

Programmieren

Wintersemester 2015/16, Prof. Dr. Jörg R. Weimar, Ostfalia, Fakultät Informatik



Laboraufgaben WS 2015/16

Organisatorisches

- Im Labor zu Programmieren werden insgesamt vier Aufgaben gestellt. In jeder zweiten Woche ist eine Aufgabe fällig.
- Die Aufgaben sollen in Zweiergruppen gelöst werden. Die Anmeldung zum Labor erfolgt über <https://laborserver1.f-i.ostfalia.de> (nur aus dem Hochschulnetz).
- Die Abgabe besteht aus zwei Teilen: zum einen wird die in das SVN-Repository der Gruppe (<https://code.ostfalia.de/svn/i-progws2015/GruppeX> wobei X die Gruppennummer ist) eingestellte Lösung automatisch bewertet (JUnit- und Checkstyle-Tests), zum anderen müssen beide Teammitglieder zur Abgabe anwesend sein und die Lösung erklären und bei Nachfrage modifizieren können.
 - bei der Checkstyle-Prüfung darf **kein** Fehler gemeldet werden.
 - es dürfen **maximal 4 Warnungen** gemeldet werden.
 - Bei den JUnit-Tests darf je nach Aufgabe eine gewisse Anzahl von Fehlern nicht überschritten werden. Sie erhalten für einige Aufgaben fertige Tests, um dies selbst prüfen zu können.
- Eine korrekte Abgabe einer Laboraufgabe umfasst das geforderte Programm (mit Dokumentation als Klassendiagramm und in JavaDoc), sowie mehreren (selbst erstellten) Testfällen, alles abgelegt in Ihrem SVN Repository. Sie erklären Ihre Lösung dem Dozenten bzw. Betreuer, demonstrieren das Programm und führen eventuell geforderte Modifikationen aus. Eine der Abgaben muss zwingend bei Prof. Weimar geschehen.
- Wenn Sie **alle** Laboraufgaben erfolgreich bearbeitet haben, dann bekommen Sie 20% der möglichen Punkte für die Note. Weitere 10% erhalten Sie durch Bearbeiten der elektronischen Aufgaben, 70% in der Klausur. Eine Laboraufgabe darf verspätet abgegeben werden, Sie müssen dann aber mit Punktabzügen rechnen.
- Sie verwenden bitte als Projektnamen "ProgWS15Ax", wobei x die Aufgabennummer ist (1..4) und als Paketnamen verwenden Sie "de.ostfalia.prog.ws15". Die zu verwendenden Klassennamen sind in den Aufgabestellungen beschrieben.


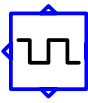
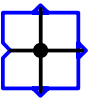
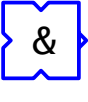

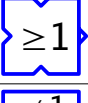
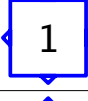
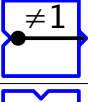

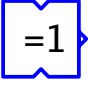

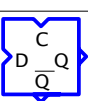
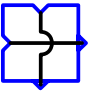
Terminplan: Späteste Abgabe

<i>Aufgabe</i>	<i>Prog</i>
Einarbeitung	29.9.2015
1	13.10.2015
2	27.10.2015
3	10.11.2015
4	24.11.2015
Reserve	8.12.2015

Ziel

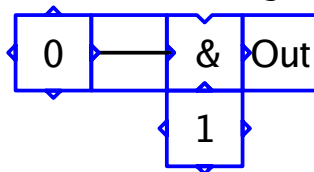
Ziel des Labors ist die Programmierung einer einfachen Simulation von logischen Schaltungen. Wir verwenden Bausteine wie Draht, Und, Oder, Nicht, Xor, Verbindung, Eingabe, Ausgabe, Kreuzung, Flip-Flop. Diese Bausteine werden auf einem zweidimensionalen Quadratgitter angeordnet. Jeder Baustein hat an einigen der vier Seiten einen Eingang oder einen Ausgang. Die Verbindung zwischen zwei Bausteinen kann den Wert 0, den Wert 1 oder den Wert *offen* annehmen. Die Simulation erfolgt dadurch, dass in einem Aktualisierungsschritt jeder Baustein auf dem Gitter den Zustand seiner Eingänge liest und je nach Funktion den Zustand seiner Ausgänge ändert. Dabei werden zuerst alle Bausteine auf den hellen Positionen eines Schachbrettes ($(x+y)\%2 == 0$) aktualisiert, dann alle Bausteine auf dunklen Positionen ($(x+y)\%2 == 1$). Die Koordinaten eines Bausteines sind als (Zeile, Spalte) angegeben, wobei Zeilen nach unten, Spalten nach rechts gezählt werden. Die Größe des Gitters wird zu Beginn der Simulation festgelegt. Dann werden Bausteine platziert, schließlich kann simuliert werden.

Es kommen folgende Bausteine vor, nicht alle müssen in der ersten Aufgabe programmiert werden:

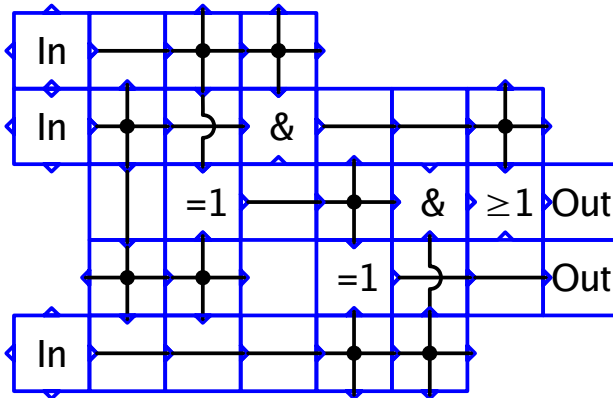
Kürzel	Bild	Beschreibung			
W		Draht (Wire), gerichtet.	F		Signalgenerator, wechselt alle 5 (Five) Updates den Zustand (0/1), Start mit 0.
C		Kreuzung (Cross) verbindet einen Eingang mit drei Ausgängen.	A		Und (And)-Verknüpfung. Offene Eingänge zählen nicht mit.
Z		Null (Zero) ist immer 0.	R		Oder-Verknüpfung. Offene Eingänge zählen nicht mit.
E		Eins ist immer 1.	N		Nicht (Not). Invertiert den Eingang.
I		Input, kann mit einer Methode auf 0 oder 1 gesetzt werden.	X		Exklusiv-Oder (Xor). Ist 1, wenn beide Eingänge unterschiedliche Werte haben.
O		Output. Alle Zustände der O-Zellen können zusammengesammelt und ausgegeben werden.	D		D-Flip-Flop. Wenn C=1 ist, wird der Wert von D gespeichert und an Q ausgegeben, \bar{Q} ist das Komplement von Q. Wenn C=0 ist, behält Q seinen Wert.
B		Brücke, verhält sich wie zwei Drähte (Wire), die unabhängige Werte annehmen können.	–		Leere Zelle

Einige der Bausteine haben eine definierte Ausrichtung. Diese Bausteine sollen in alle vier Richtungen gedreht angeordnet werden können. Dabei ist die oben angegebene Richtung mit 0 bezeichnet, 1 ist um 90° nach links (mathematisch positiv) gedreht, 2 um 180° gedreht, etc. Die Bausteine lesen nur auf den Eingängen. Wenn zwei Ausgänge aneinander liegen, so muss dies nicht separat als Konflikt behandelt werden.

Eine einfache Schaltung wäre die folgende:



Eine komplexere Schaltung ist der Volladdierer:



Vorgaben

In diesem Labor sollen Sie die Klassenstruktur für Ihre Simulation selbst festlegen und erarbeiten. Daher wird nur folgendes festgelegt: Es gibt ein Interface `Grid`, welches Sie in einer eigenen Klasse implementieren müssen:

```
package de.ostfalia.prog.ws15;
public interface Grid {
    class IllegalCellTypeException extends Exception {
        private static final long serialVersionUID = 1L;
        public IllegalCellTypeException(String message) {
            super(message);
        }
    }
    public abstract void update();
    public abstract String getOutputs();
    public abstract void setInputs(String in);
    public abstract String getCellString(int row, int column);
    public abstract void createCell(int row, int column,
        char type, int orientation) throws IllegalCellTypeException;
}
```

Dabei ist `createCell` eine Methode, die eine Zelle vom Typ `type` (ein Buchstabe, siehe Tabelle) an der Stelle `(row, column)` in der angegebenen Orientierung anlegt. `setInputs` bekommt einen String aus 0en und 1en und belegt die Input-Felder mit diesen 0en und 1en in der Reihenfolge, in der die Input-Felder erzeugt worden waren. `getOutputs` ermittelt analog die Belegung der Output-Felder und erzeugt einen String aus 0en und 1en. Die Methode `getCellString` liefert eine Zeichenkette, die die Zelle beschreibt. Sie besteht aus einem Zeichen für den Typ, einer 0 oder 1 (oder '-', wenn offen), und einer Ausrichtung (>^<v). Beispiele sind: „-“ (leere Zelle), „A0 >“ (And mit Zustand 0 und Standard-Ausrichtung nach rechts), „W1 ^“ (Draht/Wire mit Zustand 1 und Ausrichtung nach oben), „X- v“ (Xor mit offenem Ausgang (Startwert oder beide Eingänge offen) und Ausrichtung nach unten). Je Klasse sollte auch die `toString()`-Methode sinnvoll überschrieben werden.

Weiterhin ist vorgegeben, dass Sie eine Klasse `GridFactory` erstellen müssen, die mit jeder Aufgabe weitere Methoden bekommt. In Aufgabe 1 ist es:

```
public class GridFactory {
    public static Grid emptyGrid(int rows, int columns) { ... }
}
```

Aufgabe 0: Einarbeitung in Aufgabenstellung, keine Abgabe (29.9.2015)

Melden Sie sich (in Zweiergruppen) für das Labor an (s.o.). Sie verwenden die Werkzeuge

- eclipse
- SVN (subversion)
- checkstyle

Zu den Werkzeugen gibt es Anleitungen bzw. Filme in LON-CAPA. Erarbeiten Sie sich die Nutzung der Werkzeuge und der Laborrechner.

Entwerfen Sie eine **objektorientierte Struktur** für Ihre Aufgabe. Verwenden Sie eine mindestens zweistufige Vererbungshierarchie, definieren Sie mindestens ein enum (Aufzählungstyp),

Aufgabe 1: Klassenstruktur entwerfen, Implementierung einfacher Zell-Typen

- Projektname: ProgWS15A1
- Paketname: de.ostfalia.prog.ws15.model
- Entwerfen Sie eine Klassenstruktur für die verschiedenen Zellen und das Gitter. Jede Art von Zellen soll in einer eigenen Klasse implementiert sein, aber Code, der in verschiedenen Klassen identisch ist, soll natürlich nur einmal geschrieben sein (Stichwort „Vererbung“). Verwenden Sie bei den Zellen eine mindestens zweistufige, besser dreistufige Vererbungshierarchie.
- Beschreiben Sie Ihre Klassenstruktur durch ein UML-Diagramm, welches Sie im Kolloquium erläutern.
- Definieren Sie mindestens ein enum.
- Definieren Sie eine (oder mehrere) Klasse(n), die das Interface Grid implementieren.

Dabei muss die Methode

```
createCell(int row, int column, char type, int orientation)
```

für diese Aufgabe 1 nur folgende Zell-Typen beherrschen: (W, Z, E, F, A, R, N, C).

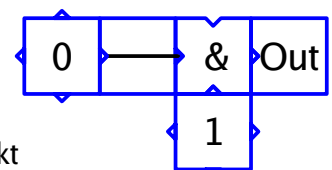
Entsprechend müssen auch zunächst nur für diese Zelltypen Implementierungen realisiert werden.

- Implementieren Sie eine Klasse

```
public class GridFactory {
    public static Grid emptyGrid(int rows, int columns) { ... }
}
```

- deren einzige Methode ein leeres Gitter der angegebenen Größe erzeugt.

- Testen Sie Ihre Gitter und Ihre Zellen mit einfachen Tests, wie dem Beispiel



- In dieser Aufgabe müssen Sie die Methoden setInputs und getOutputs, die vom Interface Grid gefordert werden, nicht korrekt implementieren. Schreiben Sie dummy-Methoden, die nichts tun.
- Wichtig ist, dass die Methode update() der Klasse Grid den Zellzustand korrekt ändert,

und dass `getCellString(int row, int column)` korrekte Werte liefert, denn diese Werte werden getestet. Ein Beispiel für einen Test ist:

@Test

```
public void testWireExplicitRight3() throws IllegalCellTypeException {
    Grid g1 = GridFactory.emptyGrid(1, 4);
    g1.createCell(0, 0, 'E', 0);
    g1.createCell(0, 1, 'W', 0);
    g1.createCell(0, 2, 'W', 0);
    g1.createCell(0, 3, 'W', 0);
    String res = g1.getCellString(0, 3);
    assertEquals("Grid with wire cell should initially be", "W->", res);
    g1.update();
    String res2 = g1.getCellString(0, 3);
    assertEquals("Grid with wire cell should after one update still be", "W->", res2);
    g1.update();
    String res3 = g1.getCellString(0, 3);
    assertEquals("Grid with wire cell should after 2 updates be", "W1>", res3);
}
```

- Speichern Sie Ihr Projekt im SVN, es wird dann automatisch getestet.

Aufgabe 2: Implementierung weiterer Zell-Typen

- Projektname: ProgWS15A2
- Paketnamen: `de.ostfalia.prog.ws15`, `de.ostfalia.prog.ws15.model` und `de.ostfalia.prog.ws15.view`
- Ergänzen Sie die Implementierung aus Aufgabe 1 um die weiteren Zelltypen (B, H, X, I, O)
- Stellen Sie sicher, dass die Zellen auch in anderen Orientierungen als der Standardorientierung funktionieren. Dies wurde in Aufgabe 1 nicht getestet, wird nun aber geprüft.

Kommandozeilen-Interface

- Schreiben Sie eine Klasse `de.ostfalia.prog.ws15.view.CommandLine`, die ein Kommandozeilen-orientiertes Benutzerinterface bereitstellt.

```
package de.ostfalia.prog.ws15.view;
import java.io.InputStream;
import java.util.Scanner;
import de.ostfalia.prog.ws15.model.Grid;
...
public class CommandLine {
    static Grid execute(InputStream is) {
        ArrayGrid grid = null;
        try(Scanner sc = new Scanner(is)) {
            while (sc.hasNextLine()){
                String line = sc.nextLine(); ...
            }
        }
    }
}
```

- Das Kommandozeileninterface ist ganz simpel: Jede Zeile enthält ein Kommando, welches mit einem Kleinbuchstaben beginnt (g, c, s, r, i, o, p, q). Dann folgen je nach Kommando andere Zeichen:

- g (bedeutet grid), gefolgt von der Größe in Zeilen, Spalten. Beispiel: g 5,5
- c (Cell), gefolgt vom Buchstabentyp der Zelle (Großbuchstaben), dann die Orientierung (0..3), dann die Position (Zeile, Spalte), in der die Zelle angelegt werden soll. Beispiel: cW0 3,3 oder cA1 2,3 oder cE0 4,3
- s (Step) ruft einmal die Methode update des Gitters auf.
- r (Run) gefolgt von einer natürlichen Zahl, ruft die Methode update so oft auf, wie die Zahl angibt.
- i (Input) gefolgt von einer binären Zeichenkette aus 0en und 1en setzt die Input-Zellen auf die entsprechenden Werte. Dabei ist beschreibt das erste Zeichen den Zustand derjenigen Input-Zelle, die zuerst im Gitter eingefügt worden war. Beispiel: i 0100
- o (Output) druckt in einer binären Zeichenkette alle Zustände der Output-Zellen, Reihenfolge wie bei Input.
- p (Print) gibt das gesamte Gitter auf Standard-Out aus.
- q (wie spspsp...) gefolgt von einer Zahl macht so viele Updates, und nach jedem Update ein print, wie der Parameter angibt.
- # Kommentarzeile, wird ignoriert.
- x (Exit) verläßt den Kommandozeileninterpreter und liefert das Grid-Objekt im aktuellen Zustand zurück

```
# Beispieleingabe
g 4,4
cF0 0,0
cC0 0,1
cN0 0,2
cA0 0,3
cE0 1,0
cR0 1,1
cW0 1,2
cC0 1,3
cF0 2,0
cW0 2,1
cW0 2,2
cB0 2,3
cW3 3,3
p
s
p
r 3
p
q 15
letzte Ausgabe:
[[F1  , C1 >, N0 >, A0 >],
 [E1  , R1 >, W1 >, C1 >],
 [F1  , W1 >, W1 >, B11>],
 [--  , --  , --  , W1 v]]
```

- Testen Sie das Kommandozeileninterface interaktiv. Überlegen Sie sich, welche Schaltungen Sie testen möchten.

Aufgabe 3: Graphisches User-Interface

- Projektname: ProgWS15A3
- Paketnamen: `de.ostfalia.prog.ws15.model` und `de.ostfalia.prog.ws15.view`

Entwerfen und implementieren Sie ein graphisches User-Interface (mit JavaFX)!

Bilder für die einzelnen Zellen sind in LON-capa zu finden (Bilder.zip). jeweils 100x100 Pixel, davon außen 10 Pixel für die Ein/Ausgabe-Pfeile.

Funktionalitäten:

- Grid interaktiv erstellen
- Simulationsschritt ausführen
- Status aller Zellen sehen

Aufgabe 4: Graphisches User-Interface verbessern

- Projektname: ProgWS15A4

Fügen sie folgende Funktionalitäten zu Ihrem User-Interface hinzu:

1. Kommandos von einer Datei einlesen, mit einer Syntax wie beim Kommandozeileninterpreter aus Aufgabe 2.
2. Eine Simulation starten und laufen lassen, die von einem separaten Thread ausgeführt wird.
3. Ein Menu verwenden.