Practicum opdracht Design Patterns.

Naast theoretische basiskennis is het leren van design patterns voornamelijk een kwestie van doen. Daarom hoort er bij het vak Design Patterns een uitgebreide programmeeroefening in de vorm van een practicum.

Docent:

Bert Hoeks
Kamer OG018
Tel: 088 5256131
Email: ltm.hoeks@avans.nl

Opdrachtomschrijving:

In groepjes van maximaal 3 mensen maak je een simulator voor circuits bestaande uit (elektronische) logica-componenten. Dit programma moet voldoen aan de volgende eisen:

- De te simuleren logicacomponenten zijn resp. de AND-, de OR-, de NOT, de NAND, de NOR en de XOR poort. Behalve deze poorten wordt het circuit gevoed door zogenaamde *input's*. Deze worden gepresenteerd door bolletjes waaruit een logische '1' of een logische '0' komt. Om de uitkomsten van het logische circuits te verkrijgen worden zogenaamde *probes* gedefinieerd. Deze worden ook gerepresenteerd door bolletjes en geven het logische niveau op hun ingang weer. Voor alle logica componenten geldt dat ze ten minste 1 ingang hebben en altijd 1 uitgang. Voor *input's* geldt dat ze slechts 1 uitgang hebben. Voor de *probes* geldt dat ze slechts 1 ingang bezitten. De logische functies van alle logica componenten zijn gegeven in bijlage A. Omdat logische componenten niet ideaal zijn, heeft elk logisch component een zogenaamde propagation delay (t_p) attribuut. Dit is de tijd die het duurt voordat de uitgang van het logische component stabiel is bij *stabiele* ingangen. (typical t_p=10ns[1]). Deze propagation delay levert dus een beperking in schakelsnelheid op. Voor de rest mogen de logica-componenten als ideaal worden beschouwd.
- Het aantal te simuleren logica componenten is (in theorie) onbeperkt. Dit wil zeggen: er wordt tijdens de implementatie gebruik gemaakt van een dynamische datastructuur.
- In het te simuleren logische circuit bevinden zich geen terugkoppelingen. Je dient hier wel op te controleren.
- De simulator heeft als input een file die het te simuleren circuit beschrijft. Voor de lay-out van deze file zie bijlage B.
- De simulator levert een output die de simulatie resultaten beschrijft (dit is het logisch niveau gemeten door alle probes in het circuit) behorende bij een bepaalde input file. De lay-out van deze output mag naar eigen inzicht worden gemaakt.
- Het programma moet vanuit een object georiënteerde gedachte worden gemaakt. Dit betekent ook dat de uiteindelijke implementatie in een object georiënteerde programmeer omgeving gemaakt moet worden. Voorkeur gaat uit naar C++.
- Bovendien dienen minimaal drie design patterns geïmplementeerd te worden.

Behalve bovenstaande eisen zijn de volgende zaken *could-have*. Implementatie van (een of meerdere van) onderstaande zaken levert punten op.

- Grafische representatie van het te simuleren circuit inclusief de simulatieresultaten.
- Toevoeging van andere (zelf te definiëren) logische componenten.

[1] https://en.wikipedia.org/wiki/Propagation_delay

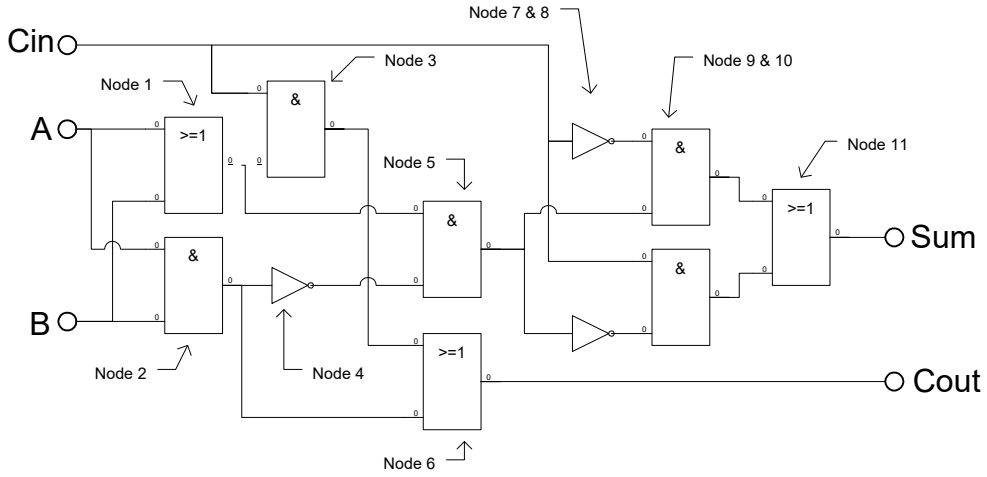
- Doorrekening van de totale propagation delay van de ingangen naar elke probe. (tip: dit kan bijvoorbeeld met het breath-first-seach (BFS) algoritme worden uitgevoerd.)
- …

Voorbeeld:

Een voorbeeld om de opdracht wat duidelijker te maken. In dit voorbeeld komen ook een paar implementatie tips naar voren.

De full-adder:

In onderstaande tekening is het te simuleren circuit gegeven. Het gaat hier om een zogenaamde fulladder. Dit is een schakeling die 2 getallen modulo 2 (binair) bij elkaar op kan tellen inclusief carry-bit. De full-adder is een basisschakeling van elke moderne microprocessor.



Figuur 1: Logica circuit van full-adder

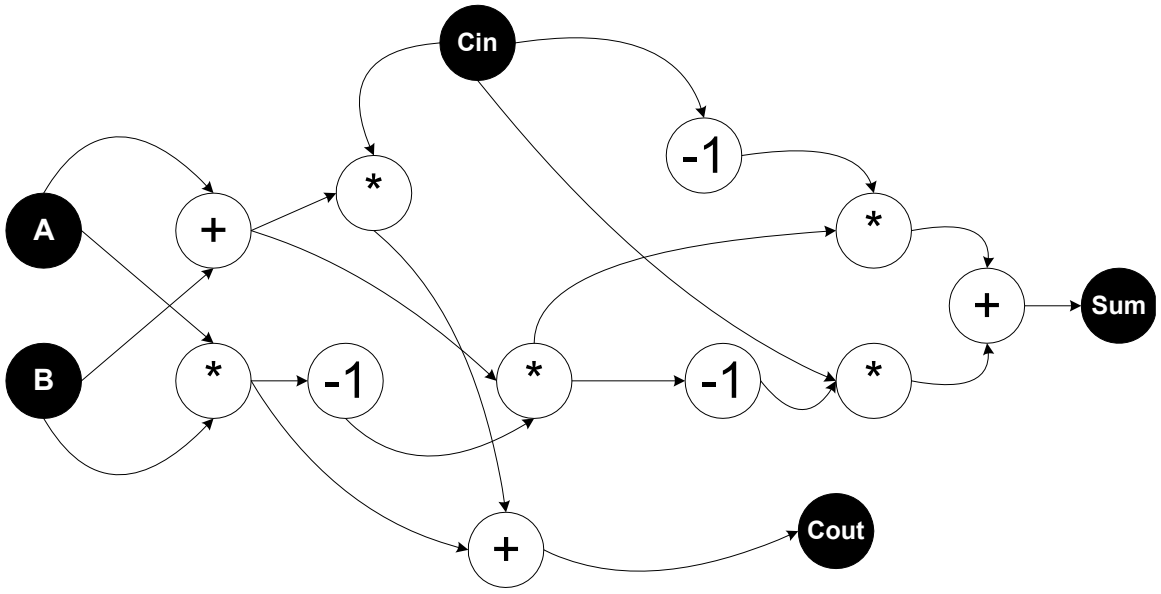
Het gegeven circuit wordt in onderstaande input file beschreven volgens de specificaties gegeven in bijlage B.

```
#
#
# Description of all the nodes
#
A: INPUT_HIGH; B:
INPUT_HIGH;
Cin:        INPUT_LOW;
Cout:       PROBE; S:
PROBE;
NODE1: OR;
NODE2: AND;
NODE3: AND;
NODE4: NOT;
NODE5: AND
NODE6: OR;
NODE7: NOT;
NODE8: NOT;
NODE9: AND; NODE10:
AND;
NODE11: OR;
```

```
#
#
# Description of all the edges
#
Cin:        NODE3,NODE7,NODE10;
A:          NODE1,NODE2;
B:          NODE1,NODE2;
NODE1: NODE3,NODE5; NODE2:
NODE4,NODE6;
NODE3: NODE6;
NODE4: NODE5;
NODE5: NODE8,NODE9;
NODE6: Cout;
NODE7: NODE9;
NODE8: NODE10;
NODE9: NODE11; NODE10:
NODE11;
NODE11: S;
```

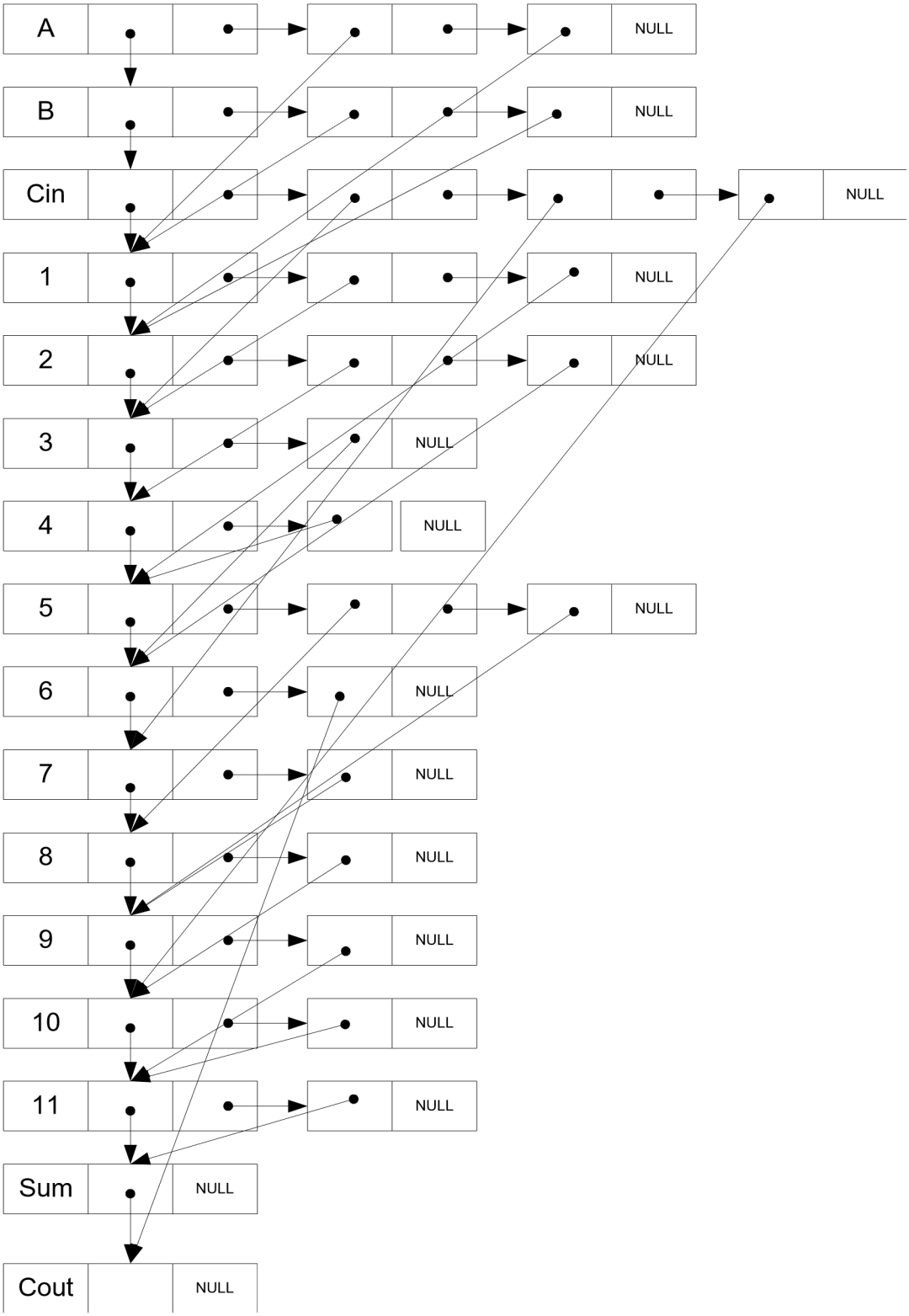
De schakeling kan nu geabstraheerd worden in een zogenaamde (gerichte) graaf. Zo'n graaf kan relatief makkelijk in een computer worden gestopt. Hierna kunnen allerlei algoritmen op de graaf worden losgelaten. Kortom: het real-world model is omgezet in een computervriendelijke representatie.

De gerichte graaf van de full-adder:



Figuur 2: Graaf representatie van full-adder

Een mogelijkheid om zo'n graaf op te slaan in bijvoorbeeld een lijststructuur of te wel een *adjacency list*. Dit is een enkelvoudig gelinkte lijst van alle knooppunten (nodes) in willekeurige volgorde. Elke knooppunt bestaat uit weer een gelinkte lijst die de takken (edges) van alle knooppunten met elkaar verbind. Van de full-adder ziet de adjacency list er als volgt uit.



Figuur 3: Adjacency list van de graaf beschrijvende de full-adder

In C++ kun je voor implementatie o.a. het type *vector* of *list* gebruiken. (Indien je in tijdsproblemen komt of je niet begrijpt hoe dynamische structuren werken kun je **na overleg met de docent** de keuze maken voor een implementatie met behulp van arrays.)
Opmerking: er zijn slimmere datastructuren die je kunt gebruiken.

Simulatieresultaat:

Als uitvoer van de simulator zullen de twee logische niveaus van de *probes* worden gegeven als functie van de *inputs*. In ons geval zal het simulatie algoritme de volgende uitvoer genereren:

Logica simulator versie 0.14 build 1.
Output for all probe nodes:

Probe 'S' = 0
Probe 'Cout' = 1
…bye

Werkwijze:

Tijdens de practica uren is de mogelijkheid om deze opdracht uit te werken. Hierbij ***kan*** de student de volgende planning aanhouden (niet verplicht):

Week nummer	Practicum
1	• Ontwerp objecten model / klassen inclusief samenhang. (is-een en heefteen relaties) in UML notatie. • Bepaal de rolverdelingen binnen de groep
2	Ontwerp de klasse(n) voor inlezen van input-file. Na goedkeuring volgt de implementatie. Denk hierbij aan de *stream* mogelijkheden van de diverse OO talen.
3	Implementatie voor inlezen van input-file in datastructuur
4	Ontwerp algoritme van circuit simulatie startende vanuit de datastructuur. Dus: 'hoe moet ik de graaf doorlopen om de waarden die bij de *probes* horen te kunnen berekenen?'. Probeer het probleem eerst in gewoon Nederlands op te lossen; probeer daarna jouw oplossing te formaliseren. Bespreek de oplossing met de docent.
5	Implementatie algoritme
6	
7	Bespreking van de opdracht in een assessment.

Tabel 1: Werkwijze en planning

Natuurlijk is het altijd mogelijk de docent vragen te stellen zodat onduidelijkheden geen geldige reden is voor opgelopen vertraging.

Documentatie:

Tijdens de uitwerking van de opdracht worden alle ontwerpen, beslissingen, correspondentie, overwegingen enzovoort vastgelegd in een documentatiemap. Deze wordt gebruikt als referentie voor zowel de docent als de groepsleden.

Beoordeling:

Beoordeling vindt plaats aan de hand van de volgende criteria:

- De modellering van de opdracht in een object model (UML notatie)
- Gebruikte datastructuur

- Oplossing van het simulatiealgoritme
- Gebruikte design patterns
- Uitgevoerde tests, in het bijzonder het resultaat van de testbestanden
- Refactoring
- Samenwerking in de groep

Indien de docent van mening is dat de het product aan minimale eisen voldoet. wordt de opdracht met een voldoende afgesloten. Indien de docent van mening is dat slechts een deel van de groep bijgedragen heeft aan het positieve resultaat, dan wordt slechts dat deel van de groep met een voldoende beloond.

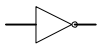
Bijlage A: Logica functies van diverse basis componenten.

Waarheidstabellen voor diverse logische componenten. Merk op dat alle mogelijke poorten kunnen worden gemaakt van de *not*- en de *and* poort. (oftewel: de moeder van alle logische poorten is de *NAND* poort.) Kunt u dit gebruiken in uw object model?

De NOT poort of 'Inverter'

De uitgang van een *NOT* poort is logisch '1' wanneer de ingang logisch '0' is.

Symbool:



Vergelijking:

$$F = \overline{A}$$

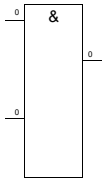
Waarheidstabel:

Input A	Output
0	1
1	0

De AND poort

De uitgang van een *AND* poort is logisch '1' wanneer beide ingangen logisch '1' zijn.

Symbool:



Vergelijking:

$$F = A \cdot B$$

Waarheidstabel:

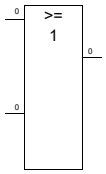
Input A	Input B	Output
0	0	0

0	1	0
1	0	0
1	1	1

De OR poort

De uitgang van een *OR* poort is logisch ‘1’ wanneer een of meer ingangen logisch ‘1’ zijn.

Symbool:



Vergelijking:

$$F = A + B$$

Waarheidstabel:

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

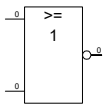
Stelling:

Met behulp van *AND* en *NOT* poorten kan ook een OR poort gemaakt worden. Met behulp de wetten van de propositielogica geldt: $F = A + B = \overline{\overline{A} + \overline{B}} = A \cdot B$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De NOR poort (Not OR)

De uitgang van een *NOR* poort is logisch ‘1’ wanneer beide ingangen logisch ‘0’ zijn.

Symbool:



Vergelijking:

$$F = \overline{A + B}$$

Waarheidstabel:

Input A	Input B	Output
0	0	1
0	1	0
1	0	0
1	1	0

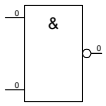
Stelling:

Met behulp van *AND* en *NOT* poorten kan ook een *NOR* poort gemaakt worden. Met behulp de wetten van de propositie logica geldt: $F = \overline{A + B} = \overline{A} \cdot \overline{B}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De NAND poort (Not AND)

De uitgang van een *NAND* poort is logisch ‘1’ wanneer niet beide ingangen logisch ‘1’ zijn.

Symbool:



Vergelijking:

$$F = \overline{A \cdot B}$$

Waarheidstabel:

Input A	Input B	Output
0	0	1
0	1	1
1	0	1
1	1	0

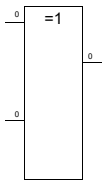
Stelling:

Met behulp van *AND* en *NOT* poorten kan ook een *NAND* poort gemaakt worden. Met behulp de wetten van de propositie logica geldt: $F = \overline{A \cdot B}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De XOR poort (eXclusieve OR)

De uitgang van een *XOR* poort is logisch ‘1’ wanneer beide ingangen verschillend zijn.

Symbool:



Vergelijking:

$F=A \oplus B$

Waarheidstabel:

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Stelling:

Met behulp van *AND* en *NOT* poorten kan ook een *XOR* poort gemaakt worden. Met behulp de wetten van de propositie logica (zie ook de waarheidstabel) geldt: $F = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B = \overline{\overline{A \cdot \overline{B}} \cdot \overline{\overline{A} \cdot B}}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

Bijlage B: Input-file specificatie.

De input specificatie files bestaat uit twee secties. Als eerste worden alle *nodes* van het circuit benoemd als tweede alle *edges*.
Elke regel in de specificatie file bestaat uit slechts één aparte node of edge beschrijving. Alles achter een '#' teken tot aan het einde van de regel wordt gezien als commentaar. Spaties en/of andere whitespace karakters zijn niet relevant. Regels worden afgesloten met een line feed en vervolgens een carriage return LF ev.CR zoals gebruikelijk bij ASCII-bestanden.

Node-beschrijving:

<node_identifier>:<node_descriptor >;'

- Hierin is *node_identifier* een string bestaande uit ten minste 1 karakter uit de verzameling {az,A-Z,_}.
- Hierin is *node_descriptor* een element uit de verzameling { INPUT_HIGH, INPUT_LOW, PROBE, OR, AND, NOT, NAND, NOR, XOR}. Deze geeft dus aan welke logische bewerking op de node zal plaatsvinden.

Voorbeelden:

```
#
# Commentaar
#
NODE1: OR;          # De node met identifier 'NODE1' heeft een OR functie.
OUT_B: PROBE; # De node met identifier 'OUT_B' heeft een monitor functie.
```

Edge beschrijving:

<node_source_identifier>:<node_target_identifier, … ,node_target_identifier>;

- *Node_source_identifier* beschrijft de bron vanuit waar een tak begint. De *node_source_identifier* moet in de node sectie gedefinieerd zijn.
- *Node_target_identifier* beschrijft de nodes waar de bron tak naar wijst. De *node_target_identifier* moet in de node sectie gedefinieerd zijn.

Voorbeeld:

```
# De uitgang van node met identifier 'Cin' is verbonden met de nodes NODE3, NODE7 en NODE10.
Cin:        NODE3,NODE7,NODE10;
# 'Node 6' is verbonden met node 'Cout' NODE6:
Cout;
```