Exercise `03_logic_sim`

Dr. Günter Kolousek

28. Januar 2019

Dieses Beispiel soll (letztendlich) ein kleines Simulationsprogramm entstehen lassen, das es uns erlaubt, logische Schaltungen (einfach) zu simulieren.

1 Allgemeine Instruktionen

- Just follow the instructions of the previous excercise… **but** now we are creating several assemblies within our project. Hence, just create a bare directory called `03_logic_sim` beforehand and **inside** this directory we will create several .NET Core project!
- In addition to the already well-known facts, the purpose of this and the other excercises is not in the stupid and boring copying of program text or stupidly following instructions. It's clear as daylight, you are a 4th year student and it's **your** own responsibility to train your programming skills!

2 Überblick

- Die Idee ist eine Klassenbibliothek und zwei Programme zu erstellen, die die erstellte Klassenbibliothek verwenden.
- Das erste Programm soll eine kleine Applikation sein, die für verschiedene Logiken die Junktoren jeweils als Tabelle ausgibt.
- Das zweite Programm soll dann die eigentliche Simulationsumgebung für unsere Schaltungen darstellen.

3 Nun zum Programmieren!

Für dieses Beispiel (und auch die folgenden) gilt, dass die Klassen sich in einem Namespace befinden müssen, der auf deine Matrikelnummer lautet. Prinzipiell soll sich jede Klasse in einer eigenen Datei befinden, auch wenn dies in C# nicht unbedingt notwendig ist.

3.1 Erstellen und Verwenden einer Klassenbibliothek in C#

1. Lege in deinem Beispielverzeichnis jetzt das erste .NET Core Projekt für dieses Beispiel an:

   ```
   $ dotnet new classlib --name logics
   ```

   Wie gewohnt werden wir alle C# Artefakte in einem eigenem Verzeichnis `src` erstellen…

2. Im Namespace `logics` werden wir einen weiteren Namespace `propositional` anlegen und dort eine Utility-Klasse mit dem Namen `Operators`, die für jeden Junktor jeweils eine Funktion enthält, die je zwei boolsche Operanden erhält und einen boolschen Wert zurückliefert.

   Die Namen der Funktion sollen lauten: `conj`, `disj`, `neg`, `anti`, `imp`, `equiv`!

3. Diese Klassenbibliothek kann/soll mittels `dotnet build` erstellt werden.

3.2 Erstellen eines Testprogrammes `logic_printer`

Jetzt geht es daran, diese Klassenbibliothek zu testen und dies werden wir mit einem Konsolenprogramm erledigen, das alle Junktorentabellen (siehe Folien) mittels der gerade erstellten Klassenbibliothek auf der Konsole hintereinander ausgibt.

1. Erstelle daher in deinem Beispielverzeichnis (also `03_logic_sim`) wie gewohnt eine .NET Core Konsolenanwendung mit dem Namen `logics_printer`.

2. Jetzt muss allerdings noch die Klassenbibliothek der Konsolenanwendung bekannt gemacht werden. Dies geschieht auf folgende Art und Weise:

   ```
   $ dotnet add reference ../logics/logics.csproj
   ```

3. Schreibe nun ein Hauptprogramm, das die Tabellen folgendermaßen ausgibt:

```
Propositional Logic
===================

 a | b | a & b
---------------
 0 | 0 |   0
 0 | 1 |   0
 1 | 0 |   0
 1 | 1 |   1

 a | b | a | b
---------------
 0 | 0 |   0
 0 | 1 |   1
 1 | 0 |   1
 1 | 1 |   1

 a | b | a ^ b
---------------
 0 | 0 |   0
 0 | 1 |   1
 1 | 0 |   1
 1 | 1 |   0

 a | !a
--------
 0 |  1
 1 |  0

 a | b | a -> b
---------------
 0 | 0 |   1
 0 | 1 |   1
 1 | 0 |   0
 1 | 1 |   1

 a | b | a <-> b
---------------
 0 | 0 |   1
 0 | 1 |   0
 1 | 0 |   0
 1 | 1 |   1
```

Beachte allerdings das bekannte DRY (don't repeat yourself) Prinzip!

Hier ein paar Tipps:

- Klarerweise verwendest du die entwickelte Klasse `Operators`!
- Da es in C# keine freien Funktionen gibt, ist dies wieder eine gute Anwendungsmöglichkeit für eine Utility-Klasse. Als Name würde sich `Printer` anbieten.
- Wie du siehst verwenden wir hier `0` für `false` und `1` für `true` wie es oft verwendet wird, kompakter zu lesen ist und auch leicht in Schleifen verwendet werden kann. Damit muss natürlich auch zwischen den Datentypen konvertiert werden. Ein casten wie in `C++` ist nicht möglich… Du musst/kannst hierfür die Klasse `System.Convert` mit den Methoden `ToBoolean` und `ToInt32` verwenden.
- Verwende Formatstrings!
- Das Drucken des Headers der Tabelle kann sicher abgespalten werden (→ DRY).
- Das Drucken einer einzelnen Tabelle kann auch allgemein gelöst werden. Lediglich die eigentliche Operation muss getrennt übergeben werden. Hier helfen Lambdaausdrücke und der Typ `System.Func`. Damit kann/soll die Signatur zum Drucken der Tabelle folgendermaßen aussehen:

  ```
  static void print_table(Func<bool, bool, bool> op) {
  ```

  Beim Aufruf… sind Lambdaausdrücke zu verwenden!

3.3 Erweitern um Logik L_3

Jetzt wird unsere Klassenbibliothek um eine Lukasiewicz L_3 Logik erweitert und außerdem unser Programm `logic_printer` entsprechend erweitert:

1. Entwickle in einer Datei `lukasiewicz3.cs` und den Namensraum `lukasiewicz3` wieder in einer Klasse `Operators` die entsprechenden Funktionen.

   Um etwas Neues zu lernen (und auch die Schwierigkeiten damit zu erkennen), werden wir die Signaturen der Funktionen folgendermaßen gestalten:

   ```
   static bool? conj(bool? op1, bool? op2) {
   ```

Du siehst, dass der dritte Wert hier offensichtlich durch den Wert `null` gekennzeichnet wird. Damit gehen wir in diesem konkreten Fall den Weg wie dies auch in SQL gehandhabt wird (bzgl. der Semantik siehe jedoch die Folien).

Als **Einschränkung** gilt: Es ist für uns keine Konvertierung in eine Zahl erlaubt und weiters müssen `if` Anweisungen verwendet werden!!! Warum? Weil du lernen sollst mit "nullable" Typen umzugehen und auch die Tabellen auszulesen und in `if` Anweisungen auszudrücken.

Hier wieder ein paar Tipps:

- Für die Antivalenz soll gelten: $a \underline{\vee} b \Leftrightarrow (a \vee b) \wedge \neg(a \wedge b)$
- Für die Äquivalenz soll gelten: $a \leftrightarrow b \Leftrightarrow (a \rightarrow b) \wedge (b \rightarrow a)$

2. Jetzt ist es wieder an der Zeit, die Tabellen für die Logik L_3 auszugeben. Dies geschieht wieder im Programm `logic_printer`. Dort haben wir derzeit eine Klasse `Printer`, die das Ausdrucken der Tabellen für die Aussagenlogik erledigt. Dort die neuen Funktionen anzusiedeln wäre nicht sinnvoll, allerdings ist die Klasse mit dem Namen `Printer` nicht so schlecht gewählt… Was ist zu tun?

   Ok, ein Refactoring muss zuerst wieder her: Packe die ursprüngliche Klasse `Printer` in einen Namensraum `propositional_logic` (als Unternamensraum von `logic_printer`) und lege einen weiteren Namensraum `lukasiewicz3` als Unternamensraum von `logic_printer` an und du hast wieder den Namen `Printer` als Klassennamen zur Verfügung.

   Wenn du damit fertig bist, dann sollte die Ausgabe deines Programmes um die folgenden Ausgaben erweitert worden sein:

```
Lukasiewicz L3
==================

 a | b | a & b
---------------
 0 | 0 |   0
 0 | ½ |   0
 0 | 1 |   0
 ½ | 0 |   0
 ½ | ½ |   ½
 ½ | 1 |   ½
 1 | 0 |   0
 1 | ½ |   ½
 1 | 1 |   1
```

a	b	a	b
0	0		0
0	½		½
0	1		1
½	0		½
½	½		½
½	1		1
1	0		1
1	½		1
1	1		1

a	b	a ^ b
0	0	0
0	½	½
0	1	1
½	0	½
½	½	½
½	1	½
1	0	1
1	½	½
1	1	0

a	!a
0	1
½	½
1	0

a	b	a -> b
0	0	1
0	½	1
0	1	1
½	0	½
½	½	1
½	1	1
1	0	0
1	½	½
1	1	1

a	b	a <-> b

```
0 | 0 |   1
0 | ½ |   ½
0 | 1 |   0
½ | 0 |   ½
½ | ½ |   1
½ | 1 |   ½
1 | 0 |   0
1 | ½ |   ½
1 | 1 |   1
```

Auch hier wieder einige Tipps:

- Jetzt dürfen ja keine Konvertierungen verwendet werden und damit auch keine Zahlen. Wie können jetzt die Schleifen realisiert werden?

  Unter Zuhilfename von Arrays mit den richtigen Werten. Dann kann leicht mittels eines `foreach` über die Werte des Arrays iteriert werden.

- Eine Ausnahme bzgl. des Konvertieren muss es natürlich geben, da wir bei der Ausgabe ja 0 resp. 1 für `false` resp. `true` ausgeben wollen (bzgl. ½ siehe nächsten Punkt). Schreibe am besten eine Hilfsfunktion, die die Konvertierung eines `bool?` in eines unserer 3 Zeichen übernimmt.

- Wo kommt nur das "½" her? Beziehungsweise wie kann man dieses ausgeben? Es handelt sich um ein Unicode-Zeichen mit dem Codepoint U+00BD! Unter Linux ist die Konsole in der Regel UTF-8 fähig und auch dementsprechend eingestellt, um Unicode-Zeichen in UTF-8 auszugeben.

  Wichtig ist, dass dein Editor so eingestellt ist, dass dieser den Programmtext in UTF-8 abspeichert! `dotnet new` legt die Dateien standardmäßig UTF-8 kodiert ab, aber das darf natürlich nicht geändert werden und neue Dateien müssen ebenfalls so angelegt und abgespeichert werden.

  Jetzt muss also nur noch mehr dieses Zeichen im Editor richtig eingegeben werden. Dazu benötigst du einen vernünftigen Editor, in meinem funktioniert das problemlos: `CTRL-x 8 RET 00BD RET` (oder `M-x insert-char RET 00bd RET` oder…) und fertig `;-)` Wie es in deinem funktioniert, musst du selber herausfinden! Wenn es partout nicht klappen will, dann hilft dir vielleicht auch `https://graphemica.com/` weiter.

3.4 Erweitern um Logik L_n

Jetzt wird unsere Klassenbibliothek um eine Lukasiewicz L_n Logik erweitert und außerdem unser Programm `logic_printer` entsprechend erweitert:

1. Entwickle in einer Datei `lukasiewiczn.cs` und den Namensraum `lukasiewiczn` wieder in einer Klasse `Operators` die entsprechenden Funktionen.

   Da es bei L_n ja um rationale Zahlen $0, \frac{1}{n-1}, \frac{2}{n-1}, ..., \frac{n-2}{n-1}, 1$ geht, werden wir der Einfachheit Gleitkommazahlen verwenden.

   ```
   static double conj(double op1, double op2) {
   ```

   Damit wird die Ermittlung der Werte auch ensprechend einfacher, da direkt gerechnet werden kann.

2. Jetzt ist es wieder an der Zeit, die Tabellen auszugeben. Beispielhaft wollen wir dies für L_4 erledigen. Dies geschieht wieder im Programm `logic_printer`. Entwickle analog zu L_3. Die Ausgabe sollte folgendermaßen aussehen:

```
 Lukasiewicz L4
==================

   a   |   b   | a & b
-----------------------
 0.000 | 0.000 | 0.000
 0.000 | 0.333 | 0.000
 0.000 | 0.667 | 0.000
 0.000 | 1.000 | 0.000
 0.333 | 0.000 | 0.000
 0.333 | 0.333 | 0.333
 0.333 | 0.667 | 0.333
 0.333 | 1.000 | 0.333
 0.667 | 0.000 | 0.000
 0.667 | 0.333 | 0.333
 0.667 | 0.667 | 0.667
 0.667 | 1.000 | 0.667
 1.000 | 0.000 | 0.000
 1.000 | 0.333 | 0.333
 1.000 | 0.667 | 0.667
 1.000 | 1.000 | 1.000

   a   |   b   | a | b
-----------------------
 0.000 | 0.000 | 0.000
 0.000 | 0.333 | 0.333
 0.000 | 0.667 | 0.667
 0.000 | 1.000 | 1.000
 0.333 | 0.000 | 0.333
```

```
 0.333 | 0.333 | 0.333
 0.333 | 0.667 | 0.667
 0.333 | 1.000 | 1.000
 0.667 | 0.000 | 0.667
 0.667 | 0.333 | 0.667
 0.667 | 0.667 | 0.667
 0.667 | 1.000 | 1.000
 1.000 | 0.000 | 1.000
 1.000 | 0.333 | 1.000
 1.000 | 0.667 | 1.000
 1.000 | 1.000 | 1.000
```

```
   a   |   b   | a ^ b
-----------------------
 0.000 | 0.000 | 0.000
 0.000 | 0.333 | 0.333
 0.000 | 0.667 | 0.667
 0.000 | 1.000 | 1.000
 0.333 | 0.000 | 0.333
 0.333 | 0.333 | 0.333
 0.333 | 0.667 | 0.667
 0.333 | 1.000 | 0.667
 0.667 | 0.000 | 0.667
 0.667 | 0.333 | 0.667
 0.667 | 0.667 | 0.333
 0.667 | 1.000 | 0.333
 1.000 | 0.000 | 1.000
 1.000 | 0.333 | 0.667
 1.000 | 0.667 | 0.333
 1.000 | 1.000 | 0.000
```

```
   a   |  !a
---------------
 0.000 | 1.000
 0.333 | 0.667
 0.667 | 0.333
 1.000 | 0.000
```

```
   a   |   b   | a -> b
-----------------------
 0.000 | 0.000 | 1.000
 0.000 | 0.333 | 1.000
 0.000 | 0.667 | 1.000
 0.000 | 1.000 | 1.000
```

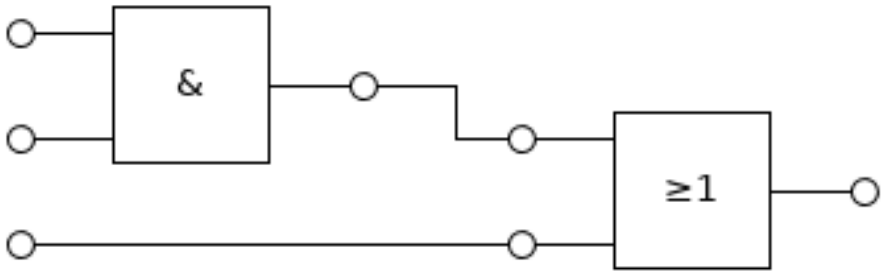
```
 0.333 | 0.000 | 0.667
 0.333 | 0.333 | 1.000
 0.333 | 0.667 | 1.000
 0.333 | 1.000 | 1.000
 0.667 | 0.000 | 0.333
 0.667 | 0.333 | 0.667
 0.667 | 0.667 | 1.000
 0.667 | 1.000 | 1.000
 1.000 | 0.000 | 0.000
 1.000 | 0.333 | 0.333
 1.000 | 0.667 | 0.667
 1.000 | 1.000 | 1.000

   a   |   b   | a <-> b
-----------------------
 0.000 | 0.000 | 1.000
 0.000 | 0.333 | 0.667
 0.000 | 0.667 | 0.333
 0.000 | 1.000 | 0.000
 0.333 | 0.000 | 0.667
 0.333 | 0.333 | 1.000
 0.333 | 0.667 | 0.667
 0.333 | 1.000 | 0.333
 0.667 | 0.000 | 0.333
 0.667 | 0.333 | 0.667
 0.667 | 0.667 | 1.000
 0.667 | 1.000 | 0.667
 1.000 | 0.000 | 0.000
 1.000 | 0.333 | 0.333
 1.000 | 0.667 | 0.667
 1.000 | 1.000 | 1.000
```

D.h. nutze die Möglichkeiten der Formatierungsstrings von C# aus!

3.5 Logiksimulator

In weiterer Folge werden wir ein weiteres Programm mit dem Namen `logsimy` entwickeln, das es uns ermöglichen wird (einfache) logische Schaltungen zu simulieren.

Ein Beispiel für solch eine logische Schaltung wäre:



1. Lege daher ein weiteres .NET Core Projekt in deinem Beispielverzeichnis an, das `logsimy` heißt und vergiss nicht, dass die Klassenbibliothek `logics` zu diesem .NET Core Projekt hinzugefügt werden muss.

2. Bevor wir die einzelnen Gatter implementieren, werden wir eine Klasse implementieren, die es uns erlaubt, Änderungen zwischen den Gattern weiterzureichen. Die Änderung wird zwischen je einem Ausgang und je einem Eingang von einem Gatter zum nächsten Gatter weitergereicht.

   Jeden Eingang bzw. Ausgang wollen wir als eine Art Variable ansehen (da sich die Werte ja ändern können). Jede Variable soll in weiterer Folge Änderungen an andere Variable weiterreichen können.

   Gut, jetzt wollen wir ein bisschen konkreter werden und zum Programmieren übergehen. Implementiere eine Klasse `Variable`, die über einen Namen (`name`) und einen boolschen Wert (`value`) verfügt. Für Name und Wert bieten sich wieder Properties an. Diese können vorerst durchaus "auto implemented" sein. Der Name sollte der Einfachheit halber im Konstruktor *optional* zur Verfügung gestellt werden können.

   Als (Unter-)Namensraum wäre `variables` nicht so schlecht, aber YMMV…

   Schreibe weiters im Hauptprogramm Testcode, der zwei Variable `v1` und `v2` anlegt und den Wert der ersten Variable setzt und den Wert der zweiten Variable auf der Konsole ausgibt. Klarerweise wird auf der Konsole `False` erscheinen, da diese beiden Variablen noch nicht miteinander verbunden sind und damit auch die Änderung der ersten Variable an die zweite Variable weitergegeben wird.

   Auf zum nächsten Punkt!

3. Ok, da ist eigentlich noch nicht so richtig viel passiert, denn eigentlich wollen wir ja Änderungen von einer Variable zu einer anderen Variable weiterreichen. Wir

werden uns dazu des Observer-Patterns bedienen, für das es in C# eine direkte Unterstützung in von Delegates und Events gibt.

Zuerst benötigen wir ein Delegate und dazu bietet sich folgende Deklaration an:

```
public delegate void NotifyHandler(bool value);
```

Um damit etwas anzufangen, benötigen wir noch einen Event:

```
public event NotifyHandler notify;
```

Ok, wenn sich der Wert von `value` ändert, dann wollen wir das Event `notify` feuern, womit wir in `set` des Property folgenden Code schreiben müssen:

```
// must be checked
// if no event handler registered then no call is possible...
if (notify != null)
    notify(value);
```

Klar, dass damit das ”auto implemented” für `set` hinfällig geworden ist. Aber so ist es nun einmal das Leben…

Noch etwas ist zu beachten: Wenn sich der Wert **nicht** geändert hat, dann ist dieser auch nicht neu zu setzen und damit soll auch keine Änderungs-Nachricht verschickt werden. Baue dies in `set` noch ein!!!

Fein, jetzt ist im Hauptprogramm nur mehr ein entsprechender Handler zu implementieren, der die erste Variable mit der zweiten Variable verbindet, so etwas wie der folgende Code sollte eigentlich den gewünschten Zweck erfüllen:

```
v1.notify += (v => v2.value = v;);
```

Damit sollte eigentlich das Weiterreichen einer Änderung von einer Variable zu einer anderen Variable prinzipiell möglich sein, aber…

4. Schon etwas mühsam, so mit `notify + ...`, nicht wahr? Gehe daher her und schreibe eine Utility-Klasse `Utilities` im Namensraum `variables`, die eine Funktion `connect` beinhaltet, sodass zwei Variablen im Hauptprogramm auf folgende Art und Weise ”verbunden” werden können:

   ```
   variables.Utilities.connect(v1, v2);
   ```

Damit wird das Verbinden zweier Variable aus Anwendersicht unseres ausgefeilten Variablenkonzeptes bequem. Ok, besser geht es schon noch, aber noch ist ja nicht aller Tage Abend.

Trotzdem, du denkst an Commits, oder?

5. Eigentlich handelt es sich um eine feine abgeschlossene Klasse, von der eigentlich nicht mehr abgeleitet werden sollte. D.h. wir werden diese "versiegeln"! Ändere die Klassendefinition dementsprechend ab.

6. Hmm, es ist ja schön und gut, dass wir den neuen Wert erhalten, aber wir haben keine Ahnung welche Variable sich ursprünglich geändert hat. Du meinst, das ist nicht wichtig? Ok, dann schreibe weiters eine Funktion `enable_logging` in unserer Utiltity-Klasse `Utilities`, die für eine Variable einen weiteren Event-Handler hinzufügt, der folgende Ausgabe tätigt:

   ```
   v2: True
   ```

   In diesem konkreten Fall soll es sich bei `v2` klarerweise um den Namen der Variable handeln.

   Der Aufruf dieser Funktion sollte folgendermaßen erfolgen:

   ```
   variables.Utilities.enable_logging(v2);
   ```

   Hmm, das geht nicht? Klar das geht nicht, da wir ja keine Information haben, bei *welcher* Variable die Änderung stattgefunden hat.

   Deshalb ist jetzt Zeit, ein weiteres Refactoring vorzunehmen. Ändere deshalb das Delegate folgendermaßen ab:

   ```
   public delegate void NotifyHandler(Variable v);
   ```

   Der Aufruf des Events sollte dann folgendermaßen erfolgen:

   ```
   if (notify != null)
       notify(this);
   ```

   Klar, jetzt wissen wir bei welcher Variable die Änderung aufgetreten ist und den Wert kennen wir auch…

   Ändere dein Programm so ab, dass alles so funktioniert wie vorher (ein Refactoring eben).

   Commiten!

7. Gut, dann bist du jetzt in der Lage, die Utility-Funktion `enable_logging` entsprechend zu implementieren und auch in das Hauptprogramm einzubauen. Damit kann jedes manuelle `WriteLine` aus unserem Hauptprogramm jetzt verschwinden!

8. In diesem Sinne erweitern wir unsere Klasse `Variable` um eine Methode `void reset(bool v=false)`, die den Wert der Variable wieder zurücksetzt. Auch, wenn es sich beim Zurücksetzen um eine Änderung handelt, ist es doch ein Unterschied, den wir mittels unserer Abstraktion derzeit nicht erkennen können.

   Deshalb ist es wieder an der Zeit unser Delegate wieder anzupassen:

   ```
   public delegate void NotifyHandler(Variable v, NotificationReason reason);
   ```

   Bei `NotificationReason` soll es sich um eine Aufzählung handeln, die die Werte `changed` und `reset` aufweist.

   Den Aufruf `notify` noch in `reset` und `set` von `value` anpassen und fertig ist die Geschichte.

   Ok, damit wir es richtig bewundern können, werden wir auch `enable_logging` anpassen, sodass es zu dieser Ausgabe kommt (bei geeigneten Aufruf von `reset` bzw. Setzen von `p1`!):

   ```
   p2: False (reset)
   p2: True (changed)
   ```

9. Jetzt geht es daran, die einzelnen Gatter zu implementieren. Kommentiere dazu zuerst alle Anweisung im Hauptprogramm aus, da wir diese in dieser Form nicht mehr benötigen. Natürlich könnten wir diese auch einfach löschen, da wir jede essentielle Änderung in unserem Repository wiederfinden… Trotzdem, in unserem Beispielprojekt werden wir diese Änderungen einfach unter Kommentar setzen.

   Fangen wir mit einem "UND" Gatter an: Ich denke, dass eine eigene Datei `gates.cs` für die Gatter sicher nicht verkehrt ist und auch, dass darin im Namensraum `logsimy` die (Unter-)Namensräume `gates` und `propositional` durchaus ihre Berechtigung haben.

   Nach diesen Vorarbeiten geht es direkt daran eine Klasse `AndGate` anzulegen.

   Es ist es sinnvoll, einem Gatter auch einen Namen zu geben. Hier ist ein normale Instanzvariable ausreichend und werden dies auch so implementieren, allerdings legen wir fest, dass diese nur *gelesen* werden darf! Der Name sollte der Einfachheit halber im Konstruktor *zwingend* zur Verfügung gestellt werden.

Weiters soll `AndGate` über die Variablen `i0`, `i1` und `o` (jeweils vom Typ `Variable`) als Property verfügen, die die beiden Eingänge und den Ausgang des Gatters darstellen. Diese sollen jeweils mit dem Namen des Gatters gefolgt von einem Punkt und weiters gefolgt von `i0`, `i1` bzw. `o` benannt werden (wegen der Ausgabe).

Vergiss nicht einen Konstruktor zu schreiben und die Variablen anzulegen.

Natürlich ist da noch nicht viel passiert, da der Ausgang ja noch nicht in Abhängigkeit der Eingänge gesetzt wird. Weiter davon im nächsten Punkt.

10. Natürlich kann man auch hier die "auto-implemented" Properties umschreiben, dass der Ausgang in Abhängigkeit der Eingänge gesetzt wird. Aber, die Anwendung des Observer-Patterns kann man auch dafür nutzen:

    a) Implementiere ein Interface `Observer` in `variable.cs` mit folgender Methode:

    ```
    void update(Variable v, NotificationReason r);
    ```

    Was soll diese Methode eigentlich tun? Diese soll den Wert des Ausgangs setzen, aber nur dann, wenn der Grund kein Rücksetzen des Gatters ist. Erstens braucht in diesem Fall keine Berechnung erfolgen und zweitens wird der Wert von dem Ausgang beim Reset gesetzt und nicht ermittelt.

    b) Lasse die Klasse `AndGate` dieses Interface implementieren und setze in `update` darin den Ausgang in Abhängigkeit der beiden Eingänge gemäß der Funktion `conj`, allerdings nur, wenn es sich bei der Änderung um kein `reset` gehandelt hat.

    c) Jetzt fehlt noch eine entsprechende Möglichkeit die Variable mit dem Gatter zu "verbinden". Schreibe deshalb in `Utilities` noch eine Funktion mit folgender Signatur:

    ```
    public static void inform(Variable src, Observer obs)
    ```

    Der Inhalt sollte eigentlich klar sein…

    Es ist für `obs` die `update` - Methode entsprechend aufzurufen.

    d) Weiters muss noch im Konstruktor der Klasse `AndGate` die `inform` Funktion jeweils für `i0` und für `i1` aufgerufen werden.

    e) Eine Methode `reset`, die die Variablen zurücksetzt wäre auch nicht schlecht. Go!

11. Teste jetzt mit folgender Konfiguration:

- Variable `i0`, `i1` und `i2`
- UND-Gatter `and1` und `and2`
- Verbinde `i0` mit `and1.i0`, `i1` mit `and1.i1` und `i2` mit `and2.i1`
- Verbinde weiters `and1.o` mit `and2.i0`
- Aktiviere das Logging von `and2.o`
- setze am Schluss die Eingänge `i0`, `i1` und `i2` hintereinander auf `true`

Interpretiere die Ausgabe… Alles klar?

12. Jetzt haben wir *ein* Gatter implementiert, aber wir wollen alle Gatter implementieren. Natürlich kann man einfach den Code duplizieren. Kann man, aber…

    Wenn du dir den Code von `AndGate` so ansiehst und überlegst welche Änderungen für ein `OrGate` notwendig wären, dann siehst du, dass es eigentlich nicht allzu viel ist.

    Ändere daher deinen Code so ab, dass du eine neue abstrakte Klasse `Gate2` (für ein Gatter mit zwei Eingängen) anlegst, das über folgenden Konstruktor verfügt:

    ```
    public Gate2(string name, BooleanOperator2 op)
    ```

    Bei `BooleanOperator2` soll es sich klarerweise um ein entsprechendes Delegate handeln.

    Weiters wird natürlich auch eine abstrakte Klasse `Gate1` für die Negation benötigt.

    Jetzt ist es einfach: Schreibe für jeden Operator eine Subklasse (`AndGate`, `OrGate`, `XOrGate`, `NegGate`, `XNOrGate` (Äquivalenz)) von `Gate1` bzw. `Gate2`, die nur einen entsprechenden Konstruktor verfügt. Vergiss nicht, dass es durchaus sinnvoll sein kann, wenn verschiedene Methoden als `virtual` markiert sind, damit Polymorphismus erreicht wird, wenn dieser benötigt wird.

    Füge weiters hinzu:

    - `NAndGate`: ein AND-Gatter, dessen Ausgang negiert ist (ein NAND-Gatter)
    - `NOrGate`: ein OR-Gatter, dessen Ausgang negiert ist (ein NOR-Gatter)

Da wir für diese beiden Gatter keine entsprechende Funktion in `propositional.Operators` haben, musst du auf einen fein konstruierten Lambda-Ausdruck zurückgreifen!

In der Schaltungstechnik werden diese oft verwendet, da allein aus `NAND`-Gattern jede beliebige andere boolsche Schaltung realisiert werden kann. Selbes gilt für die NOR-Gatter und voilà: Fertig ist die Geschichte. Ende gut, alles gut!

Naja, nicht ganz, denn testen ist schon noch angesagt! Weiter zum nächsten Punkt.

13. Teste jetzt das System, indem du ein Refactoring des Hauptprogramms vornimmst, dass der momentane Testcode in eine eigene Datei `circuits` und dort in einen (Unter)Namespace und in eigene Funktion kommt, z.B. `and3` und eine weitere Funktion schreibst, die die Schaltung aus dem Abschnitt Logiksimulator implementiert.

14. Zum glorreichen Schluss geht es jetzt noch daran ein Flip-Flop zu implementieren. Genau gesagt soll es sich um ein nicht taktgesteuertes RS-Flip-Flop handeln. Was ist das? Eigentlich nichts anderes als eine Speicherstelle, die in der Lage ist, ein Bit zu speichern. Siehe `https://de.wikipedia.org/wiki/Flipflop#Nicht_taktgesteuerte_Flipflops`.

    Hier wieder ein paar Tipps zum Abarbeiten (in der richtigen Reihenfolge):

    a) Implementiere diese Funktionalität auf Basis von NOR-Gattern wieder in einer eigenen Funktion `flip_flop`.

    b) Baue die Funktion `flip_flop` so auf, dass

        i. zuerst die Variablen und die Gatter angelegt werden

        ii. dann diese korrekt miteinander verbunden werden

        iii. dann das Logging für den Ausgang des zweiten NOR-Gatters aktiviert wird

        iv. dann die Variablen und Gatter zurückgesetzt werden

        v. Dann fünf Mal hintereinander

            A. der S-Eingang auf `true` gesetzt

            B. der S-Eingang auf `false` gesetzt

            C. der R-Eingang auf `true` gesetzt

            D. der R-Eingang auf `false` gesetzt

            wird.

Dann sollte es zu folgender Ausgabe kommen:

```
nor2.o: False (reset)
start flip-flopping
set: true
set: false
reset: true
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
```

Analysiere jetzt diese Ausgabe! Was ist daran eigentlich nicht in Ordnung? Denke nach (mind. 5 Minuten)!

- Gefunden: sehr gut und weiter mit dem nächsten Punkt.
- Nicht gefunden: …

c) Das Problem ist, dass am Anfang zu keiner Änderung des Ausgangs kommt:

```
set: true       <-
set: false
reset: true
reset: false
set: true
nor2.o: True (changed)
```

Woran liegt das? Denke nach (mind. 5 Minuten)!

- Gefunden: sehr gut und weiter mit dem nächsten Punkt.
- Nicht gefunden: …

d) Der Grund ist, dass wir die die NOR-Gatter zurückgesetzt haben und das Rücksetzen auf den Ausgang `false` stattfindet. Jetzt kann man argumentieren, dass es hierfür der Wert `true` besser wäre und das ist in gewissen Maße auch richtig. Überschreibe daher die `reset` Methode in den Klassen `NOrGate`, `NAndGate` und auch `NegGate` entsprechend, dass die Ausgänge am Anfang auf `true` gesetzt werden. Weiter mit den nächsten Punkt.

e) Sehr gut, wie wirkt sich diese Änderung auf die Ausgabe aus?

```
nor2.o: True (reset)
start flip-flopping
set: true
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
```

Wir sehen, dass es sehr wohl zu einem Unterschied gekommen ist: Der Speicherbaustein ist jetzt anfangs gesetzt! Das ist nicht unbedingt ein "natürliches" Verhalten. Eigentlich sollte anfangs zurückgesetzt sein!

D.h. nach dem Rücksetzen von `nor2` sollte der Wert manuell auf `false` gesetzt werden.

Damit kommt es zu folgender Ausgabe:

```
nor2.o: True (reset)
nor2.o: False (changed)
start flip-flopping
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
```

Damit stimmt alles, nur der Anfang ist unhübsch: Eigentlich sollte für das eine NOR-Gatter der Wert auf `true` zurückgesetzt werden und für das andere NOR-Gatter auf `false`. D.h. weiter zum nächsten Punkt.

f) Ändere daher die Methode `reset` von `Gate1` und `Gate2` so ab, dass diese ebenfalls einen optionalen Parameter erhalten und dieser den initialen Wert des Ausgangs darstellt.

Überschreibe dann die Methode `reset` in den Klassen `NAndGate`, `NOrGate` und auch `NegGate` entsprechend.

Danach passe dein Hauptprogramm an:

```
nor1.reset(true);
nor2.reset(false);
```

Dann sollte es zu folgender Ausgabe kommen:

```
nor2.o: False (reset)
start flip-flopping
set: true
set: false
reset: true
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
```

Dummerweise ist diese jetzt "wieder" falsch. Warum?

Denken!

g) Ok, das Problem ist, dass der Eingang `i0` von dem zweiten NOR-Gatter auf false ist (da zurückgesetzt) und es damit nicht gleich zu einem Wechsel kommt.

Setze daher, nach dem Rücksetzen des zweiten NOR-Gatters den besagten Eingang manuell auf `true`. Dann wird es zu folgender (richtiger) Ausgabe kommen:

```
nor2.o: False (reset)
start flip-flopping
set: true
nor2.o: True (changed)
set: false
reset: true
```

```
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
set: true
nor2.o: True (changed)
set: false
reset: true
nor2.o: False (changed)
reset: false
```

Es ist halt gar nicht so einfach ein Schaltwerk, das prinzipiell echt parallel abläuft in ein sequentielles Programm zu pressen.

Fix und fertig! Und hoffentlich stolz!

3.6 Klassendiagramm erstellen

Erstelle für dein gesamtes Projekt mittels UMLet ein UML Klassendiagramm! Es geht hier vorerst nur um die prinzipielle Struktur, aus der Sicht der Analyse. D.h. nur die relevanten Attribute und Methoden sind notwendig (sicher keine privaten).

4 Übungszweck dieses Beispiels:

- C# lernen!
  - Klassenbibliotheken erstellen und verwenden
  - Verständnis für (verschachtelte) Namensräume vertiefen
  - Klasse `System.Convert` kennenlernen

  - Generische Klasse `System.Func` kennenlernen
  - Lambda-Ausdrücke üben
  - Mit nullable Typen arbeiten
  - Arbeiten mit Properties üben und vertiefen
  - `readonly` für Attribute
  - `sealed` für Klassen
  - Delegates und Events kennen und einsetzen lernen
  - Aufzählungen üben
- Unicode-Zeichen verwenden und erkennen, dass sowohl Editor als auch Ausgabegerät (in unserem Fall die Konsole) die richtige Kodierung unterstützt.
- Verständnis für mehrwertige Logik vertiefen
- Observer-Pattern informell kennenlernen
- Logik-Schaltungen weiter üben
- UML Klassendiagramme erstellen