

Simulator für digitale Schaltungen

Vorname: Alexander

Nachname: Schmidt

Matrikelnummer: 5502046

Zweck der Anwendung

Der Simulator für digitale Schaltungen soll Nutzern ermöglichen, digitale Schaltkreise virtuell zu testen und zu analysieren. Die Anwendung zielt darauf ab, ein Werkzeug zur Verfügung zu stellen, mit dem man etwas komplexe Schaltungen ohne physische Bauteile mit Hilfe der Turtle konstruieren und verstehen kann.

Wichtig: Das Porgramm wird nur mit der jshell gesteuert. Wenn beim ersten Versuch des Öffnens des Programmes ein Fehler auftritt, einfach noch mal öffnen, dann sollte es immer funktionieren.

Wahrheitstabellen für Gattertypen

AND-Gatter

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR-Gatter

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR-Gatter

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

NAND-Gatter

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	0

NOR-Gatter

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	0

XNOR-Gatter

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	1

NOT-Gatter

Input	Output
0	1
1	0

Dokumentation

Szenario 1 - Grundlegende Gatter

Ziel

Ein Benutzer soll verschiedene grundlegende Gatter (NOT, AND, OR, NAND, NOR, XOR, XNOR) gemäß der IEC-Norm erstellen und deren Funktionsweise testen können. Zum Beispiel kann ein Benutzer ein AND-Gatter mit zwei Eingängen erstellen und die Logik überprüfen.

1. Gatter und Eingänge erstellen

Konstruktoren:

- **Input:** Erstellt einen Eingang mit einem bestimmten Namen und optional einem Anfangswert (0 oder 1).

```
Input(String name, int inputValue)
Input(String name) // inputValue = 0
```

Beispiel:

```
Input x1 = new Input("x1", 1);
Input x2 = new Input("x2", 0);
```

- **Gate:** Erstellt ein Gatter mit einem Typ (z. B. AND, OR) und einem Namen. Jedes Gatter (außer NOT) hat zwei Eingänge.

```
Gate(String type, String name)
```

Beispiel:

```
Gate andGate = new Gate("AND", "andGate1");
Gate orGate = new Gate("OR", "orGate1");
```

2. Komponenten hinzufügen

Die Methode `addComponent` fügt eine Komponente (z. B. ein Gatter oder einen Eingang) an eine bestimmte Position im Schaltungsfeld hinzu. Dabei ist zu beachten, dass Eingänge in die erste Spalte und Gatter nicht in die erste Spalte hinzugefügt werden können.

```
void addComponent(int row, int col, T component) {
    // vorhandene Position prüfen
    if (!isValidPosition(row, col))
        throw new IllegalArgumentException("Ungültige Position: (" + row + ", " + col + "). Diese Position existiert nicht im Schaltungsfeld.");

    // Spalte 1 auf Input-Objekte prüfen
    if (col == 1 && !(component instanceof Input)) throw new IllegalArgumentException("Spalte 1 ist nur für Input-Objekte reserviert.");

    // andere Spalten auf Input-Objekte prüfen
    if (col != 1 && (component instanceof Input)) throw new IllegalArgumentException("Input-Objekte dürfen nur in Spalte 1 hinzugefügt werden.");

    Point position = new Point(col, row);
    Rectangle gateArea = getGateArea(row, col);

    // freie Position prüfen
    if (components.containsKey(position)) throw new IllegalArgumentException("An dieser Position existiert bereits eine Komponente!");
}
```

```

// bereits existierendes Objekt im Schaltungsfeld prüfen
if (components.containsValue(component))
    throw new IllegalArgumentException("Das Objekt " + component + " existiert bereits im Schaltungsfeld und kann nicht erneut hinzugefügt werden.");

// prüft, ob der Bereich durch ein Kabel blockiert ist
for (Point wirePoint : wirePoints)
    if (gateArea.contains(wirePoint))
        throw new IllegalArgumentException("In dieser Zelle verläuft ein Kabel. Komponente kann hier nicht platziert werden.");

// Komponente zur Map hinzufügen
components.put(position, component);

turtle1.moveTo(getPixel(col), getPixel(row)).backward(25).left(90).forward(25).right(90); // zur Position gehen

// Typprüfung und Zeichnen der Komponente
if (component instanceof Gate) selectGateAndDraw(component);
else if (component instanceof Input) drawInput(component);
else throw new IllegalArgumentException("Unbekannter Komponententyp: " + component.getClass().getSimpleName());

System.out.println(component + " an Position (" + row + ", " + col + ") hinzugefügt.");
}

```

Beispiel:

```

Circuit<Object> c1 = new Circuit<>("Schaltkreis 1", 10, 5); // 10 Zeilen und 5 Spalten
c1.addComponent(2, 1, x1); // x1 in Zeile 2, Spalte 1 hinzufügen
c1.addComponent(3, 1, x2);
c1.addComponent(2, 3, andGate1);

```

Herausforderungen:

- **Überprüfung von Positionen:** Es muss gewährleistet sein, dass die Komponente in eine gültige Position eingefügt wird (z. B. Eingänge nur in Spalte 1).
- **Prüfung auf Überschneidungen:** Stellen Sie sicher, dass an einer Position keine andere Komponente oder Kabel bereits existiert.
- **Unterstützung unterschiedlicher Typen:** Die Methode muss sowohl mit Input- als auch mit Gate-Objekten umgehen können. Dies erfordert Typprüfungen und geeignete Zeichnungslogik für die Turtle.

3. Komponenten verbinden

Die Methode `connectComponents` verbindet die Ausgänge einer Quelle mit den Eingängen eines Ziels.

```

void connectComponents(T sourceComponent, T destinationComponent, int inputNumber) {
    // Position prüfen
    if (!isValidConnection(sourceComponent, destinationComponent)) return;

    if (inputNumber != 1 && inputNumber != 2) throw new IllegalArgumentException("Es gibt nur zwei Eingänge. Bitte nur Eingang 1 oder 2 auswählen.");

    Point sourceOutput = outputPositions.get(sourceComponent);
    Point destInput = (inputNumber == 1) ? firstInputPositions.get(destinationComponent) : secondInputPositions.get(destinationComponent);

    if (sourceOutput == null || destInput == null) throw new IllegalArgumentException("Verbindung nicht möglich. Komponentenposition nicht gefunden.");

    // doppelte Verbindungen vermeiden
    if (isConnectionPresent(destinationComponent, inputNumber))
        throw new IllegalArgumentException("Verbindung existiert bereits: " + sourceComponent + " -> " + destinationComponent + " (Eingang " + inputNumber + ")");

    // prüfen, ob ein offset in x-Richtung erforderlich ist
    boolean applyXOffset = connections.stream()
        .filter(conn -> conn.destination.equals(destinationComponent))
        .map(conn -> conn.source)

```

```

        .anyMatch(otherSource ->
            checkSourcesYPositions(sourceComponent, otherSource, destinationComponent));

// prüfen, ob ein offset in y-Richtung erforderlich ist
boolean applyYOffset = connections.stream()
    .filter(conn -> !conn.destination.equals(destinationComponent) && conn.inputNumber == inputNumber)
    .anyMatch(conn -> compareHorizontalPositions(sourceComponent, destinationComponent));

// prüft, ob eine Quelle bereits eine Verbindung hat
boolean isAlreadyConnected = isSourceConnected(sourceComponent);

// Verbindung in die Liste eintragen
connections.add(new Connection<>(sourceComponent, destinationComponent, inputNumber));
System.out.println("Verbindung hinzugefügt: " + sourceComponent + " -> " + destinationComponent + " (Eingang " + inputNumber + ")");

turtle1.moveTo(sourceOutput.x, sourceOutput.y);
drawConnection(sourceOutput, destInput, sourceComponent, destinationComponent, applyXOffset, applyYOffset, isAlreadyConnected);

if (!isRedrawing) {
    isRedrawing = true;    // Schutz aktivieren
    evaluateCircuit();
    drawNewCircuit();      // Alles neu zeichnen
    isRedrawing = false;   // Schutz deaktivieren
}
}

```

Beispiel:

```

c1.connectComponents(x1, andGate, 1); // x1 mit Eingang 1 von andGate verbinden (1 = oberer Eingang des Gatters)
c1.connectComponents(x2, andGate, 2); // x2 mit Eingang 2 von andGate verbinden (2 = unterer Eingang des Gatters)

```

Herausforderungen:

- **Verbindung zwischen Quellen und Zielen:** Sicherstellen, dass die Verbindung nur dann erstellt wird, wenn die Quelle und das Ziel in der Schaltung vorhanden sind.
- **Gattertyp beachten:** Nur zulässige Verbindungen (z. B. zwei Eingänge bei AND-Gattern) dürfen hergestellt werden.
- **Fehlerhafte Verbindungen vermeiden:** Das Ziel darf nicht links von der Quelle liegen oder in derselben Spalte.
- **Offset bei Verbindung:** Wenn bereits eine Verbindung zu einer Zielkomponente besteht, muss beim Zeichnen der Verbindung ein “Offset” angewendet werden, um Überlappungen der Kabeln zu vermeiden.
- **Umzeichnen bei Hindernissen:** Falls sich ein Gatter in derselben Zeile zwischen der Quelle und dem Ziel befindet, muss die Verbindung so angepasst werden, dass sie um das Hindernis herumgeführt wird (z. B. durch Umleitung oberhalb oder unterhalb des Hindernisses).
- **Farbprüfung der Verbindung:** Jede Verbindung muss abhängig vom Logikwert korrekt gezeichnet werden:
 - **Grün (HIGH / 1):** Wenn der Ausgang der Quelle den Wert 1 hat.
 - **Schwarz (LOW / 0):** Wenn der Ausgang der Quelle den Wert 0 hat.

4. Eingang setzen

Die Methode `setInput` ändert den Wert eines Eingangs und aktualisiert automatisch die gesamte Schaltung. Nach dem Setzen wird die Schaltung sofort neu ausgewertet und gezeichnet.

```

void setInput(T component, int value) {
    isRedrawing = true;

    if (value != 0 && value != 1) throw new IllegalArgumentException("Bitte nur 1 oder 0 schalten.");
    if (component instanceof Input inputComponent) {
        inputComponent.inputValue = value; // Eingang wird umgeschaltet
        System.out.println(inputComponent.getInputName() + " wurde auf " + inputComponent.inputValue + " geschaltet.");
        evaluateCircuit();
        drawNewCircuit();
    }
}

```

```
} else throw new IllegalArgumentException("Fehler: Nicht kompatible Komponente. Bitte Input-Komponente angeben.");
```

```
isRedrawing = false;  
}
```

Beispiel:

```
c1.setInput(x1, 1); // x1 auf HIGH (1) setzen  
c1.setInput(x2, 0); // x2 auf LOW (0) setzen
```

Herausforderungen:

- **Echtzeitaktualisierung der Schaltung:** Nach dem Schalten eines Eingangs muss die gesamte Schaltung neu ausgewertet und gezeichnet werden.
- **Logikfehler vermeiden:** Die Eingaben müssen korrekt auf 0 oder 1 begrenzt sein.

5. Schaltung auswerten

Die Methode `evaluateCircuit` berechnet die Logik für alle verbundenen Komponenten basierend auf den Eingabewerten. Diese Methode wird immer automatisch aufgerufen, sobald ein Eingang geschalten wird oder man Komponente verbindet.

```
void evaluateCircuit() {  
    boolean hasChanged;  
  
    do {  
        hasChanged = false;  
  
        // iteriere über alle Verbindungen und berechne die Outputs  
        for (Connection<T> connection : connections) {  
            T source = connection.source;  
            T destination = connection.destination;  
            int inputNumber = connection.inputNumber;  
            int sourceOutput = getComponentOutput(source); // Output der Quelle ermitteln  
  
            // Zielkomponente prüfen (muss immer Gate sein)  
            if (destination instanceof Gate gate) {  
                int previousOutput = gate.output;  
  
                // setze den Eingangswert abhängig vom Eingang (1 oder 2)  
                if (inputNumber == 1) gate.setInput1(sourceOutput);  
                else if (inputNumber == 2) gate.setInput2(sourceOutput);  
  
                gate.inputToOutput();  
  
                // prüfen, ob sich der Output geändert hat  
                if (gate.output != previousOutput) {  
                    System.out.println "[" + gate.getName() + "] Output geändert:";  
                    System.out.println("  Vorher: " + previousOutput);  
                    System.out.println("  Nachher: " + gate.output);  
                    hasChanged = true;  
                }  
            }  
        }  
    }  
    while (hasChanged); // wiederholen, solange sich Outputs ändern  
}
```

Herausforderungen:

- **Iterative Auswertung:** Jede Komponente muss in der richtigen Reihenfolge ausgewertet werden, abhängig davon, welche Verbindungen existieren.
- **Änderungen verfolgen:** Wenn sich ein Ausgang ändert, muss die Änderung korrekt in der Konsole angezeigt werden.

- **Rekursivität vermeiden:** Es muss verhindert werden, dass Schleifen in den Verbindungen zu endlosen Auswertungen führen.

6. Schaltung dynamisch halten

Die Methode `drawNewCircuit` zeichnet die gesamte Schaltung auf dem Feld `neu`. Diese Methode wird immer automatisch aufgerufen, sobald die Schaltung neu gezeichnet werden muss, um Veränderungen optisch in der Turtle zu sehen.

```
void drawNewCircuit() {
    turtle1.reset();
    drawCircuitField();
    for (Map.Entry<Point, T> entry : components.entrySet()) {
        Point position = entry.getKey();
        T component = entry.getValue();
        addComponentWithoutChecks(position.y, position.x, component);
    }
    reconnectComponents(); // Kabel wieder zeichnen
}
```

Herausforderungen:

- **Vollständige Aktualisierung:** Alle Verbindungen und Komponenten müssen erneut gezeichnet werden.

Beispielablauf: Mit einem AND-Gatter und zwei Eingängen

jshell:

```
// Schaltkreis erstellen
Circuit<Object> c1 = new Circuit<>("Schaltkreis 1", 10, 5);

// Eingänge erstellen
Input x1 = new Input("x1", 1);
Input x2 = new Input("x2", 0);

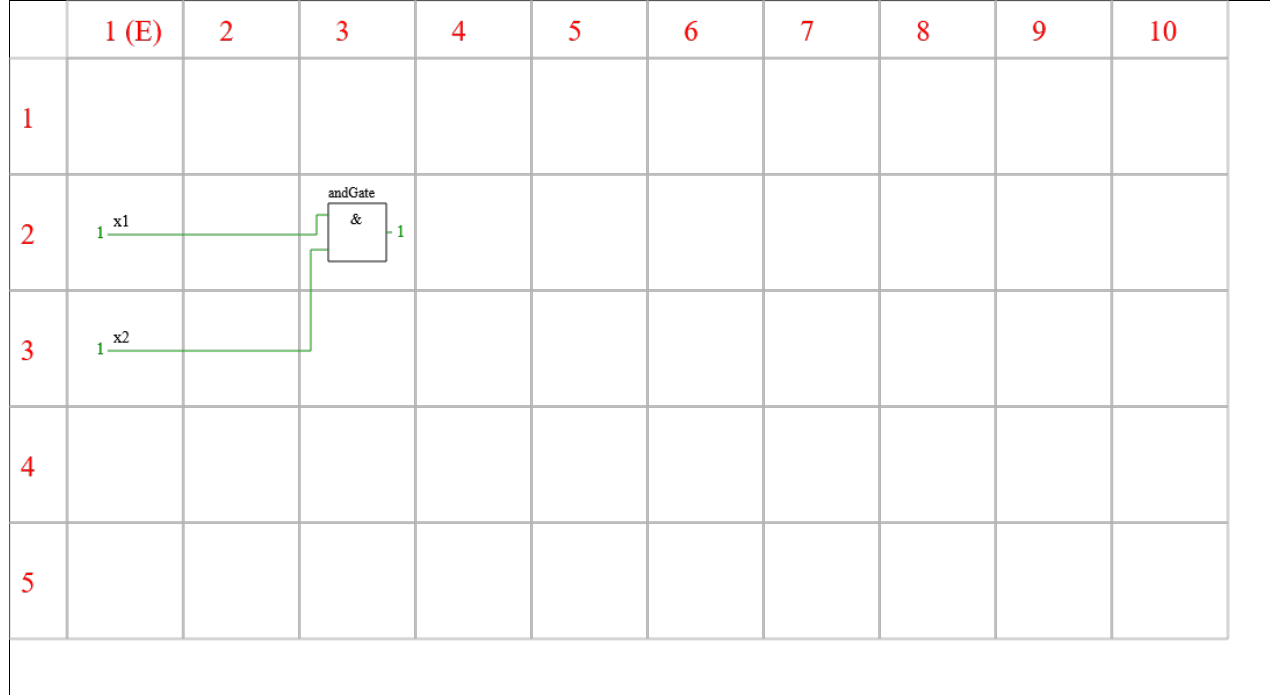
// Gatter erstellen
Gate andGate = new Gate("AND", "andGate");

// Komponenten hinzufügen
c1.addComponent(2, 1, x1);
c1.addComponent(3, 1, x2);
c1.addComponent(2, 3, andGate);

// Komponenten verbinden
c1.connectComponents(x1, andGate, 1);
c1.connectComponents(x2, andGate, 2);

// Eingang schalten
c1.setInput(x1, 1);
c1.setInput(x2, 1);
```

Turtle:



Szenario 2 - Schaltungsaufbau

Ziel

in Benutzer erstellt eine Schaltung, die verschiedene Gatter miteinander verschaltet. Zum Beispiel: Der Benutzer kann aus NOT- und NAND-Gatter eine AND- oder OR-Schaltung bauen und die Ausgabe auslesen.

Beispielablauf: AND-Schaltung aus NOT und NAND

jshell:

```
// Schaltkreis erstellen
Circuit<Object> c1 = new Circuit<>("Schaltkreis 1", 10, 5);

// Eingänge erstellen
Input x1 = new Input("x1", 1);
Input x2 = new Input("x2", 1);

// Gatter erstellen
Gate notGate1 = new Gate("NOT", "notGate1");
Gate notGate2 = new Gate("NOT", "notGate2");
Gate nandGate = new Gate("NAND", "nandGate");

// Komponenten hinzufügen
c1.addComponent(2, 1, x1);
c1.addComponent(3, 1, x2);
c1.addComponent(2, 2, notGate1);
c1.addComponent(3, 2, notGate2);
```



```

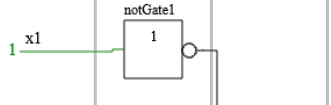
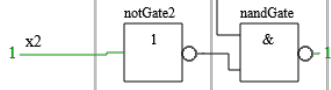
c1.addComponent(3, 3, nandGate);

// Komponenten verbinden
c1.connectComponents(x1, notGate1, 1);
c1.connectComponents(x2, notGate2, 1);
c1.connectComponents(notGate1, nandGate, 1);
c1.connectComponents(notGate2, nandGate, 2);

// Eingänge schalten
c1.setInput(x1, 1);
c1.setInput(x2, 1);

```

Turtle:

	1 (E)	2	3	4	5	6	7	8	9	10	
1											
2											
3											
4											
5											

Szenario 3 - Schaltungsanalyse

Ziel

3.3. Szenario 3 - Schaltungsanalyse Ein Benutzer erstellt eine komplexere Schaltung, die er daraufhin analysieren kann. Dies hilft einem, die einzelnen Zwischenschritte der Schaltung zu verstehen. Zum Beispiel: Der Benutzer kann sich eine Tabelle erzeugen lassen, mit allen kombinierten HIGH/LOW Eingängen, Zwischenausgaben bzw. Zwischenschritte und Endausgaben.

Wichtige Methoden

1. Gesamte Wahrheitstabelle zeichnen

Die Methode `drawTable` erstellt die gesamte Wahrheitstabelle der Schaltung, indem sie das Tabellenlayout in einer neuen Turtle zeichnet, die Eingabekombinationen und die Ergebnisse aller Gatter in die Tabelle einträgt.

```

void drawTable() {
    turtle2 = new Turtle(1600, 400);
    turtle2.reset();
    // Tabellemgitter zeichnen
    drawTableFrame();

    // Inputs-Kombinationen schreiben
    drawInputValuesInTable();

    // Gate-Outputs-Kombinationen schreiben
    drawOutputValuesInTable();
}

```

Ablauf:

1. **Tabellenlayout zeichnen:** Die Methode `drawTableFrame` zeichnet das Gitter der Tabelle basierend auf der Anzahl der Eingänge und Ausgänge.
2. **Eingabekombinationen einfügen:** Die Methode `drawInputValuesInTable` generiert alle möglichen HIGH/LOW-Eingangskombinationen und trägt sie ein.
3. **Gatterausgaben einfügen:** Die Methode `drawOutputValuesInTable` analysiert die Ergebnisse der Gatter und fügt diese der Tabelle hinzu.

2. Tabellenlayout zeichnen

Die Methode `drawTableFrame` erstellt das Layout der Tabelle. Sie berechnet die benötigte Höhe basierend auf der Anzahl der Eingänge und Ausgänge und zeichnet die Gitterlinien.

```

void drawTableFrame() {
    // Anzahl der Verbindungen abspeichern
    long outputCount = connections.stream()
        .map(conn -> conn.destination) // Nur die Zielkomponenten betrachten
        .distinct() // Doppelte entfernen
        .count(); // Zählen
    outputCount = (int) outputCount;
    if (outputCount > 6) {
        printConnections();
        throw new IllegalArgumentException("Eine Auswertung ist nur mit bis zu sechs Ausgängen möglich.");
    }

    int rowHeight = 60;

    // Anzahl der Inputs abspeichern
    List<Input> inputs = new ArrayList<>();
    for (T component : components.values()) if (component instanceof Input input) inputs.add(input);
    int inputsAmount = inputs.size();
    int totalRows = (int) Math.pow(2, inputsAmount); // Anzahl der Spalten in Abhängigkeit von der Anzahl der Inputs berechnen
    int tableHeight = rowHeight * totalRows; // Höhe der Tabelle bestimmen in Abhängigkeit der Anzahl von Inputs

    // Input-Teil der Tabelle zeichnen
    turtle2.moveTo(0, 0).penUp().right(90).forward(40).left(90).forward(10).penDown();
    for (int i = 0; i < inputsAmount; i++)
        turtle2.forward(30).right(90).forward(tableHeight).penUp().backward(tableHeight).penDown().backward(30).penUp().forward(30).left(90).penDown();
    turtle2.forward(5);

    // Output-Teil der Tabelle zeichnen
    for (int i = 0; i < outputCount; i++)
        turtle2.left(90).forward(30).penUp().backward(30).penDown().backward(tableHeight).penUp().forward(tableHeight).right(90).penDown().forward(240);
}

```

Herausforderungen:

- **Berechnung der Tabellenhöhe:** Die Höhe der Tabelle wird dynamisch anhand der Anzahl der Eingangskombinationen berechnet.
- **Begrenzung der Ausgänge:** Maximal sechs Ausgänge sind erlaubt. Bei mehr Ausgängen wird eine Ausnahme ausgelöst.

- **Einteilung in Eingangs- und Ausgangsbereich:** Die Tabelle wird in einen Bereich für Eingabekombinationen und einen Bereich für Gatterausgaben unterteilt.

3. Eingabekombinationen schreiben

Die Methode `drawInputValuesInTable` generiert alle möglichen Kombinationen von HIGH/LOW für die Eingänge und trägt sie in die Tabelle ein.

```
void drawInputValuesInTable() {
    // Anzahl der Verbindungen abspeichern
    List<Input> inputs = new ArrayList<>();
    for (T component : components.values()) if (component instanceof Input input) inputs.add(input);
    int inputsAmount = inputs.size();

    // Liste all der Kombinationen abspeichern
    List<List<Integer>> combinations = generateInputCombinations(inputsAmount);

    // Input-Namen
    turtle2.moveTo(20, 30).penUp(); // Start
    for (int i = 0; i < inputsAmount; i++)
        turtle2.left(90).text("" + inputs.get(i).getInputName(), null, 13, null).right(90).forward(30);

    // Input-Kombinationen
    turtle2.moveTo(20, 75).penUp(); // Start
    for (int i = 0; i < combinations.size(); i++) {
        List<Integer> combination = combinations.get(i);
        for (int j = 0; j < combination.size(); j++) {
            int value = combination.get(j);
            // Werte in die Tabelle schreiben
            turtle2.left(90).text("" + value, null, 14, null).right(90).forward(30);
            if ((j + 1) % inputsAmount == 0) turtle2.backward(inputsAmount * 30).right(90).moveTo(20, 75 + ((i + 1) * 60)).left(90);
        }
    }
}
```

Herausforderungen:

- **Generierung aller Kombinationen:** Mithilfe der Methode `generateInputCombinations` wird eine Liste aller möglichen HIGH/LOW-Kombinationen erstellt.
- **Platzierung der Werte:** Die Werte werden korrekt in die Zellen der Tabelle geschrieben.
- **Flexibilität:** Die Methode funktioniert dynamisch für beliebige Eingangsanzahlen.

4. Gatterausgaben schreiben

Die Methode `drawOutputValuesInTable` berechnet die Logik der Gatter für alle Eingangskombinationen und trägt die Ergebnisse in die Tabelle ein.

```
void drawOutputValuesInTable() {
    List<String> logicExpressions = getConnectionLogicStrings();
    List<List<Integer>> allOutputCombinations = evaluateLogicForAllInputs();

    List<Input> inputs = new ArrayList<>();
    for (T component : components.values()) if (component instanceof Input input) inputs.add(input);

    // Namen der Logik
    long outputCount = connections.stream()
        .map(conn -> conn.destination)
        .distinct()
        .count();
    outputCount = (int) outputCount;
    turtle2.moveTo(10, 30).penUp().forward(inputs.size() * 30 + 10);
    for (int i = 0; i < outputCount; i++) turtle2.left(90).text("" + logicExpressions.get(i), null, 13, null).right(90).forward(240);
}
```

```

// alle kombinierten Outputs
turtle2.moveTo(115 + (15 + 30 * inputs.size()), 75).penUp(); // Start
for (int i = 0; i < allOutputCombinations.size(); i++) {
    List<Integer> outputCombination = allOutputCombinations.get(i);
    for (int j = 0; j < outputCombination.size(); j++) {
        int value = outputCombination.get(j);
        // Output-Werte in die Tabelle schreiben
        turtle2.left(90).text("" + value, null, 14, null).right(90).forward(240);
        if ((j + 1) % outputCount == 0) turtle2.backward(outputCount + 240).right(90).moveTo(115 + (15 + 30 * inputs.size()), 75 + ((i + 1) * 60)).left(90);
    }
}
}
}

```

Herausforderungen:

- **Logikauswertung:** Die Methode `evaluateLogicForAllInputs` berechnet die Ergebnisse aller Gatter basierend auf den Eingangskombinationen.
- **Sortierung der Gatter:** Mithilfe der Methode `getSortedOutputs` werden die Gatter so sortiert, dass sie in der richtigen Reihenfolge ausgewertet werden.
- **Platzierung der Ausgaben:** Die Ergebnisse der Gatter werden dynamisch in die Tabelle eingetragen.

5. Generierung der Wahrheitstabelle für alle Eingänge

Die Methode `generateInputCombinations` erstellt eine Liste aller möglichen HIGH/LOW-Kombinationen für die Eingänge.

```

List<List<Integer>> generateInputCombinations(int numberOfInputs) {
    List<List<Integer>> combinations = new ArrayList<>();
    int totalCombinations = (int) Math.pow(2, numberOfInputs); // 2^numberOfInputs

    for (int i = 0; i < totalCombinations; i++) {
        List<Integer> combination = new ArrayList<>();
        for (int j = numberOfInputs - 1; j >= 0; j--) { // Bits werden von links nach rechts betrachtet
            int bit = (i >> j) & 1; // Beispiel für i = 1 und zwei Inputs: (1 >> 1) & 1 = 0, (1 >> 0) & 1 = 1
            combination.add(bit);
        }
        combinations.add(combination);
    }

    return combinations;
}

```

Ablauf:

1. **Berechnung der Kombinationen:** Es werden $2^{\text{numberOfInputs}}$ Kombinationen generiert.
2. **Bitweise Generierung:** Für jede Kombination wird geprüft, ob ein Bit HIGH oder LOW ist.

6. Logik aller Ausgänge bewerten

Die Methode `evaluateLogicForAllInputs` berechnet für jede Eingabekombination die Ausgänge aller Gatter.

```

List<List<Integer>> evaluateLogicForAllInputs() {
    // alle Input-Komponenten sammeln
    List<T> inputList = components.values().stream()
        .filter(component -> component instanceof Input) // filter nur Input-Objekte
        .collect(Collectors.toList()); // sammelt alle Inputs und fügt sie der Liste hinzu
    int inputCount = inputList.size();

    List<List<Integer>> allResults = new ArrayList<>();

    // alle möglichen Kombinationen von Inputs generieren
}

```

```

List<List<Integer>> inputCombinations = generateInputCombinations(inputCount);

// sortierte Gates
List<T> sortedOutputs = getSortedOutputs();

// über alle Kombinationen iterieren
for (List<Integer> combination : inputCombinations) {

    // Eingänge setzten
    for (int i = 0; i < inputCount; i++) {
        Input input = (Input) inputList.get(i);
        int value = combination.get(i);

        // nur setzem, wenn der Wert sich geändert hat
        if (input.inputValue != value) input.inputValue = value;
    }

    // Schaltung auswerten
    evaluateCircuit();
    isRedrawing = true;
    drawNewCircuit();
    isRedrawing = false;

    // Ergebnisse der Gates in der Reihenfolge speichern
    List<Integer> resultRow = new ArrayList<>();
    for (T output : sortedOutputs)
        if (output instanceof Gate gate)
            resultRow.add(gate.output);

    // Ergebnis zur Auswertungsliste hinzufügen
    allResults.add(resultRow);
}

return allResults;
}

```

Ablauf:

1. **Setzen der Eingänge:** Die Werte aller Eingänge werden basierend auf der aktuellen Kombination gesetzt.
2. **Schaltung auswerten:** Die Methode `evaluateCircuit` berechnet die Ausgänge aller Gatter.
3. **Ergebnisse speichern:** Die Ausgänge aller Gatter werden in der Reihenfolge gespeichert, die durch `getSortedOutputs` bestimmt wird.

Beispielablauf: Analyse einer komplexen Schaltung

jshell:

```

// Schaltkreis erstellen
Circuit<Object> c1 = new Circuit<>("Schaltkreis 1", 10, 5);

// Eingänge erstellen
Input x1 = new Input("x1");
Input x2 = new Input("x2");

// Gatter erstellen
Gate nandGate1 = new Gate("NAND", "nandGate1");
Gate nandGate2 = new Gate("NAND", "nandGate2");
Gate nandGate3 = new Gate("NAND", "nandGate3");
Gate nandGate4 = new Gate("NAND", "nandGate4");

```

```
Gate notGate = new Gate("NOT", "notGate");
```

```
// Komponenten hinzufügen
```

```
c1.addComponent(2, 1, x1);
c1.addComponent(4, 1, x2);
c1.addComponent(3, 3, nandGate1);
c1.addComponent(2, 5, nandGate2);
c1.addComponent(4, 5, nandGate3);
c1.addComponent(3, 7, nandGate4);
c1.addComponent(3, 9, notGate);
```

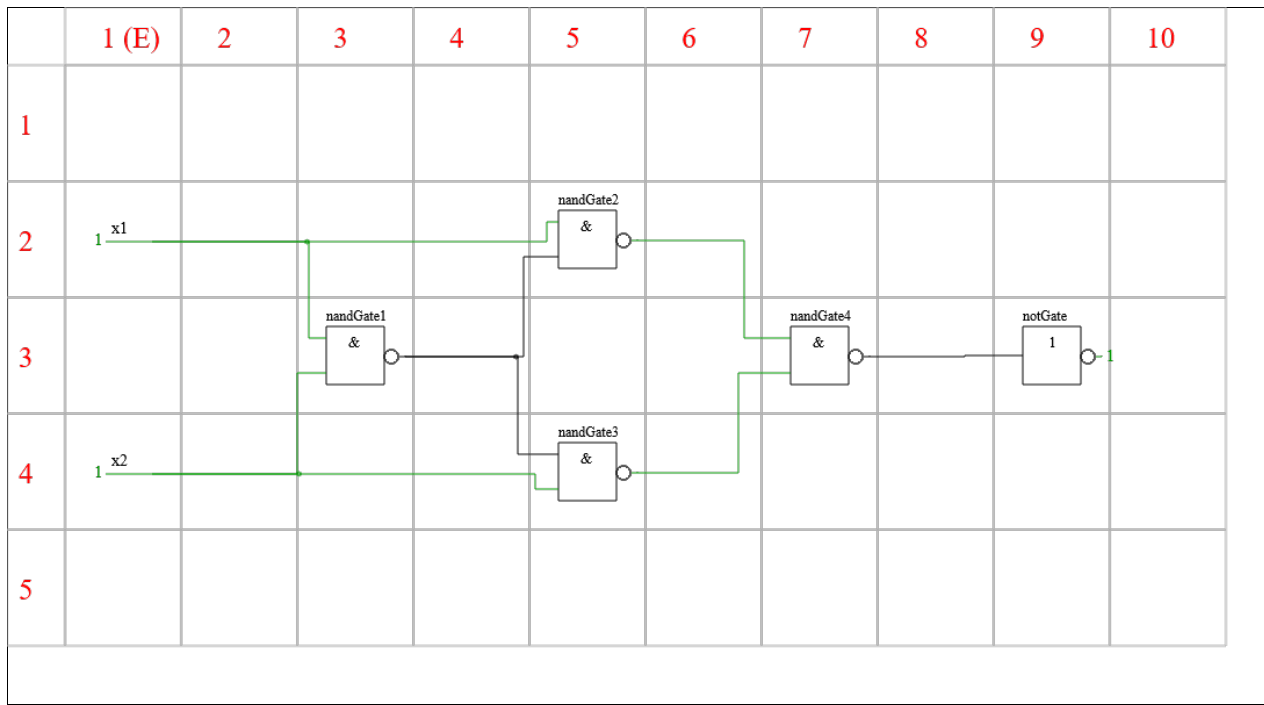
```
// Komponenten verbinden
```

```
c1.connectComponents(x1, nandGate2, 1);
c1.connectComponents(x1, nandGate1, 1);
c1.connectComponents(x2, nandGate3, 2);
c1.connectComponents(x2, nandGate1, 2);
c1.connectComponents(nandGate1, nandGate2, 2);
c1.connectComponents(nandGate1, nandGate3, 1);
c1.connectComponents(nandGate2, nandGate4, 1);
c1.connectComponents(nandGate3, nandGate4, 2);
c1.connectComponents(nandGate4, notGate, 1);
```

```
// Schaltung analysieren und Wahrheitstabelle ausgeben
```

```
c1.drawTable();
```

Turtle:



x1	x2	nandGate1 = x1 NAND x2	nandGate3 = nandGate1 NAND x2	nandGate2 = x1 NAND nandGate1	nandGate4 = nandGate2 NAND nandGate3	notGate = nandGate4 NOT
0	0	1	1	1	0	1
0	1	1	0	1	1	0
1	0	1	1	0	1	0
1	1	0	1	1	0	1

Szenario 4 - Halbaddierer

Ziel

Ein Benutzer soll die Möglichkeit haben, einen Halbaddierer zu erstellen oder einen vorgefertigten Halbaddierer zu nutzen. Der Halbaddierer kann zwei Binärzahlen (Eingänge) addieren und gibt die **Summe** und den **Übertrag (Carry)** aus.

1. Halbaddierer manuell erstellen

Ein Halbaddierer besteht aus zwei Eingängen (x1, x2), einem XOR-Gatter für die Summe und einem AND-Gatter für den Übertrag. Die Komponenten können manuell erstellt und verbunden werden.

Beispiel:

jshell:

```
// Schaltkreis erstellen
Circuit<Object> c1 = new Circuit<>("Halbaddierer", 10, 5);

// Eingänge erstellen
Input x1 = new Input("x1", 0);
Input x2 = new Input("x2", 0);

// Gatter erstellen
Gate xorGate = new Gate("XOR", "Summe");
Gate andGate = new Gate("AND", "Übertrag");

// Komponenten hinzufügen
c1.addComponent(2, 1, x1);
c1.addComponent(3, 1, x2);
c1.addComponent(2, 3, xorGate);
c1.addComponent(4, 3, andGate);

// Komponenten verbinden
c1.connectComponents(x1, xorGate, 1);
c1.connectComponents(x2, xorGate, 2);
c1.connectComponents(x1, andGate, 1);
```

```
c1.connectComponents(x2, andGate, 2);
```

```
// Eingänge schalten  
c1.setInput("x1", 1);  
c1.setInput("x2", 1);
```

2. Vorgefertigten Halbaddierer verwenden

Die Klasse `HalfAdder` ermöglicht es, einen Halbaddierer direkt zu erstellen und in den Schaltkreis einzufügen.

```
class HalfAdder {  
    Circuit<Object> circuit;  
    Input x1;  
    Input x2;  
    Gate xorGate;  
    Gate andGate;  
  
    // Konstruktor nimmt eine bestehende Schaltung und fügt in ihr den HalfAdder ein  
    HalfAdder(Circuit<Object> circuit) {  
        this.circuit = circuit;  
        createHalfAdder();  
    }  
  
    private void createHalfAdder() {  
        // Eingänge erstellen  
        x1 = new Input("x1", 0);  
        x2 = new Input("x2", 0);  
  
        // Gatter erstellen  
        xorGate = new Gate("XOR", "Summe");  
        andGate = new Gate("AND", "Übertrag");  
  
        // Komponenten hinzufügen  
        circuit.addComponent(2, 1, x1);  
        circuit.addComponent(3, 1, x2);  
        circuit.addComponent(2, 3, xorGate);  
        circuit.addComponent(4, 3, andGate);  
  
        // Komponenten verbinden  
        circuit.connectComponents(x1, xorGate, 1);  
        circuit.connectComponents(x2, xorGate, 2);  
        circuit.connectComponents(x1, andGate, 1);  
        circuit.connectComponents(x2, andGate, 2);  
  
        System.out.println("HalfAdder wurde erfolgreich gezeichnet.");  
    }  
}
```

Beispiel:

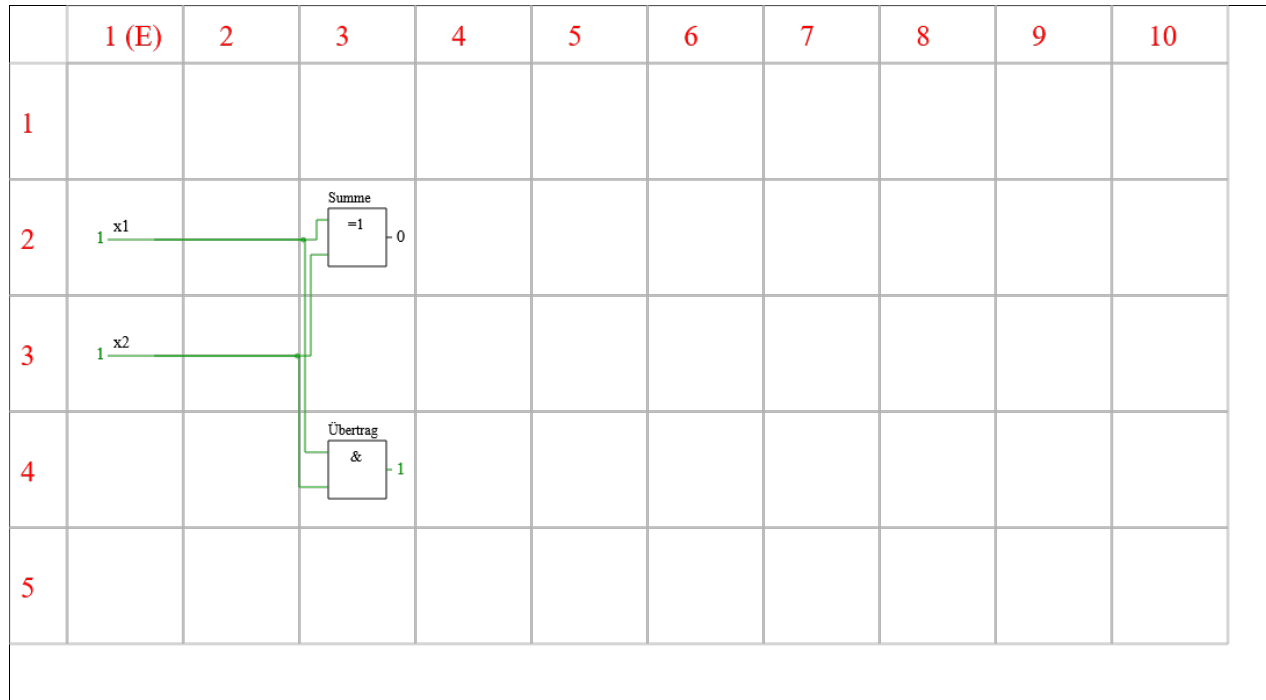
jshell:

```
// Schaltkreis erstellen  
Circuit<Object> c1 = new Circuit<>("Halbaddierer", 10, 5);  
  
// Half-Adder hinzufügen  
HalfAdder halfAdder = new HalfAdder(c1);
```



```
// Eingänge setzen
c1.setInput(halfAdder.x1, 1);
c1.setInput(halfAdder.x2, 1);
```

Turtle bei beiden Beispielen gleich:



Szenario 5 – Speichern und Laden von Schaltungen

Ziel

Das Ziel dieses Szenarios ist es, Nutzern zu ermöglichen, Schaltungen abzuspeichern und später wieder zu laden, um sie weiterzuverwenden und zu bearbeiten. Dies ist besonders praktisch, wenn komplexe Schaltungen wie ein Halbaddierer erstellt wurden und die Arbeit an der Schaltung nach einer Pause fortgesetzt werden soll.

Herausforderung

- Beim Speichern und Laden wurde festgestellt, dass die Referenzen der geladenen Objekte häufig nicht korrekt aktualisiert wurden. Dies führte dazu, dass die Schaltungen zwar optisch korrekt geladen wurden, aber keine Interaktion mit den geladenen Komponenten möglich war.
- Man kann neue Gatter oder Eingänge in die geladene Schaltung einfügen und diese neuen Komponenten weiter bearbeiten, allerdings ist es nicht möglich, diese neuen Komponenten mit den bereits geladenen Komponenten zu verbinden.

Vorgehensweise

1. Speichern der Schaltung

Die Schaltung wird in einer Datei im `.ser`-Format gespeichert, das die Serialisierung der Schaltung ermöglicht. Hierbei werden alle relevanten Informationen wie die Komponenten, Verbindungen und Positionen gespeichert. Die gespeicherte Datei wird im selben Verzeichnis wie die der Java-Datei hinterlegt.

```

class CircuitHandler<T> {
    public void saveCircuit(Circuit<T> circuit, String fileName) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fileName))) {
            oos.writeObject(circuit);
            System.out.println("Schaltung erfolgreich in " + fileName + " gespeichert.");
        } catch (IOException e) {
            System.err.println("Fehler beim Speichern der Schaltung: " + fileName);
            e.printStackTrace();
        }
    }
}

```

2. Laden der Schaltung

Beim Laden einer Schaltung wird die gespeicherte Datei gelesen und die Schaltung in die Live-Ansicht eingefügt. Dabei wird die Methode copyFrom der Circuit-Klasse verwendet, um die geladenen Daten auf das existierende Schaltungsobjekt zu übertragen.

```

class CircuitHandler<T> {
    public void loadCircuit(String fileName, Circuit<T> targetCircuit) {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fileName))) {
            Circuit<T> loadedCircuit = (Circuit<T>) ois.readObject();
            System.out.println("Schaltung erfolgreich aus " + fileName + " geladen.");
            targetCircuit.copyFrom(loadedCircuit);
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Fehler beim Laden der Schaltung: " + fileName);
            e.printStackTrace();
        }
    }
}

```

3. Kopieren der geladenen Schaltung

Die Methode copyFrom stellt sicher, dass die geladenen Komponenten und Verbindungen korrekt in die Live-Ansicht übertragen werden. Dies beinhaltet das Neuzeichnen aller Elemente und das Aktualisieren der Verbindungen.

```

void copyFrom(Circuit<T> loadedCircuit) {
    this.width = loadedCircuit.width;
    this.height = loadedCircuit.height;
    this.maxCols = loadedCircuit.maxCols;
    this.maxRows = loadedCircuit.maxRows;
    this.turtle1 = new Turtle(this.width, this.height);
    this.turtle2 = new Turtle(1600, 1000);
    this.components.clear();

    for (Map.Entry<Point, T> entry : loadedCircuit.components.entrySet()) {
        Point position = entry.getKey();
        T component = entry.getValue();
        // falls das Objekt ein Gate oder Input ist, Referenz aktualisieren
        this.components.put(position, component);
        this.addComponentWithoutChecks(position.y, position.x, component);
        if (component instanceof Gate gate) component = this.getComponentByName(gate.getName());
        else if (component instanceof Input input) component = this.getComponentByName(input.getInputName());
        System.out.println(component + " an Position (" + position.y + ", " + position.x + ") hinzugefügt.");
    }

    this.connections.clear();
    this.firstInputPositions.clear();
    this.secondInputPositions.clear();
}

```

```
this.outputPositions.clear();

restoreConnections(loadedCircuit);
}
```

Beispiel:

jshell:

```
// Schaltung erstellen
Circuit<Object> c1 = new Circuit<>("Half-Adder", 15, 6);
HalfAdder halfAdder = new HalfAdder(c1);

// Schaltung speichern und die Ansicht zurücksetzen
CircuitHandler<Object> handler = new CircuitHandler<>();
handler.saveCircuit(c1, "HalfAdder.ser");
Clerk.clear()

// Schaltung laden
Circuit<Object> c2 = new Circuit<>("Half-Adder", 15, 6);
CircuitHandler<Object> handler = new CircuitHandler<>();
handler.loadCircuit("HalfAdder.ser", c2);
```

Fazit

Beim Speichern und Laden der Schaltung wurde festgestellt, dass die geladenen Schaltungen zwar optisch korrekt dargestellt werden, jedoch keine weitere Interaktion mit den geladenen Komponenten möglich ist. Dies bedeutet, dass Eingaben nicht geschaltet und keine neuen Verbindungen mit den geladenen Komponenten erstellt werden können. Allerdings ist es möglich, neue Gatter oder Eingänge in die geladene Schaltung einzubauen und mit diesen weiterzuarbeiten. Jedoch können diese neuen Komponenten nicht mit den bereits geladenen Komponenten verbunden werden. Dieses Verhalten ist auf das Problem zurückzuführen, dass die Referenzen der geladenen Objekte nicht korrekt aktualisiert werden. Dadurch bleibt die Schaltung nur teilweise nutzbar und stellt eine bekannte Herausforderung bei der Arbeit mit serialisierten Objekten und generischen Typen dar.

Weitere nützliche Funktionen

1. deleteCircuit()

Diese Funktion setzt die gesamte Schaltung zurück. Alle Komponenten, Verbindungen und Positionen werden gelöscht, und die Zeichenfläche wird vollständig geleert.

2. addColumn(int count)

Mit dieser Funktion kann die Anzahl der Spalten in der Schaltung dynamisch erhöht werden. Der Parameter `count` gibt an, wie viele zusätzliche Spalten hinzugefügt werden sollen.

3. addRow(int count)

Ähnlich wie `addColumn()` ermöglicht diese Funktion das Hinzufügen zusätzlicher Reihen. Der Parameter `count` gibt an, wie viele Reihen zur aktuellen Schaltung hinzugefügt werden sollen.

4. printConnections()

Diese Funktion gibt alle derzeitigen Verbindungen der Schaltung in der Konsole aus. Sie zeigt an, welche Komponenten miteinander verbunden sind, inklusive der Details zu den Eingängen und Ausgängen.

