Practicumopdracht
Design Patterns

Achtergrond

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.

Opdrachtbeschrijving (deel 1)

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 gerepresenteerd door bolletjes waaruit een logische '1' of een logische '0' komt.

Opdrachtbeschrijving (deel 2)

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.

Opdrachtbeschrijving (deel 3)

- Omdat logische componenten niet ideaal zijn, heeft elk logisch component een zogenaamde propagation delay (t_p) attribuut. Dit is de tijd die het duur voordat de uitgang van het logische component stabiel is bij *stabiele* ingangen. (typical t_p=10ns). Deze propagation delay levert dus een beperking in schakelsnelheid op. Voor de rest mogen de logica-componenten als ideaal worden beschouwd.

Opdrachtbeschrijving (deel 4)

- 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 idealerwijs geen terugkoppelingen.
- De simulator heeft als input een file die het te simuleren circuit beschrijft.
- Voor de lay-out van deze file zie bijlage B van de practicumopdracht-beschrijving.

Opdrachtbeschrijving (deel 5)

- 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 objectgeoriënteerde gedachte worden gemaakt. Dit betekent ook dat de uiteindelijke implementatie in een objectgeoriënteerde programmeeromgeving gemaakt moet worden. De voorkeur gaat uit naar C++.

Opdrachtbeschrijving (deel 6)

- Er dienen minimaal drie design patterns geïmplementeerd te worden.
- Bovendien worden tijdens de lessen extra eisen besproken:
  - *Criteria van Meyer*
  - *Geen casts*
  - *Geen gebruik van ID's (...)*
  - *Low binding factory method (volgende week)*
  - *…*

Opdrachtbeschrijving (deel 7)

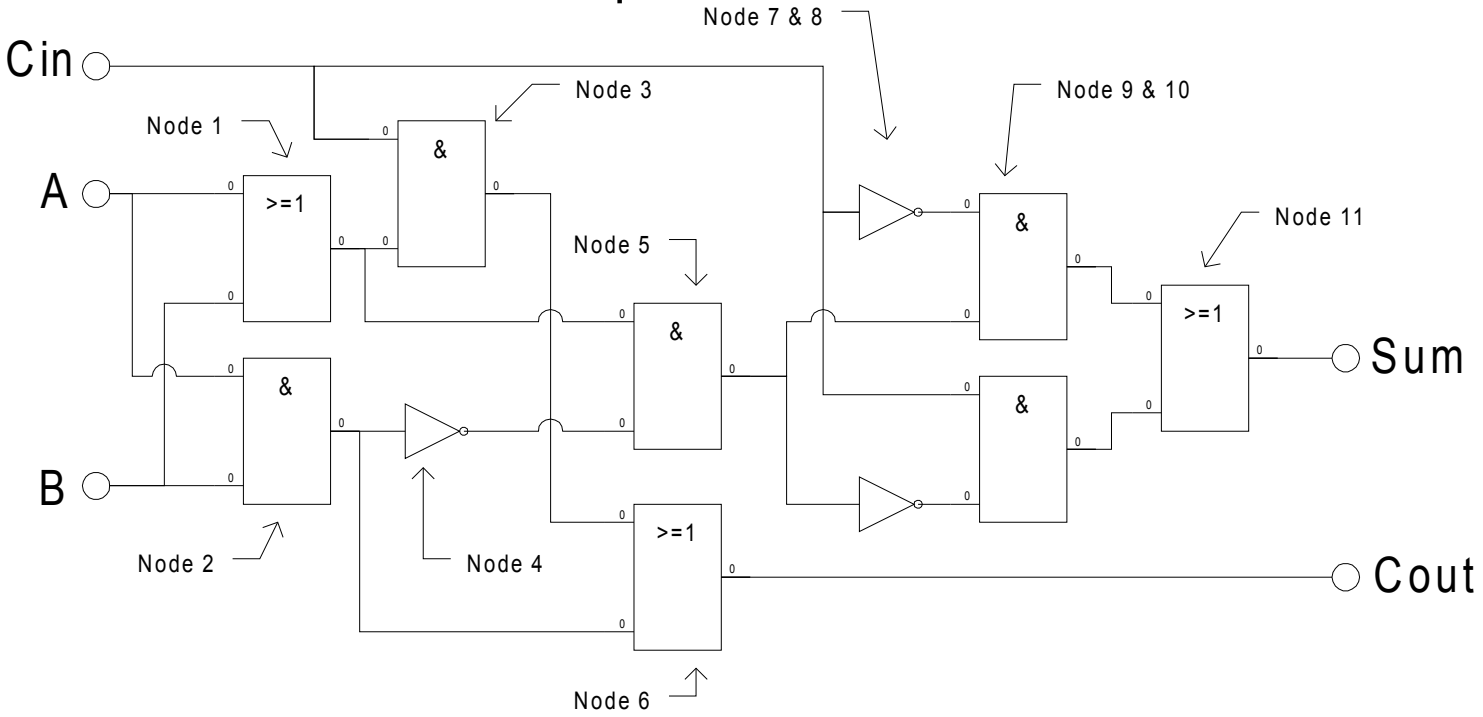
Behalve bovenstaande eisen zijn de volgende zaken *could-have*. Implementatie van (een of meerdere van) onderstaande zaken levert punten op.

- Grafische representatie van het te simuleren circuit inclusief de simulatieresultaten.
- Toevoeging van andere (zelf te definiëren) logische componenten.
- Doorrekening van de totale propagation delay van de ingangen naar de probes.
- Doorrekenen van netwerk met het “topological sort” algoritme.
- …

Voorbeeld: de full-adder

- De full-adder:

  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.



Beschrijving full adder (deel 1)

```
#
#
# 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;
```

Beschrijving full adder (deel 2)

```
#
#
# 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;
```

Gerichte graaf

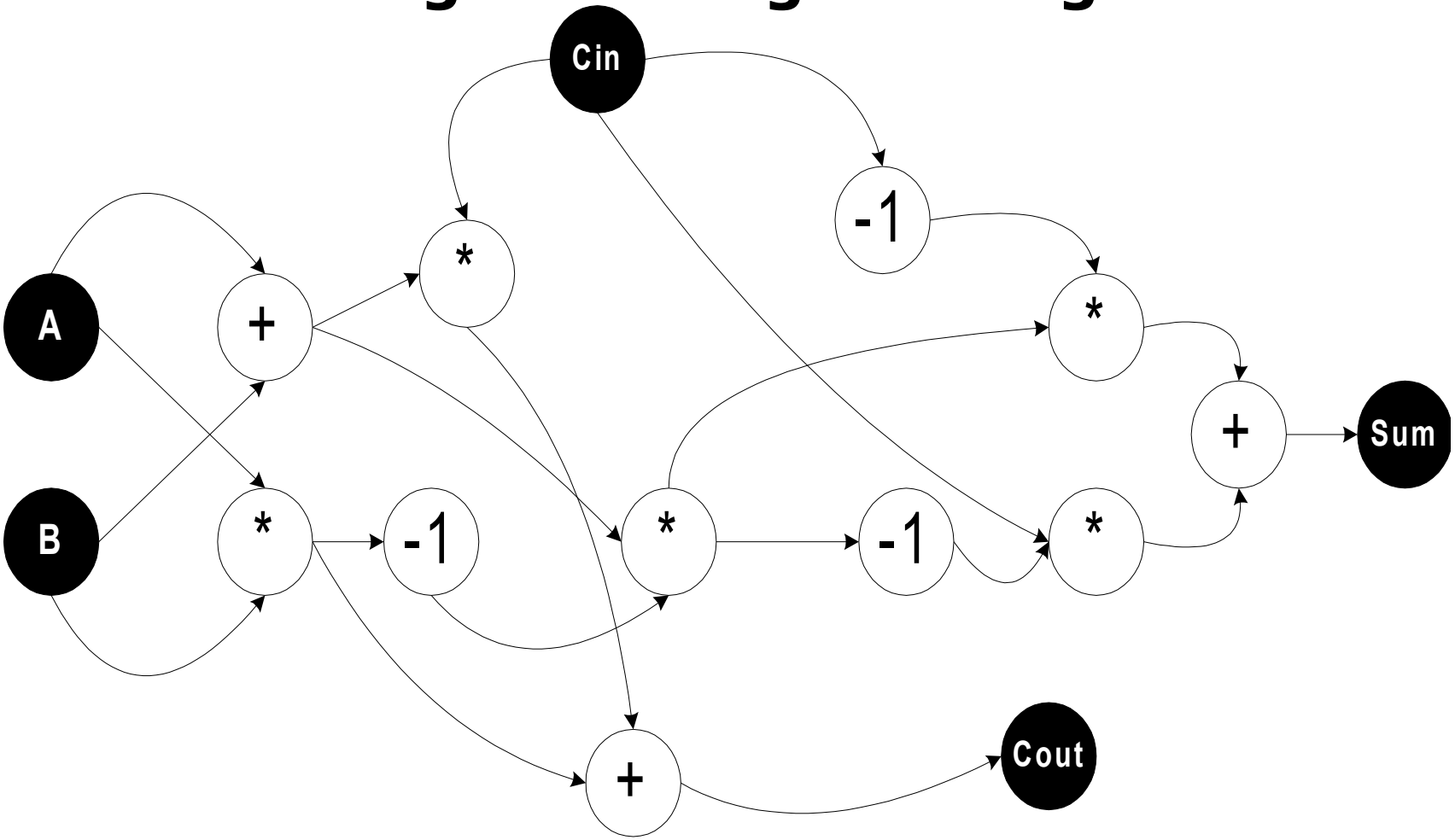
De schakeling kan nu geabstraheerd worden in een zogenaamde (gerichte) graaf. Zo'n graaf kan relatief makkelijk in een computer worden gestopt. Hierna kunnen allerlei algoritmen op de graaf worden losgelaten. Kortom: het real-world model is omgezet in een computervriendelijke representatie.

Er zijn betere datastructuren als oplossing

Beter:

- *qua prestatie*
- *qua begrijpelijkheid*
- *qua implementatiegemak*

Tekening van een gerichte graaf



Van graaf naar Datastructuur

Zo’n graaf kun je opslaan in bijvoorbeeld een combinatie van een map met een gekoppelde lijst. Dit is een map die als *values* een gekoppelde lijst bevat. Elke knooppunt bestaat uit weer een gelinkte lijst die de takken (edges) van alle knooppunten met elkaar verbindt.

Uitvoer van de simulator

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
```

Mogelijke planning

Week	Werkzaamheden
1	Ontwerp objecten model / klassen inclusief samenhang. (is-een en heeft-een 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 Refactoring tot een goed object georiënteerd product.
6	Assessment: bespreking van de opdracht.

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 / punt

Beoordeling vindt plaats aan de hand van de volgende criteria:

- De modellering van de opdracht in een object model (UML notatie)
- Oplossing van het simulatiealgoritme
- Designopties
- 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.