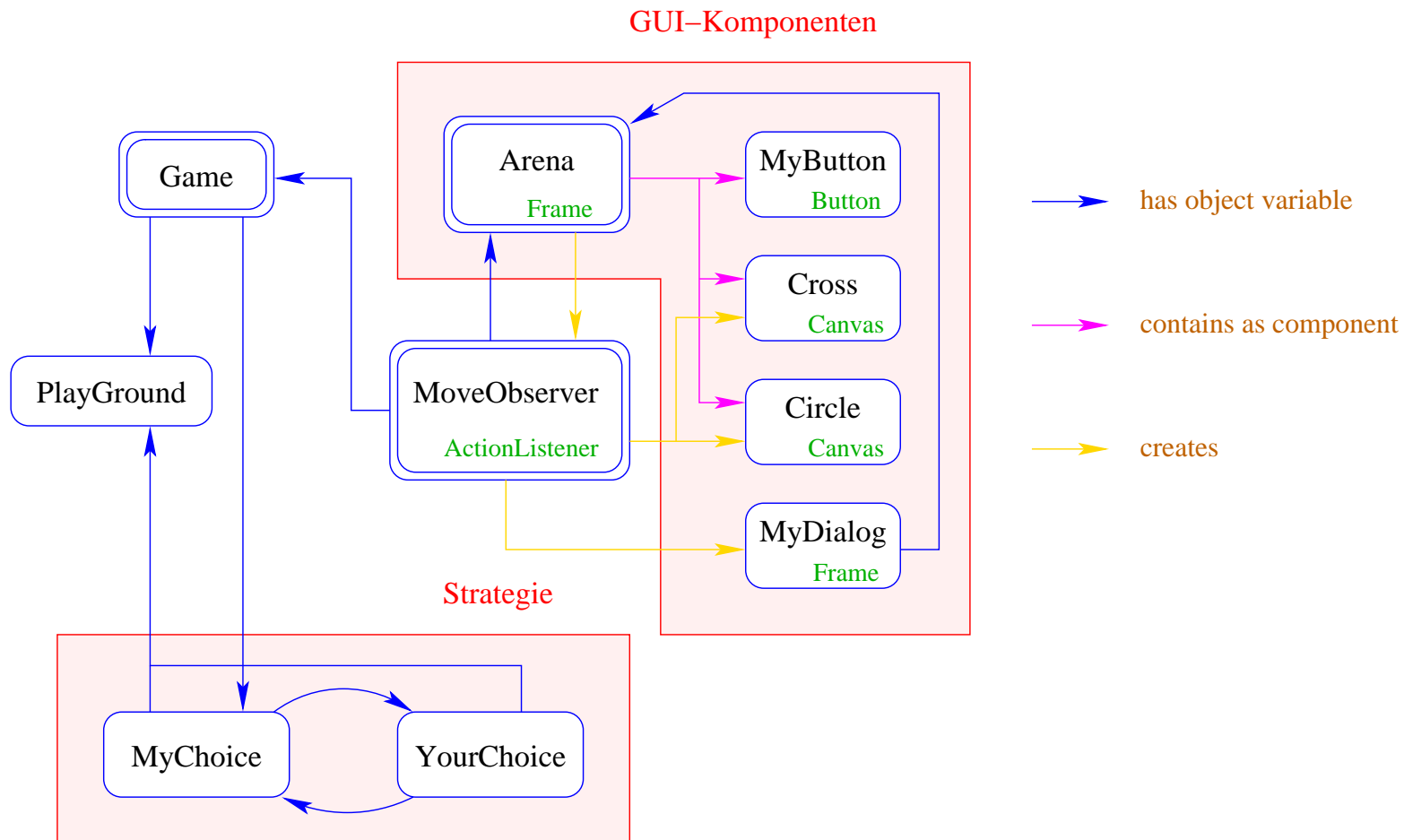
Eine **akzeptable** Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wannimmer das möglich ist ...







2.1 Überblick über das Tic-Tac-Toe-System



2.2 Berechnung einer Strategie

- Die Knoten des Spiel-Baums sind aus den Klassen `YourChoice` und `MyChoice`.
- `MyChoice` implementiert Knoten, in denen das Programm zieht.
- `YourChoice` implementiert Knoten der Spielerin.
- Jeder Knoten enthält:
 - das aktuelle Spielbrett `ground`,
 - einen garantierten Gewinn `value`
 - sowie (Verweise auf) die Nachfolger-Knoten.
- `MyChoice`-Knoten enthalten zusätzlich den empfohlenen Zug `acceptableChoice`.