Digitális áramkör szimulátor

NHF Dokumentáció

Programozás alapjai 2.

Pálinkás Lőrinc Mihály - XB0SMF

Tartalom

1. Feladat

Digitális áramkör

Készítsen egyszerű objektummodellt digitális áramkör szimulálására! A modell minimálisan tartalmazza a következő elemeket:

- NOR kapu
- vezérelhető forrás
- összekötő vezeték
- csomópont

A modell felhasználásával szimulálja egy olyan 5 bemenetű kombinációs hálózat működését, amely akkor ad a kimenetén hamis értéket, ha bementén előálló kombináció 5!

Demonstrálja a működést külön modulként fordított tesztprogrammal! A megoldáshoz ne használjon STL tárolót!

2. Feladatspecifikáció

Feladat általános leírása

A program lehetőséget ad digitális áramkörök szimulálására. A felhasználó áramköröket képes betölteni szöveges file-okból, beállítani a bemeneti jelkombinációt és a kapcsolók állapotát és ez alapján kiolvasni a kimeneti jeleket.

Megvalósított áramköri elemek

A következő elemeket képes szimulálni az áramkör:

- Forrás: állítható LOW és HIGH kimeneti jelekkel, kiolvasható az értéke
- Vezeték: két részt köt össze az áramkörben
- Csomópont: 1 bemeneti jelet több kimeneti irányba tud továbbítani
- Kapu: Több bemenetből képes pontosan 1 kimenetet produkálni. Megvalósított kapuk:
  - AND, OR, NOT
  - NAND, NOR
  - XOR, XNOR
- Lámpa: tárolja a kapott jelet, kiolvasható az értéke
- Kapcsoló: továbbítja a jelet, amennyiben zárt, egyébként LOW jelszintet ad ki

A bonyolultabb elemeket (pl. funkcionális elemek) egyelőre nem implementáljuk, mert könnyen felépíthető ezekből szimuláció során, de ha marad idő, akkor ezeket is megvalósíthatjuk.

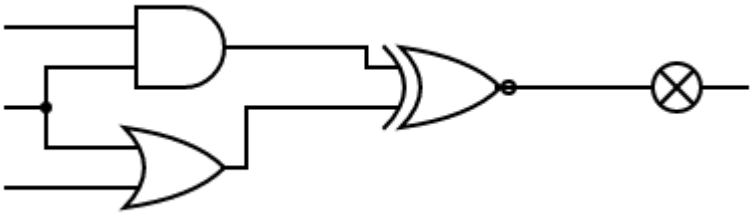
Bemenet formátuma

Az áramkörök felkonfigurálása szöveges file alapján történik. Ebben a felhasználó felsorolja a komponenseket, megadva, hogy hogyan kapcsolódnak. A kapcsolódás megadásához meg kell adni, hogy az adott lába az elemnek melyik csompópontra kapcsolódik. A csomópontokat számok jelölik megadáskor, azonos szám azonos csomópontot jelent. Tehát a konfigurációs file körülbelül így néz ki:

test.txt

```
SOURCE: (1) (2) (3)
AND: (1,2,4)[(…,…,…) …] <- ha több van
OR: (2,3,5) …
XNOR: (4,5,6) …
LAMP: (6)
```

Például erről az ábráról azt tudjuk leolvasni, hogy 3db forrás van jelen, ezek az 1-es, 2-es és 3-as csomópontokra küldik a jeleiket. Emellett van az 1 és 2-es csomópontra kapcsolódó ÉS kapu, mely a 4-es csomópontra küldi a jelét. Hasonlóan kell értelmezni a többit. Ez alapján az alábbi digitális áramkör szimulálható:



Fontos megjegyzés: A szimuláció során az összekötő vezetékeket is csomópontnak tekintünk, így tudjuk könnyen megadni formátumosan a kapcsolódásokat.

A példa azt is mutatja hogy milyen egy egyszerű kapu megadásának például általános formátuma:

GATE_NAME: (IN1, IN2, OUT1) …

Kimenet opciók

A felhasználó képes lekérdezni több információt az áramkörből:

- A lámpák státusza: minden lámpának ki tudjuk olvasni az állapotát, hogy világít-e vagy nem.
- A források státusza: minden forrásnak meg tudjuk adni és ki tudjuk olvasni a jelszintjét.
- A kapcsolók státusza: minden kapcsolónak meg tudjuk adni és ki tudjuk olvasni, hogy zárva van-e vagy sem.

Az áramkör kimenetének megadható, hogy melyik file-ba irányítjuk át a szimuláció kimenetét.

3. Pontosított specifikáció (kiegészítés)

Áramköri elemek I/O pin száma

Az alábbi táblázat mutatja az egyes elemekhez tartozó ki- és bemeneti pin-ek számát, amivel létre lehet hozni:

Áramköri elem típus	Bemeneti pin-ek száma	Kimeneti pin-ek száma
Forrás	0 (csak jelet ad ki)	1 (egy jelet ad ki)
Csomópont	1 (ahonnan kapja a jelet)	≥1 (tetszőlegesen sok helyre küldhet jelet)
Vezeték	1 (csomópont speciális esete)	1 (csomópont speciális esete, amikor 1 kimenet van)
Kapu	≥1 (minden kapu legalább egy bemenetből állít elő kimenetet, támogatva lesz több mint 2 bemenetű AND, OR, stb.)	1 (minden kapu 1 logika jelet állít elő)
Lámpa	1 (1 helyről fogad jelet)	0 (nem ad ki jelet, csak eredmény tárolásra van)
Kapcsoló	1 (1 helyről fogad jelet)	1 (1 helyre továbbít jelet)

Felhasználói felület

A felhasználó számára van biztosítva egy egyszerű menü, melyben a következő műveleteket tudja elvégezni:

- Áramkör betöltése: képes megadni egy file nevét, és innen betölteni egy áramkört
- Bemeneti adatok beállítása: meg tudja adni a források bemeneti jeleit illetve a kapcsolók állását

- Kimeneti file beállítása: meg tudja adni hogy melyik file-ba irányítsa át a kimenetet, alapvetően a std::cout-ra küldi a szimuláció kimenetét
- Szimuláció: végrehajtja a szimuláció lefuttatását
- Kilépés: leállítja a programot

4. Terv

Információ áramlása

A digitális áramkör szimulálása során az információ áramlását fogjuk modellezni. Mielőtt a tervezett objektummodell be lesz mutatva, azelőtt elengedhetetlennek tűnt, hogy előbb az információ áramlásának modelljét jellemezzem, mert ennek jelentős kihatásai lesznek az egyes osztályok tervezésére, meghatározó hogy hogyan is kommunikálnak egymással az objektumok.

A fő ötlet és inspiráció a tervezéskor a Számítógépes architektúrák tárgy keretében megismert adatáramlásos modell volt. Ha egy áramkör szimulálását vesszük figyelembe, akkor jön a gondolat, hogyan is tudjuk, hogy honnan kell kezdeni a jelek kiértékelését?

Kezdetben csak a források jele adott, a többi áramköri elemnek nem tudhatjuk, mert korábbi elemek jelére is építhetnek. Emiatt először ezek jeleit ismerve tudjuk elindítani az információ áramlását, hiszen azon elemek, amelyeknek minden lába forrásra kapcsolódik, rögtön kiértékelhetőek, majd ezek után az ezekre kapcsolt elemek, és így tovább.

Ez a viselkedés nagyon szoros párhuzamot mutatott az adatáramlásos adatfeldolgozási modellel, így erre alapozva fejlesztettem ki az adatok feldolgozásának menetét. Az áramköri kapuk, kapcsolók, stb. a precedenciagráfnak az egyes csúcsai, melyek kiértékelnek a bemeneti jel alapján egy kimeneti jelet, amit aztán tovább küldenek a következő csúcsoknak, jelen esetben egy másik áramköri elemnek.

A működése nagy vonalakban a következő: a szimuláció során mindig számon tartunk egy „aktív” FIFO-t. Ebben a FIFO-ban mindig azon elemeket tartjuk, amelyeknek minden jele meg van, tehát kiértékelhetőek. Amikor egy ilyen elemet kiértékelünk, akkor minden kapcsolódó áramköri elemnek jelezzük, hogy eggyel nőtt a „kész” bemenetek száma. Ha ez eléri a bemenetek számát, akkor meg van minden szükséges bemenete, tehát be tudjuk rakni az aktív FIFO-ba, ahol aztán ki lesz értékelve.

Így sorjában minden áramköri elemre kiértékeli és beállítja a megfelelő jelértéket, amíg van ilyen elem.

Adatáramlásos működés problémái és megoldások

Tervezés során előjött több probléma is, ami elkerülhetetlen az adatáramlásos modellből fakadóan, bár ezeknek egy része főleg az áramkör megadásának kiszámíthatatlanságából adódik. A következőekben ezekere adok megoldásokat.

1. Elszigetelet, kiértékeletlen elemek:

Tegyük fel hogy az alábbi file-t kapjuk felkonfiguráláskor:

```
SOURCE: (1)

LAMP: (2)
```

Ezen az egyszerű látni hogy mi a baj: az 1-es csomópontra kapcsolódó forrásból sosem fog eljutni a 2-es csomópontra kapcsolódó lámpába a jel. Ez azt is jelenti, hogy a kimeneti értéke nem lesz értelmes, a lámpa nem mér valós értéket.

Ez alapvetően nem is probléma, mert (mint később látjuk) minden pin alapvetően LOW jelet kap, ami egyezik azzal, ami a valóságban lenne, hogy nincs rákötve tápra = LOW jel. Ez azonban akkor baj, ha mondjuk 2 LOW jelből mondjuk egy NAND HIGH jelet kell képezzen, de ezt nem teszi meg, mert sose lesz kiértékelve.

Ha valóságban elképzeljük, akkor ez gyakorlatilag egy „levegőben lebegő”, áramkörtől független lábat jelent valamilyen áramköri elemre nézve.

Megoldás: felkonfiguráláskor futtatunk egy próba szimulációt, mely során figyeljük, hogy le lett-e szimulálva minden elem. Amennyiben ez teljesül, akkor minden rendben, az áramkör biztosan helyesen kiértékel minden elemet. Amennyiben van olyan elem, ami nem lesz kiértékelve, az jelzi, hogy ez a baj van valahol, ezt jelezzük a felhasználó felé, és az áramkör el lesz utasítva.

2. Visszacsatolás

Másik szembetűnő problémát az adatáramlásos modellel az alábbi kapcsolások szemlélteti:

1: Önhivatkozásos visszacsatolás:

```
SOURCE: (1)

AND: (1, 2, 2)

LAMP: (2)
```

A fenti áramkör szemlélteti ezt a problémát, mert itt egy kapu bemenete függ a kimenetétől, emiatt hiába is van minden rendesen összekötve, nem fog kiértékelődni.

Megoldás: Ez a probléma elsőre nehezen kezelhetőnek tűnhet, de a megoldás már létezik. A fő probléma, hogy nem kiértékelhető, hiszen saját magára alapszik. Azonban ezt az előző rész tesztje ezt is el fogja ugyanúgy kapni, hiszen sosem lesz kiértékelve a 2-es csomópontra kapcsolódó lába az elemnek.

2: Stabil visszacsatolás

```
SOURCE: (1)
NOT: (1, 2) (2, 3)
LAMP: (3)
```

A jelenlegi példában gyakorlatilag egy D flip-flopot valósítunk meg. Jogosan merül fel a kérdés, hogy mégis hogyan lenne lehetséges itt kezelni a visszacsatolást. Jelenlegihez hasonló esetekben nincsen baj a visszacsatolásból, mert ugyanazt a stabil jelet küldi vissza, mint amit kapott.

Megoldás: Amennyiben a visszacsatolás azonos jelt küld vissza, akkor nincs probléma, az elem nem lesz újra kiértékelve, hiszen nem változtat semmilyen szempontból a kapcsoláson.

3: Instabil visszacsatolás

```
SOURCE: (1)
NOT: (1, 2) (2, 3) (3, 1)
LAMP: (3)
```

Hogyan értékeljünk ki egy ilyen áramkört? Több kérdés is felmerülhet, hiszen, ebben a visszacsatolás ellentétes jelet küldi vissza, mint amit kapott. Bár elsőre ez a helyzet problémásnak tűnhet, hiszen sok szélső eset is lehetséges, de a feloldásához egy észrevétel kell.

Vegyünk egy tetszőleges ilyen áramkört. Ekkor fel tudjuk osztani stabil és instabil kapcsolású részlegekre. Forrás mindig stabil lesz, mert nincs kimenete. Ez viszont vagy direkt kapcsolódik az instabil részre, vagy stabil kapcsolású részeken keresztül, amig hasonlóan funckionálnak jelen esetben a forráshoz (stabil a kimenet, nincs instabil visszacsatolás). Ez azonban azt jelenti, hogy instabil visszacsatolás kezdetekor egy stabil jelforrást kapó csomópontra küldünk vissza ellentétes jelet (mert instabil).

Valóságban elképzelve ez hasonlóan funckionál mint ha a földet összekötjük a táppal, rövidzárat alkotva. Ezek alapján a következő megoldásra jutottam.

Megoldás: Ha egy visszacsatolás ellentétes jelet küld vissza, akkor biztosan keletkezne rövidzár az áramkörben, tehát amennyiben ez az eset következik be, akkor jelezzük a felhasználó fele, hogy rövidzár történt (megadva a csomópontot), és szimuláció nem ad információt a kimenetről, mert értelmetlen lenne.

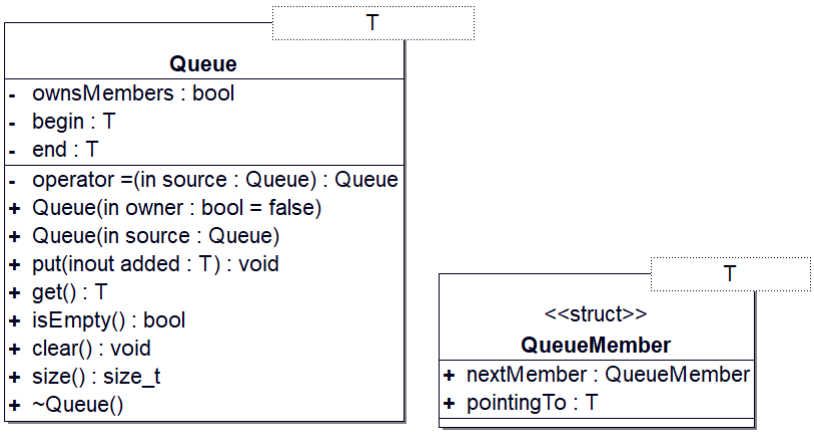
Objektummodell

Az áramköri elemek modellezése során adódott hogy az egyes áramköri elemek egymással kommunikáló objektumokként viselkednek. Ezek alapján dolgoztam ki a modellt, szemléletesség érdekében ez most bottum-up módon szeretném bemutatni, kezdve legalulról, lépésekben felépítve az elemek modelljét.

Queue osztály

Mielőtt a konkrét áramkör elemek modellezéséről beszélnék, fontosnak tartom, hogy előbb egy általánosabb szerkezetet mutassak be, amelyet több osztály is fel fog használni.

Ahogyan az információáramlás modelljének leírásában mondtam, egy aktív FIFO-ban kell tárolnunk a kiértékelendő elemeket. Ebből egyértelmű lett, hogy egy láncolt listában érdemes ezeket tárolnunk. Azonban a tervezés során kiderült, hogy nem csak erre az egy funkcióra kell egy láncolt list szerkezet, emiatt úgy döntöttem hogy egy generikus Queue osztályt hozok létre, melyet több helyen is újra fogunk használni:



Eltérés azonban van egy sima láncolt listához képest. Először is a Queue osztály alapvetően pointereket tárol objektumokra, ezek mindig dinamikusan foglaltak lesznek. Emiatt be kellett vezetnem egy „tulajdonos” attribútomot. Ennek a fő oka, hogy nem egyszer fogunk

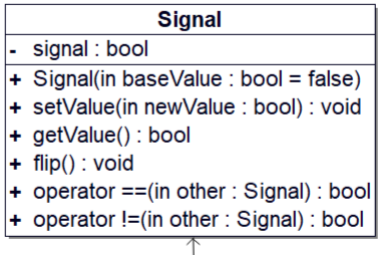
több Queue-ból ugyanarra az objektumra hivatkozni (más pointereken keresztül, máshogyan kezelve), és mivel ennek a Queue-nak felelőssége lehet törölni a memóriát, ezért létrehozáskor tudnia kell, hogy felelős-e az elemeiért.

Ezen túl a legtöbb művelet a szokásos, mint egy láncolt listában, a get() kiveszi az elejéről az elemet, a put() berak egyet a végére. Folyamatosan számon tartja a méretét, illetve lekérdezhető hogy üres-e.

Fontos!: A másoló konstruktor egy olyan queue-t hoz létre, ami ugyanazon elemeket tartalmazz, viszont NEM SZABADÍTJA fel őket, automatán, nem tulajdonos! (Ennek oka, hogy többször is kell kiszedni és valami végezni ezeken az objektumokon)
Assign operátor külső használat elkerülésére le van tiltva, ezért privát.

Signal osztály

A digitális áramkörökben jelszinteket mérünk le, emiatt döntöttem, hogy érdemes lenne egy saját osztályként működjön maga a logikai jel, ennek az eredménye lett a Signal, a digitális jelet modellező osztály:



Funkcionalitás szempontjából elég egyszerű osztály, létre tudunk hozni vele jelet, beállítani és kiolvasni, megfordítani és összehasonlítani. A jeleket boolean értékként tároljuk, mert azonos viselkedésű a digitális jelértékekkel. (true ~ HIGH (1), false ~ LOW (0))

Pin osztályok

A tervezés során következő felmerülő osztály a Pin volt, ezen keresztül tudnak kommunikálni az áramköri elemek. Két fő funkciót látnak el: egyrész jelet tárolnak, melyet ki lehet olvani, másrészt jelet adnak át a másik pin-nek.

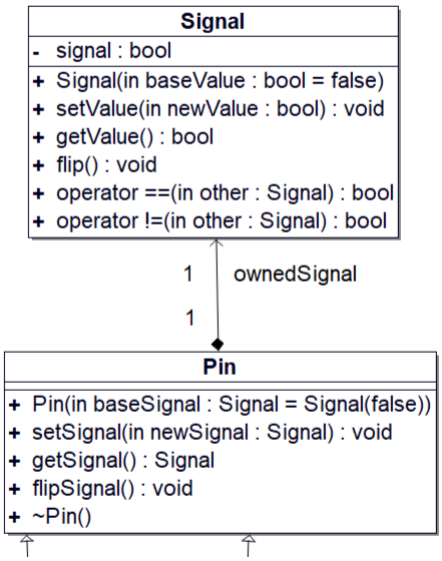
Tervezés elején az tűnt célszerűnek, hogy egyfajta Pin osztály létezzen, viszont hamar egyértelmű lett, hogy nem elég, az információ áramlásának modellje miatt szükséges volt két részre bontani.

Ennek fő oka, mint ahogy a specifikációban is látszik, az hogy alapvetően két szerepet tölthet be egy láb: információt fogad, azaz

bemenetként viselkedik, illetve információt továbbít, azaz kimenetként viselkedik. Ezek funkcionalitásokhoz szükséges műveletek teljesen másak, más információt szükséges tárolni.

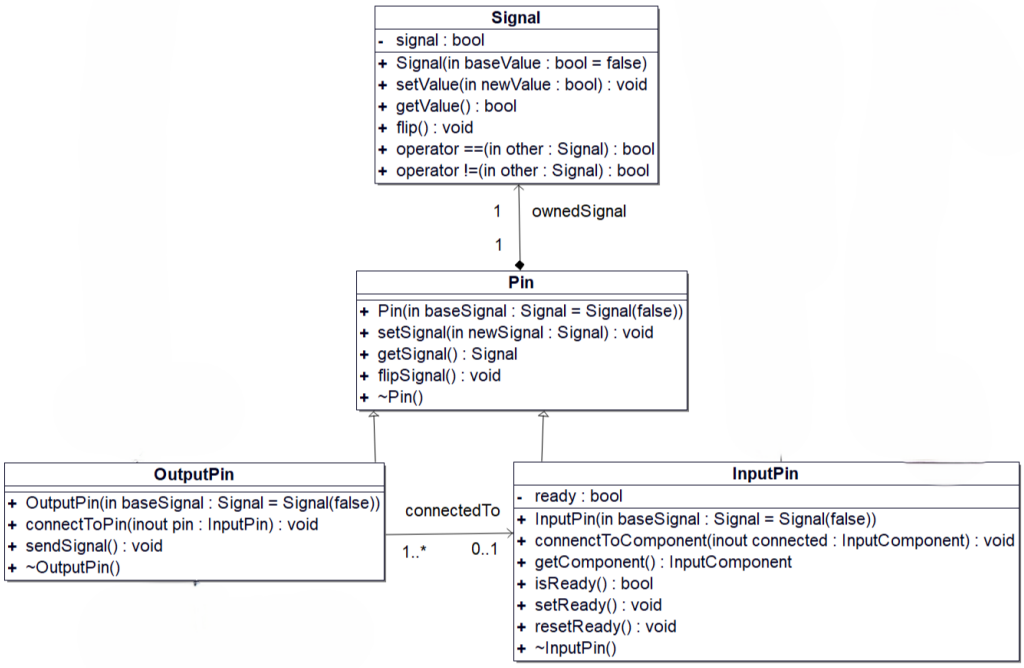
Emiatt döntöttem úgy, hogy bár fog létezni egy közös ősosztály, melyben a közös funckionalitás van megvalósítva, azonban két külön osztályként célszerűbb megvalósítani őket.

A sima Pin ősosztály tartalmaz minden azonos viselkedését egy elem lábának:



Van egy jele, amit tárol, alapértelmezetten ez LOW. Ezt lehet állítani és olvasni, illetve megfordítani (öröklés miatt van csak destruktor, memóriát nem kell felszabadítania, ez a többi osztályban is hasonló okok miatt, hogy a további öröklés esetén ne legyen baj vele).

Az InputPin és OutputPin osztályok végzik el a konkrét kimeneti és bemeneti szerep megvalósítását:



Az OutputPin osztály fő bővítése, hogy képes kapcsolódni InputPin-hez és neki jelet küldeni, egyébként ugyanolyan mint a sima Pin.

Az InputPin ezzel szemben nem másik Pin-hez, hanem egy áramköri elemhez kapcsolódik (ld. később), ennek jelzi, hogy kapott jelet egy OutputPin-től, ami ennek hatására ellenőrzi, hogy készen áll-e.

Minden InputPin tárolja a ready változóban, hogy készen áll-e információfeldolgozásra, ezt lehet beállítani és resetelni, illetve az állapotát lekérdezni (az elem részéről fogjuk). A resetelés lehetősége újraszimuláláskor lesz fontos, hiszen ekkor minden áramköri elem bemeneti lábainak készenlétét resetelni kell, hogy ne tévesen kerüljön be az aktív FIFO-ba (illetve a visszacsatolást is ezzel lehet ellenőrizhetni).

Component osztályok

Az áramköri elemek tervezés során a legcélszerűbb egy közös ősosztály volt, melyen keresztül egy heterogén kollekcióban tudjuk majd tárolni őket. A másik fő ok, hogy minden elemre ugyanazokat az általános műveleteket végezzük el: hozzáadjuk az aktív FIFO-hoz, ha készen áll kiértékelésre, illetve kiértékeljük és elvégezzük a jelkiküldést, amennyiben van kimenete.

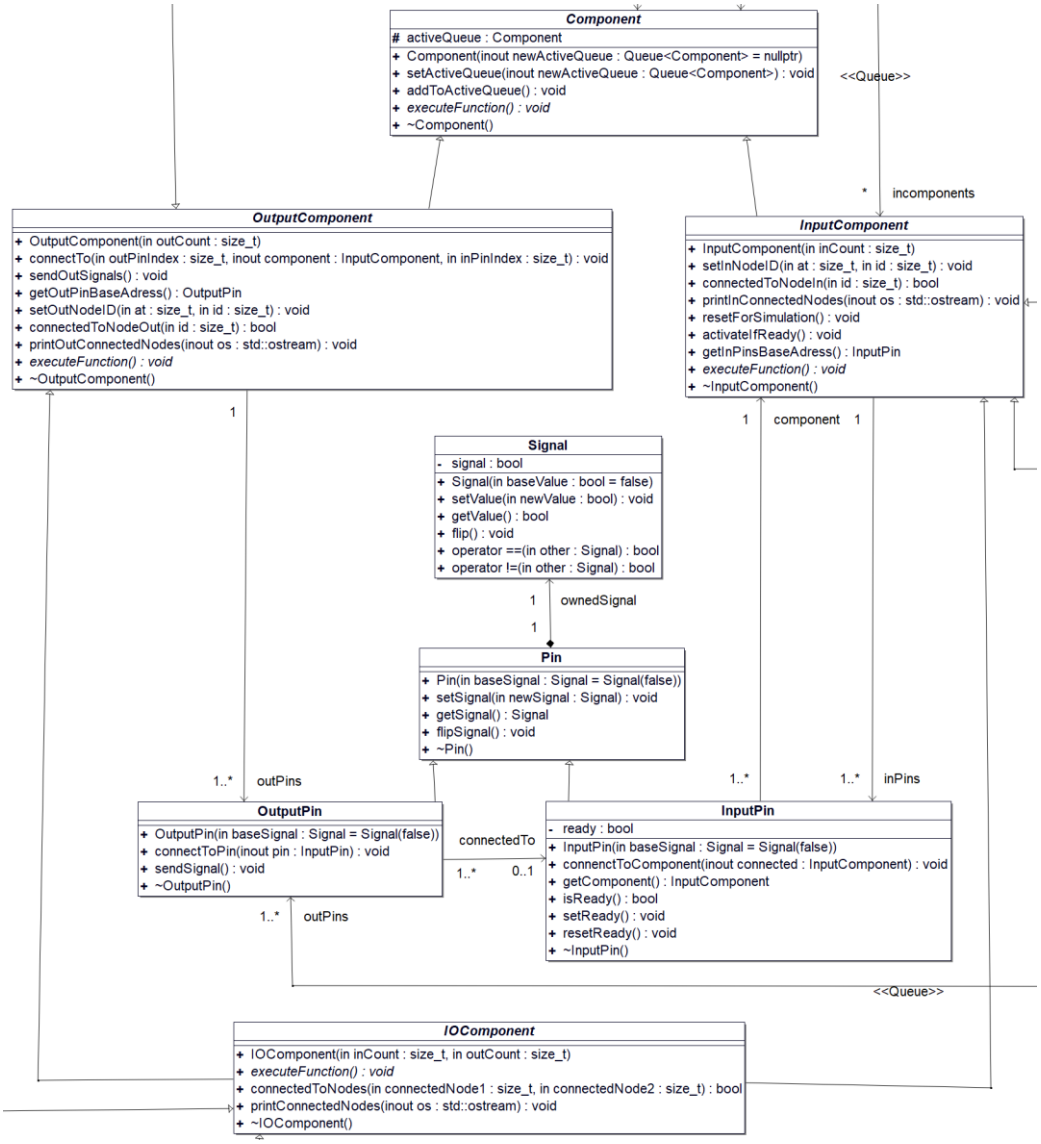
Azonban viszonylag hamar szembetűnt, hogy ez az egyetlen közös osztály nem lesz elég. Ahogyan a Pin-ek leírásában is láthattuk, szükség volt két külön célú pin osztályra, attól függően hogy milyen szerepet

töltenek be. Elsőre célszerű lehet, hogy minden elemnek legyen ki és bemenete, és vegyük 0-nak, ha nincs ilyen. Csak ezzel a problémába ütközünk, hogy sok elemre értelmetlen műveletek lesznek értelmezve, jobbnak tűnt, ha minden elemre választhatjuk hogy melyik funkciót tölti be, ha esetleg mindkettőt, akkor megkapja mindkét funkcionalitást.

Ezáltal hoztam létre az InputComponent és OutputComponent interfész osztályokat, melyeken keresztül egy áramköri elem megkaphatja vagy egyik vagy másik funckiót, esetleg mindkettőt, de akár ha valamelyiket bizonyos okok miatt máshogy kell implementálnunk (ld. később Node osztály) akkor nem okoz gondot, egyszerűen máshonnan kapja az egyik interfészt.

Gyakran azonban mindkettő interfészt közösen használjuk, emiatt döntöttem amellett, hogy egy közbenső teljesen absztrakt, mindösszeg kódírást megkönnyítő harmadik, IOComponent osztály is létrehoztam.

Ezen döntések eredményeként kaptuk meg az alábbi osztályokat:



Az ábrán olvashatunk le néhány függvényt, ami csak a konfiguráláskor és kiíráskot használatos, a többi, alapvetőbb fontosságú funckióra érdemes most koncentrálnunk (ezek változhatnak még fejlesztés alatt).

A Component osztály gyakorlatilag semmiben nem mutat újat a fent leírtakhoz képest, egyedül annyiban csak, hogy az aktív FIFO címét tároljuk, amihez aztán majd hozzá kell adnunk.

Mind az InputComponent, mind az OutputComponent osztályok tömbként tárolják a Pin-jeiket. Ezt a döntést az befolyásolta, hogy létrehozás után szinte minden elemnek konstans a lábszáma, emiatt ez tűnt a legyegyszerűbbnek. Plusz az indexeléssel könnyen lehet azonosítani a bemeneteket (pl. ha bonyolultabb elemekkel bővítjük esetleg később a modellt pl. muxi, stb.). Az egyetlen kivételt a konstans jellegre a Node class jelenti, de ezt majd ott tárgyalom részletesen.

Az OutputComponent osztály fő funkciója a jelkiküldés, míg az InputComponent class-nak aktivizálás (ha készen áll), illetve a resetelés szimuláció előtt.

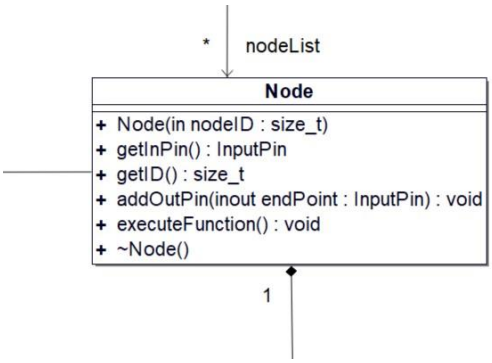
Az IOComponent csak köztes class, nem ad extra funckiót (többszörösen örököl a másik kettőtől).

Node osztály

A Node osztály fő feladat a csomópont funkcionalitás lefedése. Ezt az áramkör megadásakor a felhasználó félig implicit adja meg, hiszen számokkal jelzi, hogy melyik csomópontra kapcsolódnak az egyes elemek. Mint azt majd a konfigurálás leírásakor megfigyelhetjük, emiatt bár bemenetként konstans funkcionál (1 bemenete van ahonnan fogad jelet), mégis a kimenete folyamatosan nőni fog (ahogy több kaput csatlakoztatunk a kimenetére).

Emiatt a kimenetét nem tudjuk sima tömbben tárolni. Szerencsére rendelkezésünkre áll már ezek tárolására és bővítésére alkalmas osztály, a Queue. Emiatt azonban az OutputComponent interfész nem felel meg neki, magának kell implementálnunk a kimenetet. Mivel ez csak egy láncolt listában tárolja a kimeneti pineket, minden funkcionalitás szinte ugyanaz (bejárás persze máshogy történik, de ez semmivel nem rosszabb jelküldéskor, hiszen amúgy is az egész tömbön végig kell mennünk hogy minden pinről kiküldjük a jelet).

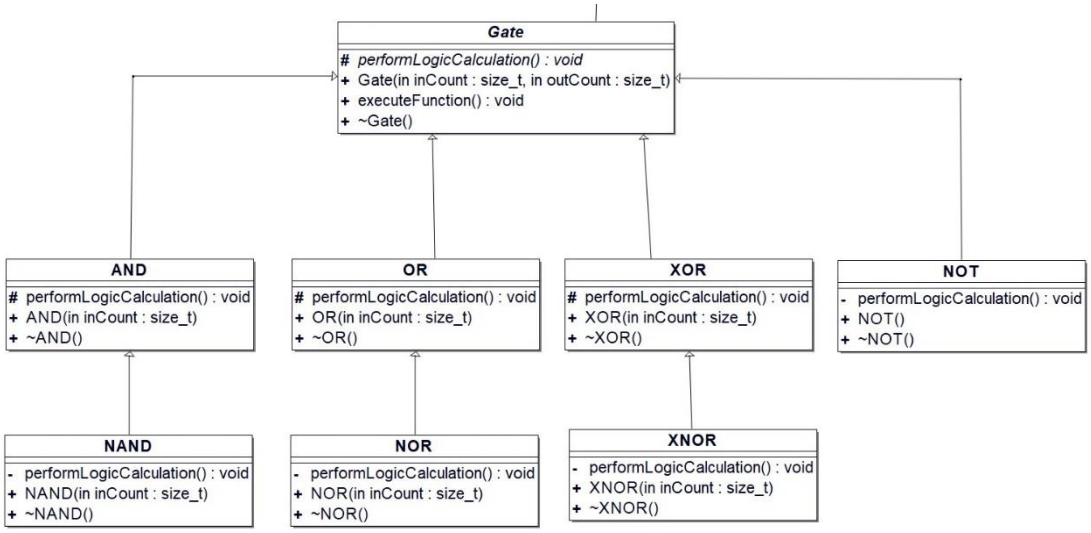
Ennek az eredménye az alábbi osztály („- id: size_t” most nem látszik, mert a felhasznált UML program reverse engingeer funckiója ezt nem adta hozzá):



(A két Pin-es elnevezésű tagfüggvény összekötéskor lesz hasznos, emellett lekérdezhetjük a hozzárendelt számot, ami ID-ként funckionál.)

Gate osztályok

A kapu osztályok valósítják meg a logikai kapuk működését. Ezek az IOComponent-től örökölnek, és nem rendelkeznek különleges saját funkciókkal, egyedül csak előállítják a logikai bemenet alapján a kimentet, amit tovább küldenek:



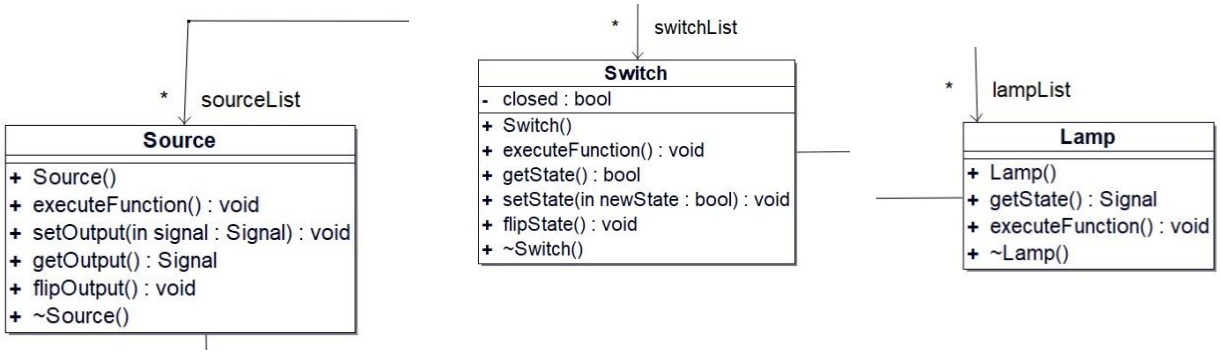
A Gate osztály főleg a többi IOComponent-től szolgál elkülönítésként, csak köztes, absztrakt osztály szerepe van.

Elsőre furcsa lehet, hogy a negált kapukat öröklés útján hozzuk létre, de itt (bár ábra nem jelzi), korlátozó öröklést alkalmazunk. Ennek fő oka, hogy minden szempontból azonosan funckionálnak, csak a kimenetüket kell megfordítani, emiatt egyszerűbb újrahasználni a sima kapukban definiált függvényeket.

Mint ahogy le is olvasható, a kapuknak tetszőleges bemenete lehet. Ennek a fő oka, hogy semmilyen komplikációt nem okoz ennek az implementálása, hiszen minden kapunak csak 1 kimenete lesz, tehát az első n-1 megadott csomópont mind bemenet, ráadásul ezek szerepe szimetrikus, ezért a sorrend mindegy.

Periféria jellegű osztályok

A specifikációban is látható volt, de alapvetően 3 elem ki és bemenetét figyeljük: a források, a kapcsolók és a lámpák. Ezek közösen hasonlóan periféria jelleget mutatnak, emiatt az alábbi módon terveztem meg őket:



A forrás osztály az egy kimeneti Pin-jén tárolja a jelét, ezt lehet állítani és lekérdezni, illetve jelet küldeni vele.

A kapcsoló tárolja, hogy zárt-e vagy nem, ez alapján dönti el, hogy milyen jelet ad tovább, ha zárt, akkor a bemeneti Pin jelét, amúgy a kimeneti pin jelét. Az állapota ennek is állítható, és lekérdezhető.

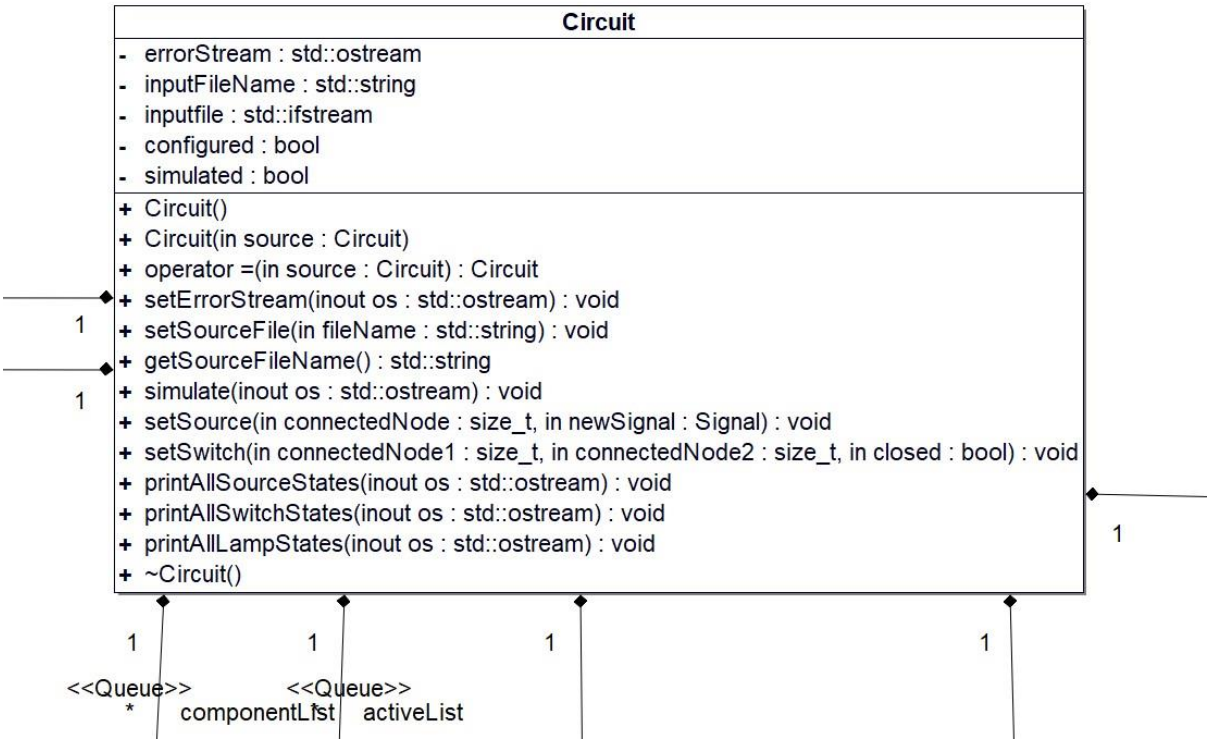
A lámpa csak az egyetlen bemeneti Pin-jén fogadja, más funckiója nincs. Az állapotot itt is le tudjuk kérdezni.

Mindhárom osztályhoz tartoznak inserter-ek is, melyekkel ki lehet íratni őket egy output stream-re.

Circuit osztály

A legfőbb osztály, és egyben a feladat „végterméke”, ezt tudja a felhasználó felkonfigurálni, adatokat beállítani, és szimulációkat futtatni rajta.

A specifikációban kitűzött célok mellett egyéb, nehezebb feladatokra is lehet esetlegesen képes, és nem kizárt, hogy fejlesztés alatt ne legyen még bővítve az API, de egyelőre kezdetlegesen az alábbi funkciókat tervezem biztosan megvalósítani:



A következők az ábráról is leolvasható, de érdemes lejegyezni őket:

- Állítható error streamje van, ha pl. file-ba karjuk átirányítani (ez alapértelmezetten a std::cerr)
- Állítható a bemeneti file-ja, amelyből felkonfigurálunk (ennek nevét is tárolja)
- Mindig tárolja, hogy fel van-e konfigurálva és le van-e szimulálva (feles konfig és szimulálás kikerüléséhez)
- Szimuláláskor megadjuk hogy hova irányítsa a kimenetet
- Be tudjuk állítani bizonyos csomópontra kapcsolt források/kapcsolók állapotát (jelzi, ha nincs megadott)
- Ki lehet vele íratni az egyes periféria elemek adatait (lesz ehhez is inserter operátor, amivel tudja a szimulátor kiírni)

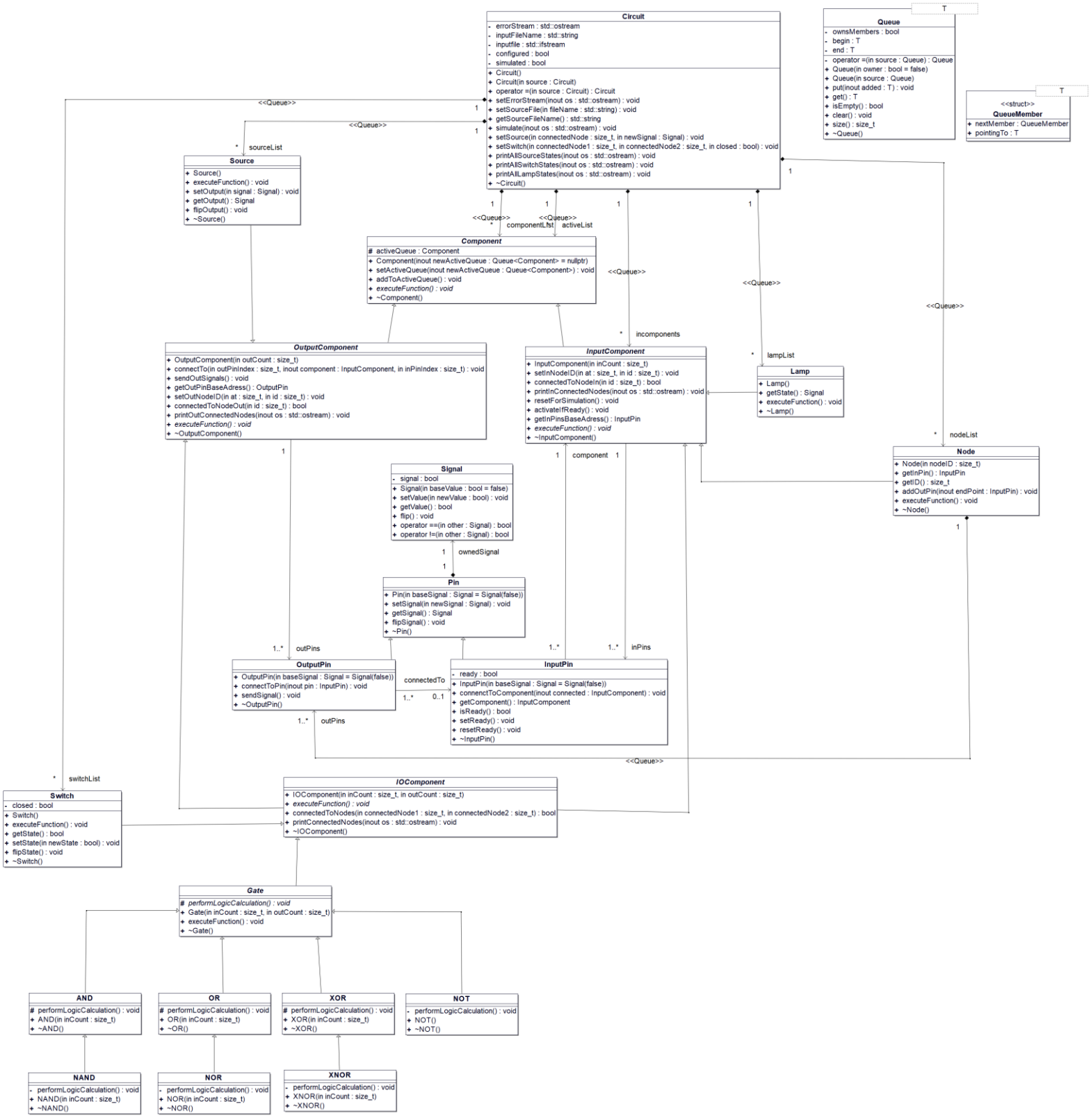
Alap konstuktorja üresen hozza létre. Felkonfigurálás automatán történik (általában első szimuláció során), de csak akkor konfigurál fel, ha szükséges, feleslegesen nem konfigurál újra.

Az áramkör maga felelős a dinamikus memória kezelésért: a copy konstruktor és assign operátor kezelik megfelelően a memóriát, azaz a copy egy teljesen azonos, független másolatot hoz létre, és ehhez hasonlóan az egyenlőség operátor is, de ez törli az előző áramkör memóriáját előtte.

Az elemek tárolására megint a Queue osztályt fogjuk felhasználni, terv szerint az alábbi listák lesznek számon tarva benne (az ábrán a vonalak ezeket jelzik, csak nem teljesen látszanak):

- Component list: Az összes komponensre pointer, ez a memória felszabadításakor lesz fontos, ez törli őket.
- Active list: Ez valósítja meg az aktív FIFO-t, minden elem ide kerül, ha készen áll a kiértékelésre
- Node list: A csomópontok listája, építéskor innen keresi elő melyikhez kell adni újabb output Pin-t
- InputComponent list: Ez a szimuláció előtti reseteléshez fog kelleni, ide rakunk minden resetelendő bemenettel rendelkező objektumra mutatót
- Switch list: a kapcsolók mutatóit tárolja a kiolvasáshoz és állításhoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)
- Lamp list: a lámpák mutatóit tárolja a kiolvasáshoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)
- Source list: a források mutatóit tárolja a kiolvasáshoz és állításhoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)

Teljes ábra


Circuit
- errorStream : std::ostream
- inputFileName : std::string
- inputfile : std::ifstream
- configured : bool
- simulated : bool
+ Circuit()
+ Circuit(in source : Circuit)
+ operator =(in source : Circuit) : Circuit
+ setErrorStream(inout os : std::ostream) : void
+ setSourceFile(in fileName : std::string) : void
+ getSourceFileName() : std::string
+ simulate(inout os : std::ostream) : void
+ setSource(in connectedNode : size_t, in newSignal : Signal) : void
+ setSwitch(in connectedNode1 : size_t, in connectedNode2 : size_t, in closed : bool) : void
+ printAllSourceStates(inout os : std::ostream) : void
+ printAllSwitchStates(inout os : std::ostream) : void
+ printAllLampStates(inout os : std::ostream) : void
+ ~Circuit()
T
Queue
- ownsMembers : bool
- begin : T
- end : T
- operator =(in source : Queue) : Queue
+ Queue(in owner : bool = false)
+ Queue(in source : Queue)
+ put(inout added : T) : void
+ get() : T
+ isEmpty() : bool
+ clear() : void
+ size() : size_t
+ ~Queue()
T
<<struct>>
QueueMember
+ nextMember : QueueMember
+ pointingTo : T
Source
+ Source()
+ executeFunction() : void
+ setOutput(in signal : Signal) : void
+ getOutput() : Signal
+ flipOutput() : void
+ ~Source()
Component
activeQueue : Component
+ Component(inout newActiveQueue : Queue<Component> = nullptr)
+ setActiveQueue(inout newActiveQueue : Queue<Component>) : void
+ addToActiveQueue() : void
+ executeFunction() : void
+ ~Component()
OutputComponent
+ OutputComponent(in outCount : size_t)
+ connectTo(in outPinIndex : size_t, inout component : InputComponent, in inPinIndex : size_t) : void
+ sendOutSignals() : void
+ getOutPinBaseAdress() : OutputPin
+ setOutNodeID(in at : size_t, in id : size_t) : void
+ connectedToNodeOut(in id : size_t) : bool
+ printOutConnectedNodes(inout os : std::ostream) : void
+ executeFunction() : void
+ ~OutputComponent()
InputComponent
+ InputComponent(in inCount : size_t)
+ setInNodeID(in at : size_t, in id : size_t) : void
+ connectedToNodeIn(in id : size_t) : bool
+ printInConnectedNodes(inout os : std::ostream) : void
+ resetForSimulation() : void
+ activateIfReady() : void
+ getInPinsBaseAdress() : InputPin
+ executeFunction() : void
+ ~InputComponent()
Lamp
+ Lamp()
+ getState() : Signal
+ executeFunction() : void
+ ~Lamp()
Node
+ Node(in nodeID : size_t)
+ getInPin() : InputPin
+ getID() : size_t
+ addOutPin(inout endPoint : InputPin) : void
+ executeFunction() : void
+ ~Node()
Signal
- signal : bool
+ Signal(in baseValue : bool = false)
+ setValue(in newValue : bool) : void
+ getValue() : bool
+ flip() : void
+ operator ==(in other : Signal) : bool
+ operator !=(in other : Signal) : bool
Pin
+ Pin(in baseSignal : Signal = Signal(false))
+ setSignal(in newSignal : Signal) : void
+ getSignal() : Signal
+ flipSignal() : void
+ ~Pin()
OutputPin
+ OutputPin(in baseSignal : Signal = Signal(false))
+ connectToPin(inout pin : InputPin) : void
+ sendSignal() : void
+ ~OutputPin()
InputPin
- ready : bool
+ InputPin(in baseSignal : Signal = Signal(false))
+ connenctToComponent(inout connected : InputComponent) : void
+ getComponent() : InputComponent
+ isReady() : bool
+ setReady() : void
+ resetReady() : void
+ ~InputPin()
Switch
- closed : bool
+ Switch()
+ executeFunction() : void
+ getState() : bool
+ setState(in newState : bool) : void
+ flipState() : void
+ ~Switch()
IOComponent
+ IOComponent(in inCount : size_t, in outCount : size_t)
+ executeFunction() : void
+ connectedToNodes(in connectedNode1 : size_t, in connectedNode2 : size_t) : bool
+ printConnectedNodes(inout os : std::ostream) : void
+ ~IOComponent()
Gate
performLogicCalculation() : void
+ Gate(in inCount : size_t, in outCount : size_t)
+ executeFunction() : void
+ ~Gate()
AND
performLogicCalculation() : void
+ AND(in inCount : size_t)
+ ~AND()
OR
performLogicCalculation() : void
+ OR(in inCount : size_t)
+ ~OR()
XOR
performLogicCalculation() : void
+ XOR(in inCount : size_t)
+ ~XOR()
NOT
- performLogicCalculation() : void
+ NOT()
+ ~NOT()
NAND
- performLogicCalculation() : void
+ NAND(in inCount : size_t)
+ ~NAND()
NOR
- performLogicCalculation() : void
+ NOR(in inCount : size_t)
+ ~NOR()
XNOR
- performLogicCalculation() : void
+ XNOR(in inCount : size_t)
+ ~XNOR()
<<Queue>>
sourceList
componentLIst
activeList
incomponents
lampList
nodeList
component
ownedSignal
outPins
inPins
connectedTo
switchList

5. Megvalósítás

Változtatások, bővítések

A fejlesztés során alapvetően nem volt szükség a modell gyökeres megváltoztatására, az osztályok perifériája többé-kevésbé azonos maradt a tervhez képest, főleg kisebb-nagyobb finomításokon ment keresztül a kommunikáció gördülékenysége érdekében.

A következőkben először szeretném bemutatni ezen fő változásokat és hatásaikat a program felépítésére.

Kommentek

A felhasználónak mostantól van lehetősége a konfigurálási file-ban kommenteket írni, hasonlóan a legtöbb programozási nyelvhez. A végleges verzióban C++ stílusú egysoros kommenteket lehet berakni a konfigurációs file-okba, ezzel segíteni a leírt áramkör működésének megértését.

Valójában ez főleg kényelmi szempontból került be a végleges implementációba, funckionális haszna valójában nincsen. Mivel nem esszenciális a feladathoz, emiatt itt úgy ítéltem meg, hogy itt értelemes megemlíteni, mivel nincs meghatározó a funckionális haszna, tisztán a szépítés miatt került be a programba.

Saját kivételek

A fejlesztés alatt viszonylag hamar egy hiány jelentkezett, mégpedig hogy a kivételkezelés típusossága nem volt elegáns, hiszen mindenhol string-eket dobtam, és ezek nem feleltek meg teljes mértékben az elvártaknak, van ahol több információt is szeretnék közölni, mint hogy mi az oka, emiatt hoztam létre saját kivétele osztályokat. (Meg kevésbé elegáns kódhoz is vezettek…)

Bár használhatóak lettek volna a szabványos kivételek is, azonban számomra nem igazán felelt meg egyik sem, nem tükrözte a program belső logikájának felépítését, emiatt úgy döntöttem, hogy saját kivételeket írok, de a std::exception-ből származtatom, hogy egységesen elkaphatóak legyenek.

A legtöbb kivétel alapvetően a belső működésben játszik szerepet, nem is találkozik vele a felhasználó, hiszen az áramkör osztály önmagában

lerendezi ezt, legtöbbször vagy le tudja kezelni a rendezését, vagy jelzi a hiba kimenetre a problémát.

Kettő kivételt azonban tud dobni, jelezve a két fő problémákat használatkor: egyik, hogy ha felkonfigurálás során történt hiba, akkor azt kiírja az errorstream-re, de emellett dob is egy ConfigurationError exceptiont is. A másik lehetőség akkor van, ha nem létező periféria elemnek változtatnánk a jelét/állapotát. Ekkor egy MatchingComponentNotFound kivételt dob az áramkör.

Queue iterátor

A következő bővítés a saját Queue osztályban történt. Gyakran volt szükség a programban, hogy végigmenjek egy listán, és valamit hajtsak végre minden elemén. Ezt kezdetben a lista lemásolásával, és ezután az elemek egyes kiszedésével kezeltem, de ez hamar célszerűtlennek tűnt, emiatt csináltam egyszerű iterátort a Queue osztályhoz, amivel végig lehet rajta futni, extra másolgatás nélkül (egyszerű, mert csak “sima” iterátor van, nincsen const és reverse sem).

Circuit interface-nek bővítése

A tesztelés miatt célszerűnek tűnt, hogy a Circuit osztály képes legyen nem csak kíirni a jeleit, hanem ki is lehessen “olvasni” egy Signal osztályként (vagy a kapcsolónak bool változóban azt, hogy zárt-e vagy nem) az információkat. Emiatt kapott mindhárom periféria elemhez (Source, Switch, Lamp) kapott lekérdező függvényeket, mellyel adott csomópontokhoz kapcsolódó elemeknek az adatját le lehet kérdezni.

Ha nincs ilyen elem, akkor viszont az áramkör dob egy kivételt (MatchingComponentNotFound), jelezve hogy nem létező elem állapotát próbáltuk lekérdezni.

Component interface bővítése

Ahogy a tervben is leírtam már, az egyik fontos szemantikai ellenőrzés a felkonfigurálászkor az, hogy nincsen-e elszigetelt elem-e, avagy nincs helytelen működéshez vezető, leszimulálatlan elemcsoport. Emiatt kell futtatni egy ellenőrző tesztet felkonfigurálászkor. Először próbáltam a Component osztály módosítása nélkül megvalósítani az ellenőrzést, de legtöbb implementáció lassú vagy nehezen értelmezhető kódhoz vezetett, emiatt döntöttem, hogy a Component interface-n keresztül tudjuk ezt ellenőrizni, ami amiatt is kényelmes, mert a már

létező tárolóban kell csak végig futnunk, ellenőrizve, hogy minden elem le lett-e szimulálva.

Ez a bővítés egy extra bool-t adott az interface-hez, melyet lehet beállítani, resetelni és olvasni. Ez a bool érték tárolja, hogy le lett-e szimulálva az adott Component, összes kiolvasásával kiderül hogy minden áramköri elem ki lett-e értékelve.

Pin-ek elérésének módosítása

A tervben még az látható, hogy az InputComponent és OutputComponent osztályok pin tömbjének alapcíme elérhető, amivel az egyes pin-ekhez lehet hozzáférni. Azonban csak a tömb címének kiadása utólag igen veszélyesnek tűnt, emiatt célszerűbbnek tűnt, hogy egy at-hez hasonló fügvénnyel kérjük le egyes pin-eket, és helytelen indexelés esetén dobunk egy kivételt, jelezve hogy rosszul használtuk, nem létező pin indexet akartunk elérni (belső működésnek szól).

Programozói dokumentáció

A programozói dokumentációról röviden

A kód részletekbe menő működése az egyes forráskódokban megtalálható, szemantikai információkat nem fejteném ki itt, mivel a fontos gondolatok már korábban megmagyarázva voltak, és egyik algoritmus sem olyan bonyolult hogy itt jelentősebb említést igényelne, amit nem magyaráztam volna meg korábban, vagy esetleg ne lehetne kiolvasni a kódból.

Azonban mégis szükségszerűnek tűnt a végleges deklarációknak a bemutatása, miután részben eltérnek a tervtől. Emiatt azt éreztem szükségesnek, hogy itt egy közös szekcióban fejtsem ki a végleges állapotokat. A legtöbb itteni információ felszínesnek tűnhet, ez a célja is, mert inkább a funkcionalitás egyszerű leírása érdekében készült, a konkrét működés megtalálható a forráskódban.

A módosításokkal kapcsolatban

Miután a terv szekciót nem kívánom módosítani, hogy a fejlesztés során történt változások követhetőek legyenek, emiatt itt röviden szeretném mégegyszer bemutatni az osztályoknak a végleges interface-ét és egymással létező kapcsolatát bemutatni, miután az előző rész alapján látható, hogy történtek módosítások a tervezési fázishoz képest az egyes osztályok API-jában. Ha van fontos extra információ, ami változott a tervhez képest, akkor azt ebben a szekcióban lehet megtalálni.

A tervezési részhez képest itt nem tervezek teljes UML diagramot mutatni, mivel nem volna a leghasznosabb, a lényegi struktúra/hierarchia és kapcsolatok már ott is megtalálhatóak, de az egyes osztályok közvetlen kapcsolatát itt is bemutatom, hogy egyértelmű legyen.

Signal

A Signal osztály a jel modellezésére készült, ennek a funkciója nem is változott a tervhez képest, mivel viszonylag szimpla osztály volt kezdetektől fogva.

Tagváltozók:

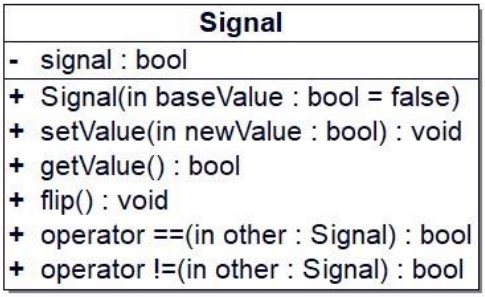
```
/**
 * @brief A tárolt jelérték.
 * @brief false = LOW (0) jel, true = HIGH (1) jel
 */
bool signal;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza a jel objektumot, adott értékkel, ha van.
 *
 * @param baseValue Az alapérték.
 */
Signal(bool baseValue = false)
/**
 * @brief Beállítja egy új értékre a jelet.
 *
 * @param newValue Az új jel értéke.
 */
void setValue(bool newValue)
/**
 * @brief Visszaadja a jel értékét.
 *
 * @return true = 1 a jelérték.
 * @return false = 0 a jelérték.
 */
bool getValue() const

/**
 * @brief Megfordítja a jelértéket.
 */
void flip()

/**
 * @brief Egyenlőséget vizsgálja két jelszint között.
 *
 * @param other A másik jel amivel hasonlítunk.
 * @return true, ha egyeznek.
 * @return false, ha nem egyeznek
 */
bool operator==(const Signal& other)
```

```
  /**
   * @brief Nem egyenlőséget vizsgálja két jelszint között.
   *
   * @param other A másik jel amivel hasonlítunk.
   * @return true, ha nem egyeznek.
   * @return false, ha egyeznek
   */
  bool operator!=(const Signal& other)
```

UML ábra:



Pin

A Pin osztály az áramkör elemek lábainak reprezentálásra szolgáló osztályként jött létre, inkább absztrakt osztály, mert származtatunk belőle, nincsen példányosítva a végleges programban (és nem is ajánlott, de lehetséges haszna lehet a jövőben, ezért nincsen tiltva).

Tagváltozók:

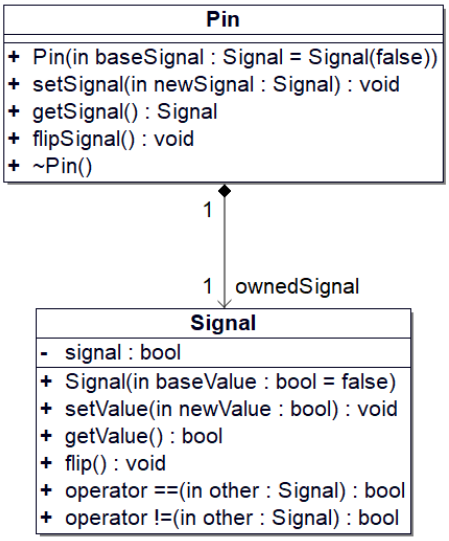
```
/**
 * @brief A pin által birtokolt jel.
 *
 */
Signal ownedSignal;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehoz egy pin-t kezdő jelértékkel.
 *
 * @param baseSignal Az alap jelérték, default-ként LOW (0) jelszinttel.
 */
Pin(Signal baseSignal = Signal(false));

/**
 * @brief Beállítja a pin jelét.
 *
 * @param newSignal Az új jel.
 */
void setSignal(const Signal& newSignal);

/**
 * @brief Visszaadja a pin jelét. Mivel kicsi a Signal osztály, ezért nem
          kell pointer/referencia.
 *
 * @return A visszaadott jel.
 */
Signal getSignal() const;

/**
 * @brief Megfordítja a pin jelét.
 *
 */
void flipSignal();
```

```
  /**
   * @brief Virtuális a destruktor az öröklés miatt.
   *
   */
  virtual ~Pin();
```

UML ábra:



InputPin

Az InputPin osztály a bemeneti pin-ek modellezésére szolgál, melyeken keresztül egy jelet tudunk átadni egy áramköri elemnek, a Pin osztály specializációja. A jelfogadás során nincs tisztában a partnerével, ő csak jelet kapja, viszont tudja melyik elemnek része, mert annak jelzi, hogy jelet kapott, hogy az tudja ellenőrizni, hogy készen áll-e jelfeldolgozásra.

Tagváltozók:

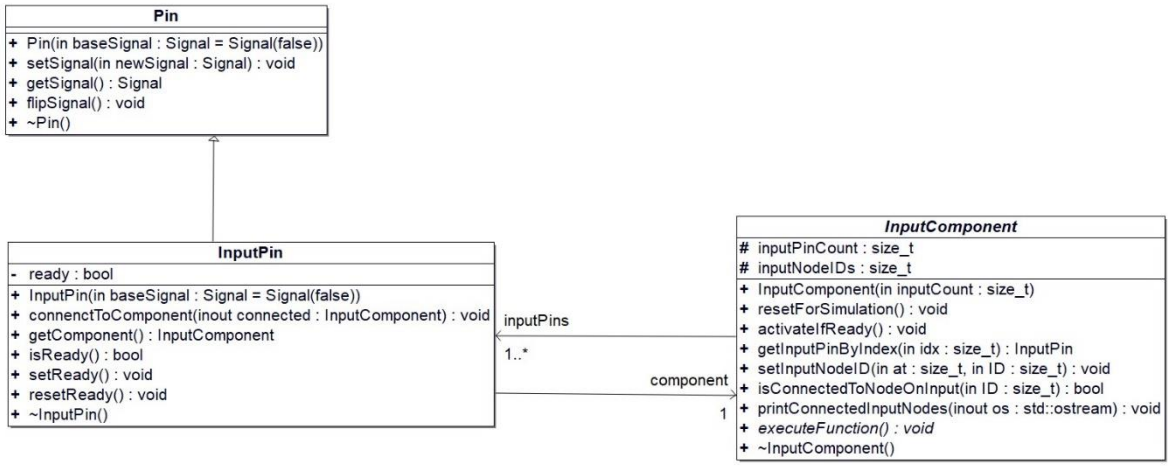
```
/**
 * @brief Az áramköri elem, melynek része a bemeneti pin.
 *
 */
InputComponent* component;
/**
 * @brief Tárolja, hogy készen áll a pin feldolgozásra.
 *
 */
bool ready;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehoz egy bemeneti pin-t, komponensét NULL-ra állítva.
 *
 * @param baseSignal Az alapjel, ha van megadva.
 */
InputPin(Signal baseSignal = Signal(false));

/**
 * @brief Egy áramköri elemhez köti a bemeneti pin-t, így tud majd neki
          üzenni.
 *
 * @param component Az áramköri elem, amihez kötjük.
 */
void connenctToComponent(InputComponent* connected);

/**
 * @brief Visszaadja a pointerét arra az elemre, amihez kapcsolódik.
 *
 * @return Az elem, aminek része.
 */
InputComponent* getComponent() const;
```

```
    /**
     * @brief Visszaadja, hogy készen áll-e kiértékelésre a bemeneti pin, azaz
     *        erről a pinről már meg van-e a helyes bemenet.
     *
     * @return true = ha készenáll.
     * @return false = ha még nem áll készen
     */
    bool isReady();

    /**
     * @brief Jelzi az áramköri elem felé, hogy ezen a lábán rendelkezésre áll
     *        a jel.
     *
     * @exception NonExistentConnection = Nincs kapcsolt InputComponent eleme.
     */
    void setReady();

    /**
     * @brief Reseteli a készenlétet, azaz a pin-nek beállítja, hogy még nem
     *        áll készen adat feldolgozásra.
     *
     */
    void resetReady();

    /**
     * @brief Virtuális a destruktor az öröklés miatt.
     *
     */
    virtual ~InputPin();
```

UML ábra:



OutputPin

Az OutputPin osztály a kimeneti pin-ek modellezését hajtja végre. Kiszámított jel továbbítását végzi el, ez a Pin osztály másik specializációja. Mindig egy InputPin-nek üzen, emiatt csak azzal van tisztában, hogy kinek üzen, azaz melyik InputPin-ek küldi a jelét, azzal nincs tisztában, hogy melyik áramköri elemhez tartozik. (Tervhez képest ez az osztály sem változott sokat, csak kivételkezelése finomult.)

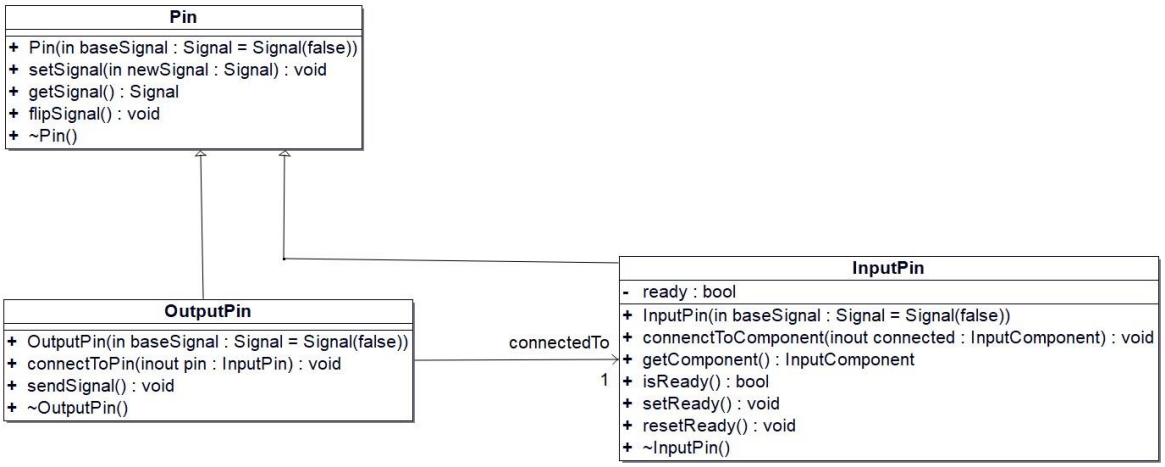
Tagváltozók:

```
/**
 * @brief A bemeneti pin, amihez van kapcsolva.
 *
 */
InputPin* connectedTo;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehoz egy kimeneti pin-t, kapcsolt bemeneti pin-jét NULL-ra
          állítva.
 *
 * @param baseSignal Az alapjel, ha van megadva.
 */
OutputPin(Signal baseSignal = Signal(false));

/**
 * @brief Összekapcsolja egy bemeneti pin-nel.
 *
 * @param pin A kapcsolni kívánt pin.
 */
void connectToPin(InputPin* pin);

/**
 * @brief Jelet küld a kapcsolt bemeneti pin-nek.
 *
 * @exception NonExistentConnection = Nincs kapcsolt bemeneti pin.
 * @exception ShortCircuit = Már aktivált bemeneti pin-nek üzen.
 */
void sendSignal() const;

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~OutputPin();
```

UML ábra:


Pin
+ Pin(in baseSignal : Signal = Signal(false))
+ setSignal(in newSignal : Signal) : void
+ getSignal() : Signal
+ flipSignal() : void
+ ~Pin()
OutputPin
+ OutputPin(in baseSignal : Signal = Signal(false))
+ connectToPin(inout pin : InputPin) : void
+ sendSignal() : void
+ ~OutputPin()
connectedTo
1
InputPin
- ready : bool
+ InputPin(in baseSignal : Signal = Signal(false))
+ connenctToComponent(inout connected : InputComponent) : void
+ getComponent() : InputComponent
+ isReady() : bool
+ setReady() : void
+ resetReady() : void
+ ~InputPin()

Component

Az áramköri elemeket megvalósító interface osztályoknak az “legősebb” tagja, ő áll az öröklési hierarchia tetején. Minden amire ez az osztály képes, arra minden áramköri elemnek képesnek kell lennie.

Absztrakt osztály, hiszen csak annyit biztosít, hogy közös ősön keresztül heterogén kollekcióban tudjuk majd tárolni az összes “legyártott” objektumot, hogy egységesen tudjuk felszabadítani őket. Másfelől a közös ellenőrzések is ezeken keresztül futnak, mint például a szimulálatlan elemek keresése.

Tagváltozók:

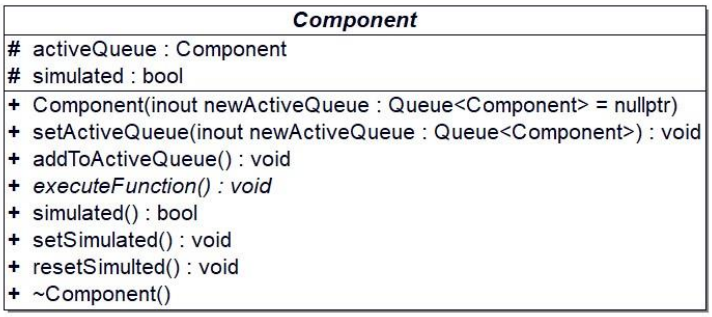
```
/**
 * @brief Az aktív sor, amihez kell hozzáadni, ha ki kell értékelni az
          elemet, azaz végrehajtani a funkcióját.
 *
 */
Queue<Component>* activeQueue;

/**
 * @brief Le lett-e szimulálva az áramkör elem. (true = igen, false = nem)
 *
 */
bool gotSimulated;
```

Publikus tagfüggvények:

```
/**
 * @brief Felparaméterezi az aktív FIFO-t.
 *
 * @param newActiveQueue Az elem aktív FIFO-ja, NULL ha nincs neki megadva
                         még.
 */
Component(Queue<Component>* newActiveQueue = nullptr);

/**
 * @brief Beállítja az aktív FIFO-t.
 *
 */
void setActiveQueue(Queue<Component>* newActiveQueue);

/**
 * @brief Hozzáadja az aktív sorhoz az áramköri elemet.
 *
 */
void addToActiveQueue();
```

```
    /**
     * @brief Végrehajtja a funkcióját az áramköri elemnek. Leszármazottban
     *        konkretizálva.
     *
     */
    virtual void executeFunction() = 0;

    /**
     * @brief Visszaadja, hogy le volt-e szimulálva az elem.
     *
     * @return true = már le volt szimulálva ;
     * @return false = még nem volt leszimulálva
     */
    bool simulated();

    /**
     * @brief Beállítja, hogy szimulálva volt az elem.
     *
     */
    void setSimulated();

    /**
     * @brief Reseteli a szimuláltság státuszát.
     *
     */
    void resetSimulted();

    /**
     * @brief Virtuális destruktor öröklés miatt.
     *
     */
    virtual ~Component();
```

UML ábra:



InputComponent

A bemenettel rendelkező áramköri elemek interface osztálya, ez az absztrakt osztály konkretizálja, hogy hogyan viselkedik egy konstans számú bemenettel rendelkező osztály.

Fontos itt az a jellemző róla hogy konstans bemenetű, ha például egy olyan áramköri elemet akarunk implementálni, amely felkonfigurálás során dinamikusan bővítheti a bemeneti lábainak számát, akkor ez az osztály nem megfelelő. Viszont a jelenlegi modellben a bemenetek mindig létrehozáskor fixáltak, szóval általánosan lefed minden bemenettel rendelkező áramköri elem viselkedését. (Ha jövőben bővítenénk dinamikusan növelhetővel, akkor a nevet érdemes változtatni, jelezve hogy ez egy statikusabb osztály.)

A bemeneteken kívül tárolja a kapcsolt csomópontok ID-kat, így minden elemre gyorsan meg lehet állapítani, hogy mely csomópontokhoz kapcsolódik, ez főleg a periféria osztályoknak hasznos, ahol meg kell tudni keresni, hogy melyik kapcsolódik a megfelelő csomópontokra (vagy ha nincs ilyen elem).

Tagváltozók:

```
/**
 * @brief A bemeneti pin-ek száma.
 *
 */
size_t inputPinCount;

/**
 * @brief A bemeneti pin-ek tömbje. Az index jelentését a specifikus
          alkatrész adja meg.
 *
 */
InputPin* inputPins;

/**
 * @brief A bemeneti csomópontok ID-jai kiolvasáshoz.
 *
 */
size_t* inputNodeIDs;
```

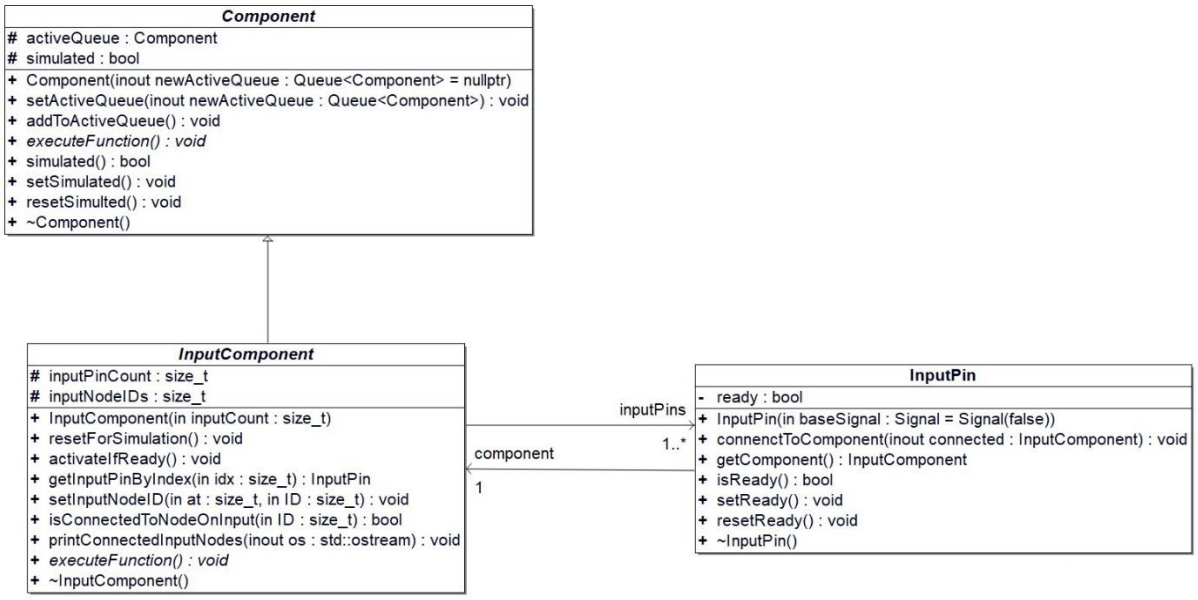
Publikus tagfüggvények:

```
/**
 * @brief Létrehozza a bemeneti pin-ek tömbjét.
 * @param inputCount A bemeneti pin-ek száma.
 */
InputComponent(size_t inputCount);
```

```
/**
 * @brief Reseteli a bemeneti pin-ek státuszát szimulációhoz.
 *
 */
void resetForSimulation();

/**
 * @brief Ellenőrzi hogy minden bemeneti pin aktív-e, és berakja az aktív
         FIFO-ba, ha igen.
 *
 */
void activateIfReady();

/**
 * @brief Vissza adja a kívánt indexű bemeneti pin címét, ha létezik.
 *
 * @param idx A kívánt indexű pin.
 * @return InputPin* Az adott indexű pin.
 * @exception std::out_of_range = ha túlindexelünk.
 */
InputPin* getInputPinByIndex(size_t idx) const;

/**
 * @brief Beállítja egy bemeneti csompópont ID-t kiolvasáshoz.
 *
 * @param at A beállított bemenet indexe.
 * @param ID A beállított csomópont ID.s
 * @exception std::out_of_range = ha túlindexelünk.
 */
void setInputNodeID(size_t at, size_t ID);

/**
 * @brief Ellenőri, hogy kapcsolódik-e egy adott csomóponthoz a bemenetén.
 *
 * @param ID A keresett csomópont ID-ja.
 * @return true = kapcsolódik hozzá a bemeneten ;
 * @return false = nem kapcsolódik hozzá a bemeneten
 */
bool isConnectedToNodeOnInput(size_t ID);

/**
 * @brief Kiírja a bemeneten csatlakoztatott csomópontok ID-ját.
 *
 * @param os A kimeneti stream, ahova akarjuk kiírni.
 */
void printConnectedInputNodes(std::ostream& os) const;
```

```
    /**
     * @brief Végrehajtja a funkcióját az áramköri elemnek. Leszármazottban
     *        konkretizálva.
     *
     */
    virtual void executeFunction() = 0;

    /**
     * @brief Törli a bemeneti pin-ek és ID-k tömbjét.
     *
     */
    virtual ~InputComponent();
```

UML ábra:



OutputComponent

A konstans kimenettel rendelkező áramköri elemeket megvalósító osztály, nagyon hasonló felépítéssel a testvéréhez, főleg pár funcionalitásban térnek el az iránybeli különbségek miatt. (Ez is egy interfész osztály, bár heterogén kollekcióként nem tároljuk.)

Hasonló jellemzők mondhatóak el róla mint az InputComponent osztályról a konstans viselkedéssel kapcsolatban, azonban itt már a jelenlegi implementációban is van szerepe ennek, ugyanis a Node osztály pont a konstans viselkedés miatt nem tudja használni, és emiatt kénytelen vagyunk annak külön implementálni a viselkedését kimenet szempontjából.

Természetesen a kapcsolódó csomópontok ID-jai itt is tárolva vannak, és ki is olvashatóak és ellenőrizhetőek.

Tagváltozók:

```
/**
 * @brief A kimeneti pin-ek száma.
 *
 */
size_t outputPinCount;

/**
 * @brief A kimeneti pin-ek tömbje. Az index jelentést a specifikus
          alkatrész adja meg.
 *
 */
OutputPin* outputPins;

/**
 * @brief A kimenet csomópontok ID-jai kiolvasáshoz.
 *
 */
size_t* outputPinIDs;
```

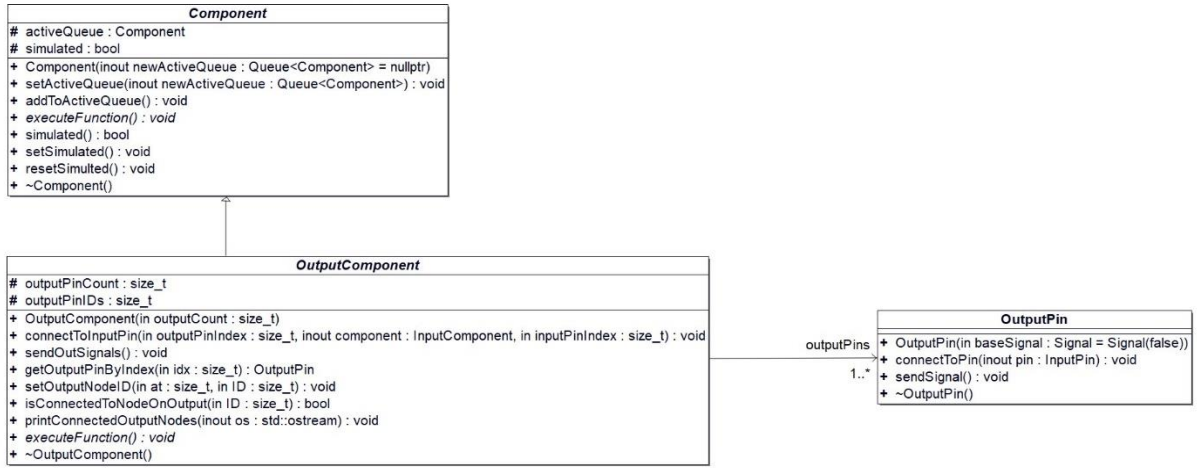
Publikus tagfüggvények:

```
/**
 * @brief Létrehozza a kimeneti pin-ek és csomópont ID-k tömbjét.
 *
 * @param outputCount A kívánt kimeneti pin-ek száma.
 */
OutputComponent(size_t outputCount);
```

```
/**
 * @brief Összeköti a megadott kimeneti pin-jét egy másik áramköri elem
 *        bemeneti pin-jével.
 *
 * @param outputPinIndex A kimeneti pin indexe.
 * @param component Amivel össze szeretnénk kötni.
 * @param inputPinIndex A bemeneti pin indexe.
 *
 * @exception std::out_of_range = ha túlindexelünk.
 */
void connectToInputPin(size_t outputPinIndex, InputComponent* component,
                       size_t inputPinIndex);

/**
 * @brief Kiküldi minden kimeneti lábán a lábakban tárolt jeleket.
 *
 */
void sendOutSignals();

/**
 * @brief Vissza adja a kívánt indexű kimeneti pin címét, ha létezik.
 *
 * @param idx A kívánt indexű pin.
 * @return OutputPin* Az adott indexű pin.
 * @exception std::out_of_range = ha túlindexelünk.
 */
OutputPin* getOutputPinByIndex(size_t idx);

/**
 * @brief Beállítja a kimeneti csomópont ID-t kiolvasáshoz.
 *
 * @param at A beállított kimenet indexe.
 * @param ID A beállított csomópont ID.
 * @exception Ha túlindexelünk, akkor dob egy std::out_of_range-t.
 */
void setOutputNodeID(size_t at, size_t ID);

/**
 * @brief Ellenőri, hogy kapcsolódik-e egy adott csomóponthoz a kimenetén.
 *
 * @param ID A keresett csomópont ID-ja.
 * @return true = kapcsolódik hozzá a kimeneten ;
 * @return false = nem kapcsolódik hozzá a kimeneten
 */
bool isConnectedToNodeOnOutput(size_t ID);
```

```
    /**
     * @brief Kiírja a kimeneten csatlakoztatott csomópontok ID-ját.
     *
     * @param os A kimeneti stream, ahova akarjuk kiírni.
     */
    void printConnectedOutputNodes(std::ostream& os) const;

    /**
     * @brief Végrehajtja a funkcióját az áramköri elemnek. Leszármazottban
     *        konkretizálva.
     *
     */
    virtual void executeFunction() = 0;

    /**
     * @brief Törli a kimeneti pin-ek tömbjét.
     *
     */
    virtual ~OutputComponent();
```

UML ábra:



IOComponent

Ahogyan már a tervben is ki lett fejtve, ez egy tisztán absztrakt osztály, melynek minimális funkciója van önmagába, csak abban segít, hogyha InputComponent és OutputComponent-től is öröklünk, akkor ne kelljen mindig két osztályt külön kezelni, ezen a közös osztályon keresztül egyszerűen lehet mindkettőt örökölni.

Tagváltozók:

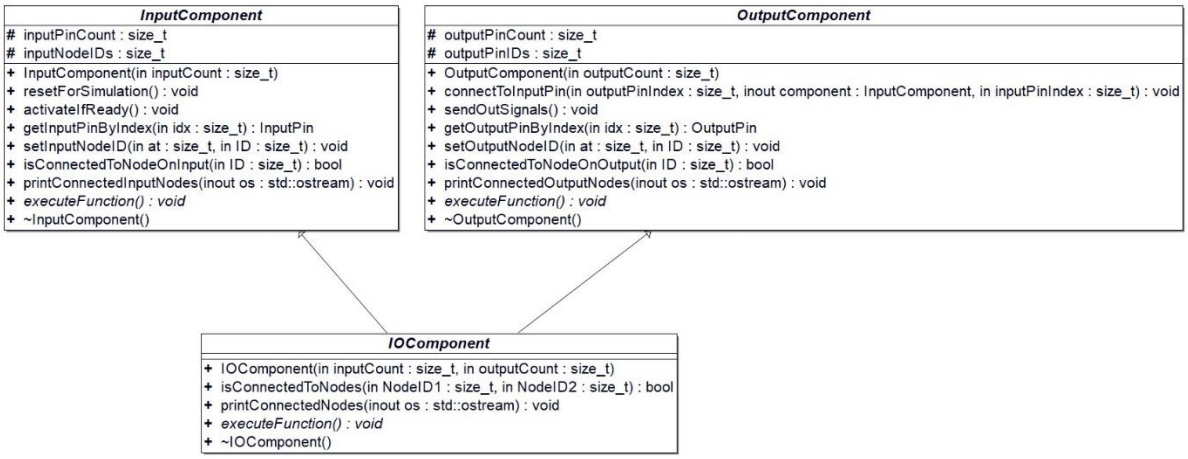
Saját tagváltozója emiatt nincs is, csak amit örökölt a másik kettőtől.

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az IO elemet, mindkét oldali tulajdonságaival.
 *
 * @param inputCount Bemeneti lábak száma.
 * @param outputCount Kimeneti lábak száma.
 */
IOComponent(size_t inputCount, size_t outputCount);

/**
 * @brief Ellenőrzi, hogy kapcsolódik-e két csomóponthoz ellentétes
          oldalról (egyik kimeneti, másik bemeneti, vagy fordítva).
 *
 * @param NodeID1 Az egyik csomópont ID-ja.
 * @param NodeID2 A másik csomópont ID-ja.
 * @return true = kapcsolódik ezekhez ;
 * @return false = nem kapcsolódik ezekhez
 */
bool isConnectedToNodes(size_t NodeID1, size_t NodeID2);

/**
 * @brief Kiírja a kapcsolódó csomópontok ID-jait mind bemenet, mind
         kimenetre (előbb bemenet, utána kimenet).
 *
 * @param os A kimeneti stream, ahova akarjuk kiírni.
 */
void printConnectedNodes(std::ostream& os) const;

/**
 * @brief Végrehajtja a funkcióját az áramköri elemnek. Leszármazottban
         konkretizálva.
 *
 */
virtual void executeFunction() = 0;
```

```
    /**
     * @brief Virtuális destruktor, mert öröklés.
     *
     */
    virtual ~IOComponent();
```

UML ábra:



Node

A csomópontokat megvalósító osztály, minden áramköri elem valójában ezeken keresztül kommunikál a többivel, emiatt is kicsit máshogyan viselkedik mint a többi, kitüntetett szerepe van a működésben.

Mint ahogyan már az OutputComponent osztályban is láttuk, a kimenete a csomópontoknak nem konstans mennyiség, hanem a konfigurálás során folyamatosan bővülő mennyiség lehet. Ennek az oka az, hogy a betöltés során folyamatosan építjük fel az áramkört, nem megyünk benne vissza, és nem is olvassuk ki előre a ki és bemenetek számát egy csomóponton, emiatt kénytelenek kell legyenünk folyamatosan bővíthető kimenetek tárolására.

Szerencsére a Queue osztály erre tökéletesen megfelel, hiszen elég láncolt listában tárolni a kimeneti lábait, mert egyiknek sincsen kitüntetett szerepe, hiszen csak minden kimeneten ki kell küldeni a fogadott jelet.

Minden csomópont az ID-ja alapján azonosítható, ez alapján lehet visszakeresni, ha valahogyan szükségünk lenne rá.

Tagváltozók:

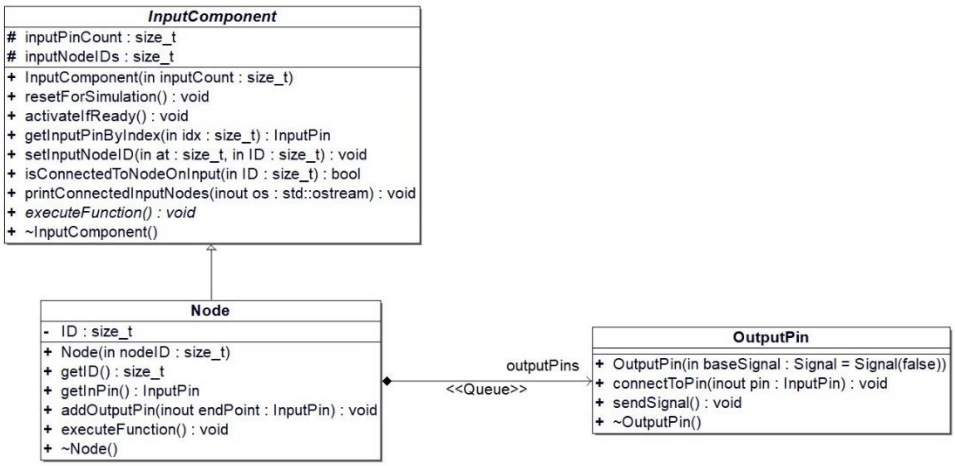
```
/**
 * @brief A csomópont ID-ja, azaz a felhasználó által megadott száma, ez
          alapján lehet azonosítani egy csomópontot összekötéskor.
 *
 */
size_t ID;

/**
 * @brief A kimeneti pin-ek láncolt listája, nem tömb a gyors növelhetőség
          miatt, hiszen létrehozáskor nem tudjuk előre hogy hány lesz.
 *
 */
Queue<OutputPin> outputPins;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza a csomópontot 1 bemenettel, és 0 kimenettel, illetve
          beállítja az ID-t.
 *
 * @param nodeID A csomópont beállított ID-ja.
 */
Node(size_t nodeID);
```

```
    /**
     * @brief Visszaadja a csomópont ID-ját, főleg azonosításhoz használjuk.
     *
     * @return size_t A csomópont ID-ja.
     */
    size_t getID() const;

    /**
     * @brief Visszaadja az egyetlen bemeneti pin-jét.
     *
     * @return InputPin* A bemeneti pin címe.
     */
    InputPin* getInPin();

    /**
     * @brief Hozzáad a kimeneti pin-ek láncolt listájához egy új pin-t, a
            kívánt végponttal.
     *
     * @param endPoint   Ahova küldi majd a jelet a csomópont ezen a kimeneti
                         pin-en.
     */
    void addOutputPin(InputPin* endPoint);

    /**
     * @brief Végrehajtja a funkcióját, azaz minden kimeneti pin-en kiküldi a
            bemeneti pin-en talált jelet.
     *
     */
    virtual void executeFunction();

    /**
     * @brief Virtuális destruktor az öröklés miatt.
     *
     */
    virtual ~Node();
```

UML ábra:



Source

A jelforrásokat megvalósító osztály, a fő szerepük a bemeneti kombináció jeleinek kiadása. A szimuláció során ezek kerülnek be kezdetben az aktív FIFO-ba, így tudnak elkezdődni a kiértékelések egymás után. Fő funkciójuk mellett lehet kiírni állapotukat és kiolvasni is lehet a jelüket is.

Csak OutputComponent-től örököl, mert nincs bemenete.

Tagváltozók:

Nincsenek sajátok, csak amit örököl az OutputComponent interface-től.

Publikus tagfüggvények:

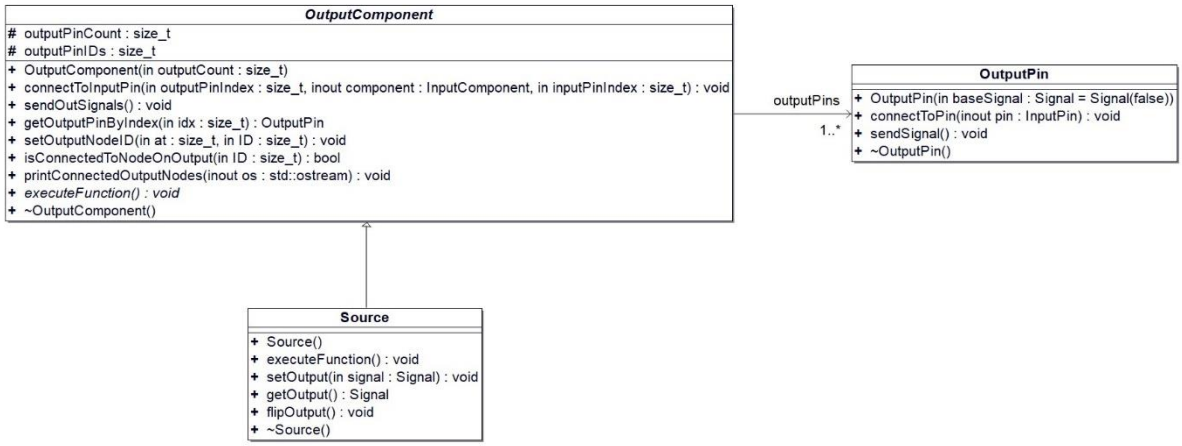
```
/**
 * @brief Létrehoz egy forrást.
 *
 */
Source();

/**
 * @brief Végrehajtja a forrás funkcióját, azaz kiküldi a jelet a
         kimenetén.
 *
 */
virtual void executeFunction();

/**
 * @brief Beállítja a forrás kimeneti jelét.
 *
 * @param signal Az új jel.
 */
void setOutput(const Signal& signal);

/**
 * @brief Visszaadja a forrás kimeneti jelét.
 *
 * @return A forrás jele.
 */
Signal getOutput() const;

/**
 * @brief Megfordítja a kimeneti jel értékét.
 *
 */
void flipOutput();
```

```
    /**
     * @brief Virtuális destruktor öröklés miatt.
     *
     */
    virtual ~Source() {}
```

Globális függvények:

```
/**
 * @brief Kiírja a forrás állapotát a kimeneti streamre.
 *
 * @param os A kimeneti stream.
 * @param x A forrás, amit ki kell írni.
 * @return A kimeneti stream-re referencia, a láncolás miatt.
 */
std::ostream& operator<<(std::ostream& os, const Source& x);
```

UML ábra:



Switch

A kapcsolókat implementáló osztályunk, állapotától függően feltételes jeltovábbítást valósítja meg. Tervhez képest nem történt jelentős változás, az ott leírtakhoz hasonlóan funkcionál.

Fontos hogy akkor is ad ki jelet ha nyitott, csak ekkor LOW jelet ad tovább (mivel valóságban is így működne), azaz a rövidzár ellen nem feltétlen véd meg, ha azt a részt “kikapcsoljuk”.

Tagváltozók:

```
/**
 * @brief Zárva van-e a kapcsoló (true = zárt, false = nyitott)
 *
 */
bool closed;
```

Publikus tagfüggvények:

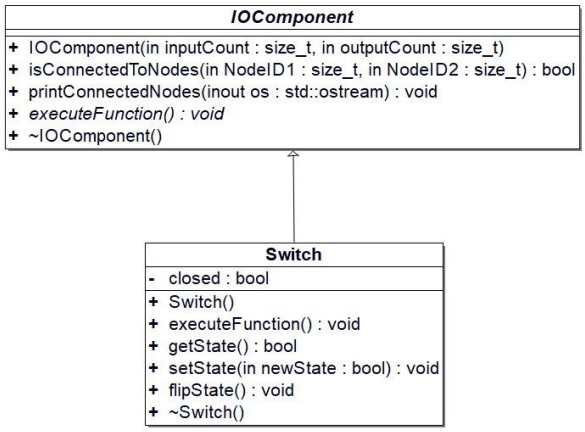
```
/**
 * @brief Létrehozza a kapcsolót.
 *
 */
Switch();

/**
 * @brief Megvalósítja a kapcsolót, azaz ha zárt akkor a bemeneti jelet
 *        továbbítja, egyébként meg LOW (0) jelet ad.
 *
 */
virtual void executeFunction();

/**
 * @brief Vissza adja hogy zárt-e a kapcsoló.
 *
 * @return true = zárt,
 * @return false = nyitott
 */
bool getState() const;

/**
 * @brief Beállítja a kapcsoló állapotát.
 *
 * @param newState Az új állapot. (true = zárt, false = nyitott)
 */
void setState(bool newState);
```

```
    /**
     * @brief Átbillenti másik állapotba a kapcsolót.
     *
     */
    void flipState();

    /**
     * @brief Virtuális destruktor az öröklés miatt.
     *
     */
    virtual ~Switch();
```

Globális függvények:

```
/**
 * @brief Kiírja a kimeneti stream-re a kapcsoló állását.
 *
 * @param os A kimeneti stream.
 * @param x A kapcsoló.
 * @return Kimeneti stream-re referencia, láncolás miatt.
 */
std::ostream& operator<<(std::ostream& os, const Switch& x);
```

UML ábra:



Lamp

A lámpákat implementáló osztály, egyetlen dolga, hogy fogadja a jeleket és ezeket tárolja kiolvasáshoz. Ezen túl nem jelentkezik extra egyedi funkciókkal, de ki lehet írni az állapotát a kimenetre.

Tagváltozók:

Nincsen saját, csak amit InputComponent interface-től örököl.

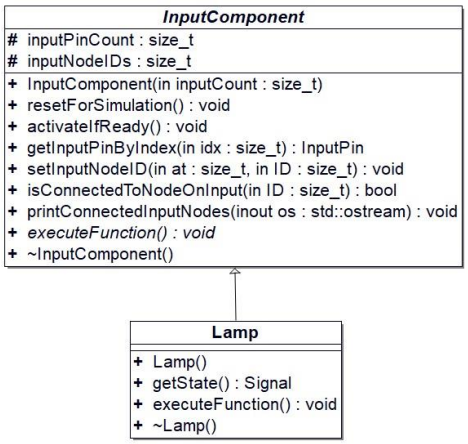
Publikus tagfüggvények:

```
    /**
     * @brief Létrehoz egy lámpát.
     *
     */
    Lamp();

    /**
     * @brief Visszaadja a lámpa jelét, azaz, hogy világít-e.
     *
     * @return A jelérték.
     */
    Signal getState() const;

    /**
     * @brief Itt igazából haszontalan, lámpának nincs végezni valója.
     *
     */
    virtual void executeFunction();

    /**
     * @brief Virtuális destruktor az öröklés miatt.
     *
     */
    virtual ~Lamp();
```

Globális függvények:

```
/**
 * @brief Kiírja a kimeneti streamre a lámpa jelének értékét.
 *
 * @param os A kimeneti stream.
 * @param x A kiírt lámpa.
 * @return A kimeneti stream-re referencia, láncolás miatt.
 */
std::ostream& operator<<(std::ostream& os, const Lamp& x);
```

UML ábra:


InputComponent
inputPinCount : size_t
inputNodeIDs : size_t
+ InputComponent(in inputCount : size_t)
+ resetForSimulation() : void
+ activateIfReady() : void
+ getInputPinByIndex(in idx : size_t) : InputPin
+ setInputNodeID(in at : size_t, in ID : size_t) : void
+ isConnectedToNodeOnInput(in ID : size_t) : bool
+ printConnectedInputNodes(inout os : std::ostream) : void
+ executeFunction() : void
+ ~InputComponent()
Lamp
+ Lamp()
+ getState() : Signal
+ executeFunction() : void
+ ~Lamp()

Gate és kapu osztályok

Ebben a szekcióban a kapu osztályokat közösen mutatnám be, mivel felépítés szerint szinte azonos osztályok, egyetlen különbség, hogy más a logikai funkció amit megvalósítanak, de nagyon hasonló struktúrát mutatnak a tagjai.

Minden kapu, kivéve a NOT és WIRE-t (melyek fixen 1-1 be és kimenettel rendelkeznek), bármilyen mennyiségű, minimum 2 lábbal rendelkezhet. Ennek fő oka, hogy nem jelent komoly implementációs problémát, nem szükséges megkötni, elég szabad a modell ehhez.

(Mellesleg közösen (Gates.h/cpp file-okban) vannak implementálva pontosan emiatt.)

Az egyik furcsa jelenség, amit majd észrevehetünk a negált kapuknál (pl. NAND), az, hogy kicsit fura módon a nem negált párjukból származtatjuk (pl. NAND-et AND-ből).

Ennek az az érdekes oka van, hogy valójában ugyanazt a folymatot szervezik le, mint a negálatlan párjaik, csak a végén megfordítják az alkotott kimeneti jeleket. Ennek folyamán jött a gondolat, miszerint jó ötlet lenne származtatni egymásból, és újrahasználni a logikai számolást végző függvényeket, majd a kimenetet megfordítani.

(Öröklés logikája alapján ez kicsit fura lehet, nincs “az-egy” kapcsolat, de a kevésbé elegáns másolást így tudjuk elkerülni.)

Gate

A kapukra jellemző közös viselkedést implementálja, teljesen absztrakt osztály, az öröklési hierarchiában vesz csak részt. Minden kapunak egy kimenete van, és a funkciója valamilyen logikai művelet elvégzése, majd a kimenetén kiküldeni ez a jelet.

Tagváltozók:

Saját nincsen, csak amit örököl IOComponent-től.

Védett tagfüggvények:

```
/**
 * @brief Végrehajtja a logikai műveletet amit specifikus kapunak kell
          végeznie.
 *
 */
virtual void performLogicCalculation() = 0;
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza a kaput.
 *
 * @param inputCount A bemenetek száma.
 * @param outputCount A kimenetek száma.
 */
Gate(size_t inputCount, size_t outputCount);

/**
 * @brief Vérgehajtja a kapu funckióját, azaz a bemeneti jelekből a belső
          logika alapján előállítja kimeneti jeleket, majd ezeket tovább
          küldi.
 *
 */
void executeFunction();

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~Gate();
```

AND

Az AND kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief AND operációt végrehajtja a bemeneti pin-ek jelein, és beállítja a
          kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az AND kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
AND(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~AND();
```

OR

A OR kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief  OR operációt végrehajtja a bemeneti pin-ek jelein, és beállítja a
           kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az OR kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
OR(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~OR();
```

XOR

A XOR kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief XOR operációt végrehajtja a bemeneti pin-ek jelein, és beállítja
          a kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az XOR kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
XOR(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~XOR();
```

NOT

A NOT kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief  NOT operációt végrehajtja a bemeneti pin-ek jelein, és beállítja
           a kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az NOT kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
NOT(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~NOT();
```

NAND

Az NAND kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief  NAND operációt végrehajtja a bemeneti pin-ek jelein, és beállítja
           a kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az NAND kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
NAND(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~NAND();
```

NOR

A NOR kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief NOR operációt végrehajtja a bemeneti pin-ek jelein, és beállítja
          a kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az NOR kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
NOR(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~NOR();
```

XNOR

A XNOR kaput implementáló osztály.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

Védett tagfüggvények:

```
/**
 * @brief XNOR operációt végrehajtja a bemeneti pin-ek jelein, és beállítja
          a kimenetet.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az XNOR kaput.
 *
 * @param inputCount A kapu bemeneteinek száma.
 */
XNOR(size_t inputCount);

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~XNOR();
```

WIRE

A vezetéket implementáló osztály. Azért itt van implementálva, mert minden kapunak vettük a negáltját, és a NOT kapunak csak egy “semmit sem csináló” vezeték a logikus ellentéte. Itt volt a legkönnyebb implementálni.

Tagváltozók:

Saját nincsen, csak amit Gate-től örökölt.

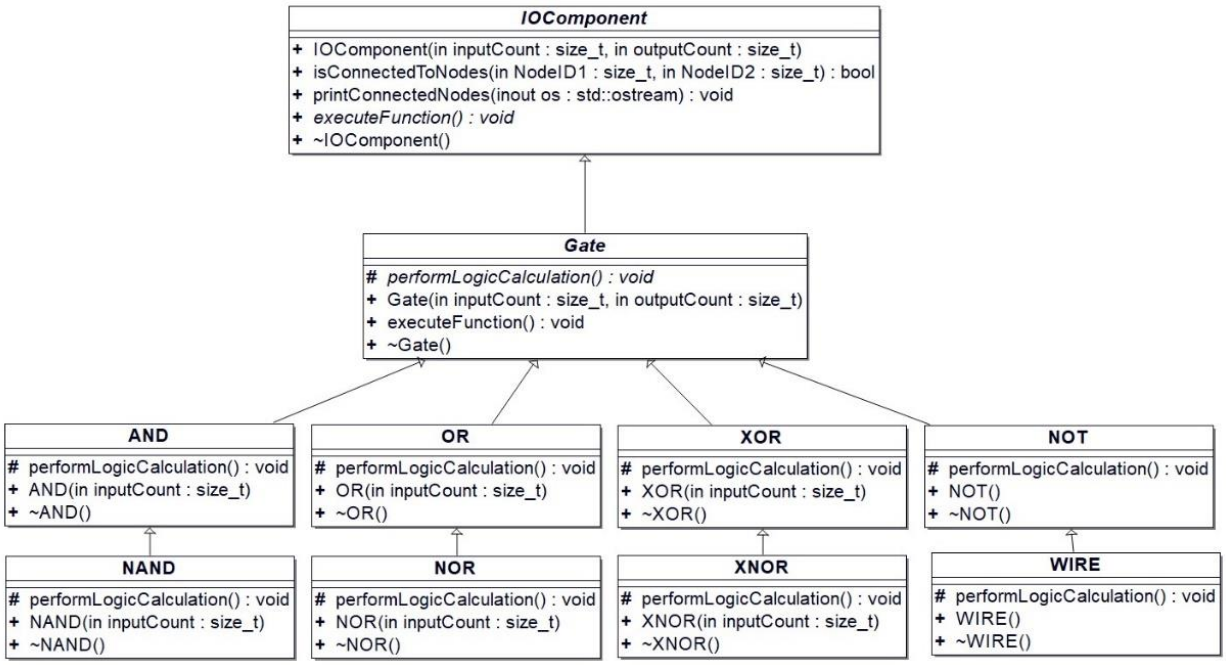
Védett tagfüggvények:

```
/**
 * @brief  Vezetéket jelvezetését hajtja végre, nem csinál semmit a jellel.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

```
/**
 * @brief Létrehozza az vezetéket.
 *
 */
WIRE();

/**
 * @brief Virtuális destruktor az öröklés miatt.
 *
 */
virtual ~WIRE();
```

Közös UML ábra



Circuit

6. Tesztelés (NEM VÉGLEGES)

Tesztelés menete

Az osztályok helyes működésének a tesztelést alapvetően a gtest_lite és a memtrace-en keresztül valósítom meg, ezeken keresztül ellenőrizzük, hogy minden a terv szerint működik.

A tesztelés kitalálásakor a fő cél az volt, hogy a Circuit osztály interface-én keresztül a működés sima legyen, hiszen ez a fő terméke a feladatunknak, ezt tudja a felhasználó hasznosítani.

Pontosan emiatt a többi osztálynak nincsenek is saját tesztjei, hiszen ezek működése garantált, amennyiben helyesen működik a Circuit osztály is, felesleges lenne külön tesztelni azokat is.

A megadott tesztesetek mellett lehetőség van egy szimpla menüvel vezérelt tesztelőre is, melyen keresztül saját file-okat is be tudunk tölteni, illetve saját bemenettel le is tudunk szimulálni.

Tesztesetek

A tesztesteket csoportosítjuk aszerint, hogy hogyan ellenőrzik a Circuit osztály helyes működését:

- SANITY:
  - Ezek az alapvető tesztek, melyek a nagyon alapvető funciókat ellenőrző műveleteket tartalmaznak (konstruktorok, másolás, konfig file beállítás, hiba stream beállítása)
- COMPONENT_CHECK:
  - Ezek az egyes áramköri elemek rendeltetésszerű működését ellenőrzik (első a kapukra, többi a periféria elemek működésére fókuszál)
- ERRORS:
  - Ezek az egyes futásidejű hibák helyes kezelését ellenőrzik (helytelen szintaxis, rövidzár, szimulálatlan elemek, stb.)
- COMPLEX_CIRCUITS:
  - Ezek komplexebb áramkörök, melyekkel bemutatjuk az implementált típusok együttműködésével kialakítható funkcionális elemeket (pl. multiplexer, dekóder, komparátor, stb.)