

# Digitális áramkör szimulátor

NHF Dokumentáció

Programozás alapjai 2.

Pálinkás Lőrinc Mihály - XB0SMF

# Tartalom

1.	Feladat.....	4
	Digitális áramkör .....	4
2.	Feladatspecifikáció.....	5
	Feladat általános leírása .....	5
	Megvalósított áramköri elemek .....	5
	Bemenet formátuma .....	5
	Kimenet opciók .....	6
3.	Pontosított specifikáció (kiegészítés).....	7
	Áramköri elemek I/O pin száma .....	7
	Felhasználói felület .....	7
4.	Terv.....	9
	Információ áramlása .....	9
	Adatáramlásos működés problémái és megoldások .....	10
	Objektummodell .....	12
	Queue osztály .....	12
	Signal osztály .....	13
	Pin osztályok .....	13
	Component osztályok .....	15
	Node osztály .....	18
	Gate osztályok .....	19
	Periféria jellegű osztályok .....	20
	Circuit osztály .....	20
	Teljes ábra .....	23
5.	Megvalósítás.....	24
	Változtatások, bővítések .....	24
	Kommentek .....	24
	Saját kivételek .....	24
	Queue iterátor .....	25
	Circuit interface-nek bővítése .....	25

Component interface bővítése .....	25
Pin-ek elérésének módosítása .....	26
Programozói dokumentáció .....	27
A programozói dokumentációról röviden .....	27
A módosításokkal kapcsolatban .....	27
Signal .....	28
Pin .....	30
InputPin .....	32
OutputPin .....	34
Component .....	36
InputComponent .....	38
OutputComponent .....	41
IOComponent .....	44
Node .....	46
Source .....	48
Switch .....	50
Lamp .....	52
Gate és kapu osztályok .....	54
Queue .....	65
Exceptions .....	69
Circuit .....	73
Menu .....	81
6.    Tesztelés.....	83
Tesztelés menete .....	83
Tesztesetek .....	84
Memóriakezelés tesztje .....	85
7.    Felhasználói dokumentáció.....	86
Program használata .....	86
Szintaxis .....	86

# 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.

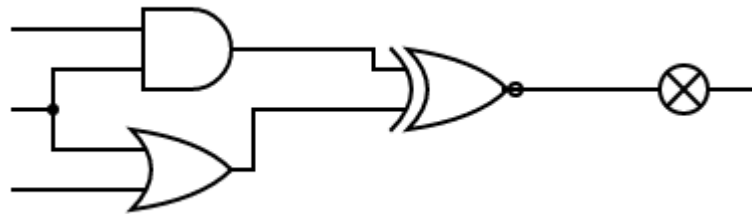
### Bemenet formátuma

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 csomóponttra 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 (kiegészítés)

### specifikáció

#### 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)	$\geq 1$ (tetszőlegesen sok helyre küldhet jelet)
Vezeték	1 (csomópont speciális esete)	1 (csomópont speciális esete, amikor 1 kimenet van)
Kapu	$\geq 1$ (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 kiértékelhetők, 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 kiértékelhetők. 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 ezekre adok megoldásokat.

### 1. Elszigeteket, 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 szembeűnő 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 csomópont-ra 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 értékeljük 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, amíg hasonlóan funkcionálnak jelen esetben a forráshoz (stabil a kimenet, nincs instabil visszacsatolás). Ez azonban azt jelenti, hogy instabil visszacsatolás kezdetekor egy stabil jelforrást kapó csomópont-ra küldünk vissza ellentétes jelet (mert instabil).

Valóságban elképzelve ez hasonlóan funkcionál 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.

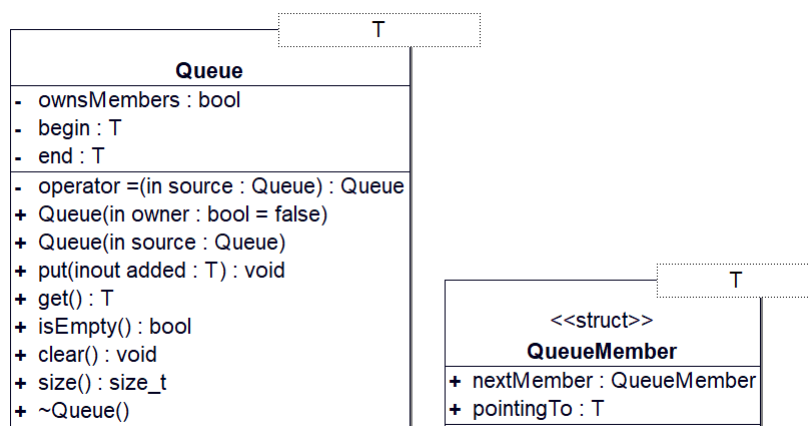
## Objektummodell

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.

Ahogy 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” attribútumot. 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 tartalmaz, 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:

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

Funkcionalitás szempontjából elég egyszerű osztály, létre tudunk hozni vele jelet, beállítani és kiolvasni, megfordítani és összehasonlítani. A jeleket boolean értéként 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: egyrészt jelet tárolnak, melyet ki lehet olvasni, másrészt jelet adnak át a másik pin-nek.

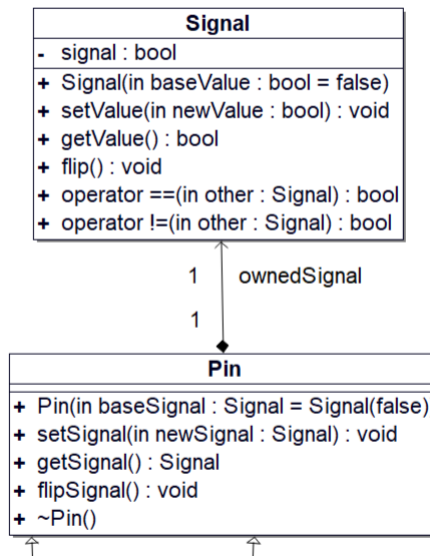
Tervezés elején az tűnt célszerűnek, hogy egyfajta `Pin` osztály létezzen, viszont hamar egyértelmű lett, hogy nem elég, az információ áramlásának modellje miatt szükséges volt két részre bontani.

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 funkcionálisokhoz szükséges műveletek teljesen másak, más információt szükséges tárolni.

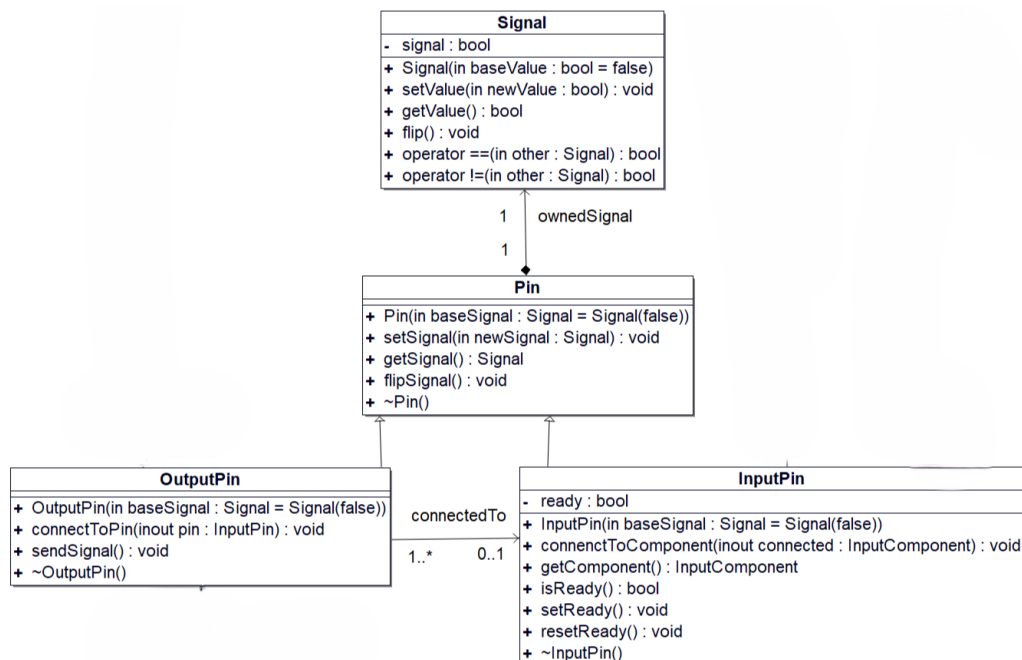
Emiatt döntöttem úgy, hogy bár fog létezni egy közös őszosztály, melyben a közös funkcionális van megvalósítva, azonban két külön osztályként célszerűbb megvalósítani őket.

A sima Pin őszosztály 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 destruktork, 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

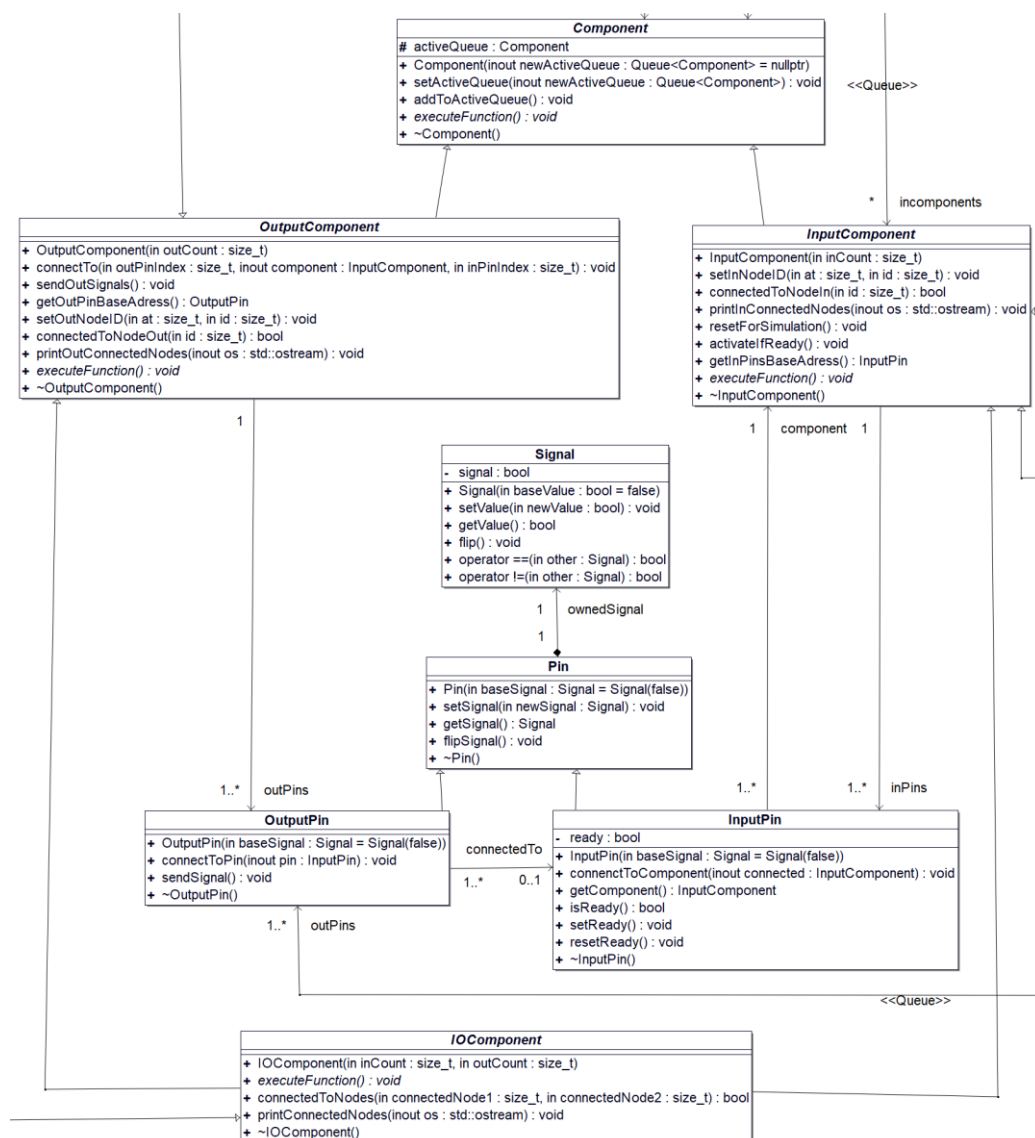
töltenek be. Elsőre célszerű lehet, hogy minden elemnek legyen ki és bemenete, és vegyük 0-nak, ha nincs ilyen. Csak ezzel a problémába ütközünk, hogy sok elemre értelmetlen műveletek lesznek értelmezve, jobbnak tűnt, ha minden elemre választhatjuk hogy melyik funkciót tölti be, ha esetleg mindkettőt,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

Ezáltal hoztam létre az InputComponent és OutputComponent interfész osztályokat, melyeken keresztül egy áramköri elem megkaphatja vagy egyik vagy másik funkciót, 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 közbelső 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 kiíráskor 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 legyegeyszerűbbnek. 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.

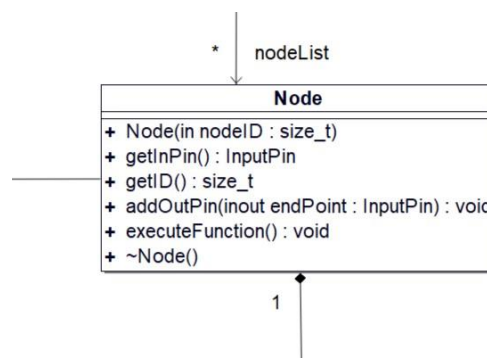
Az `IComponent` csak köztes class, nem ad extra funkciót (többszörösen örököl a másik kettőtől).

## 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).

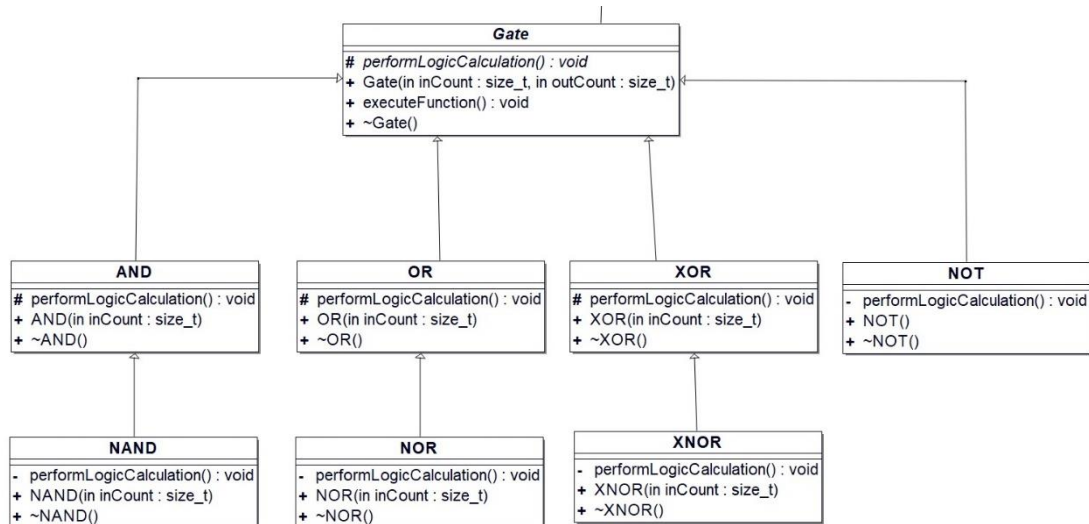
Ennek az eredménye az alábbi osztály („- id: size\_t” most nem látszik, mert a felhasznált UML program reverse enginbeer funkciójá ezt nem adta hozzá):



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

## Gate osztályok

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



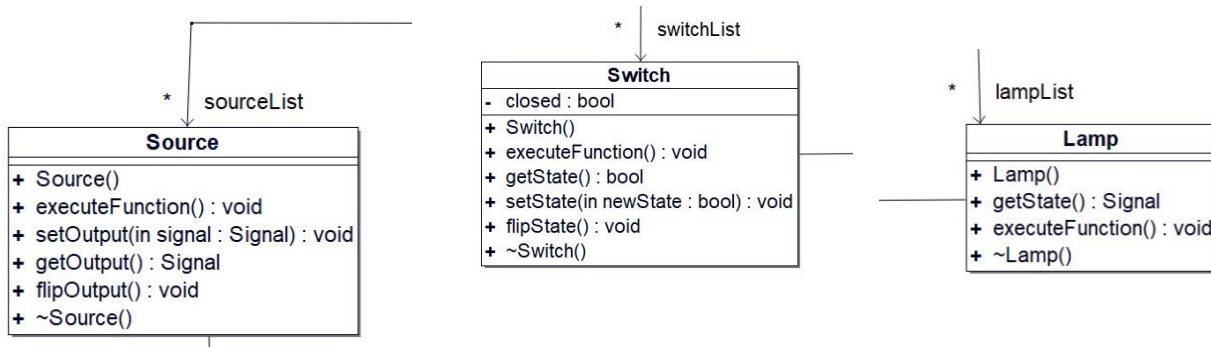
A Gate osztály főleg a többi IComponent-től szolgál elkülönítésként, csak köztes, absztrakt osztály szerepe van.

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 funkcionálnak, 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 szimmetrikus, 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.

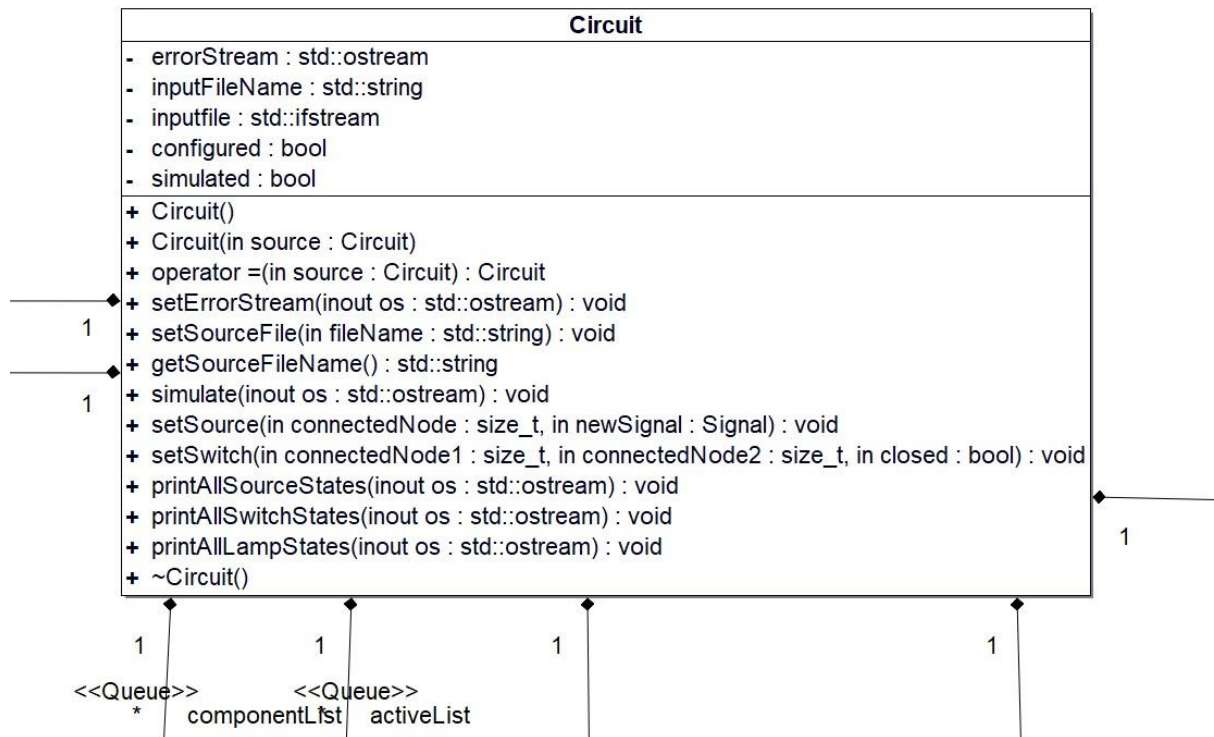
A lámpa csak az egyetlen bemeneti Pin-jén fogadja, más funckiója nincs. Az állapotot itt is le tudjuk kérdezni.

Mindhárom osztályhoz tartoznak inserter-ek is, melyekkel ki lehet írni 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 átírányítani (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 konfiguráció és szimulálás kikerüléséhez)
- Szimuláláskor megadjuk hogy hova irányítsa a kimenetet
- Be tudjuk állítani bizonyos csomópontokra kapcsolt források/kapcsolók állapotát (jelzi, ha nincs megadott)
- Ki lehet vele írni az egyes periféria elemek adatait (lesz ehhez is inserter operátor, amivel tudja a szimulátor kiírni)

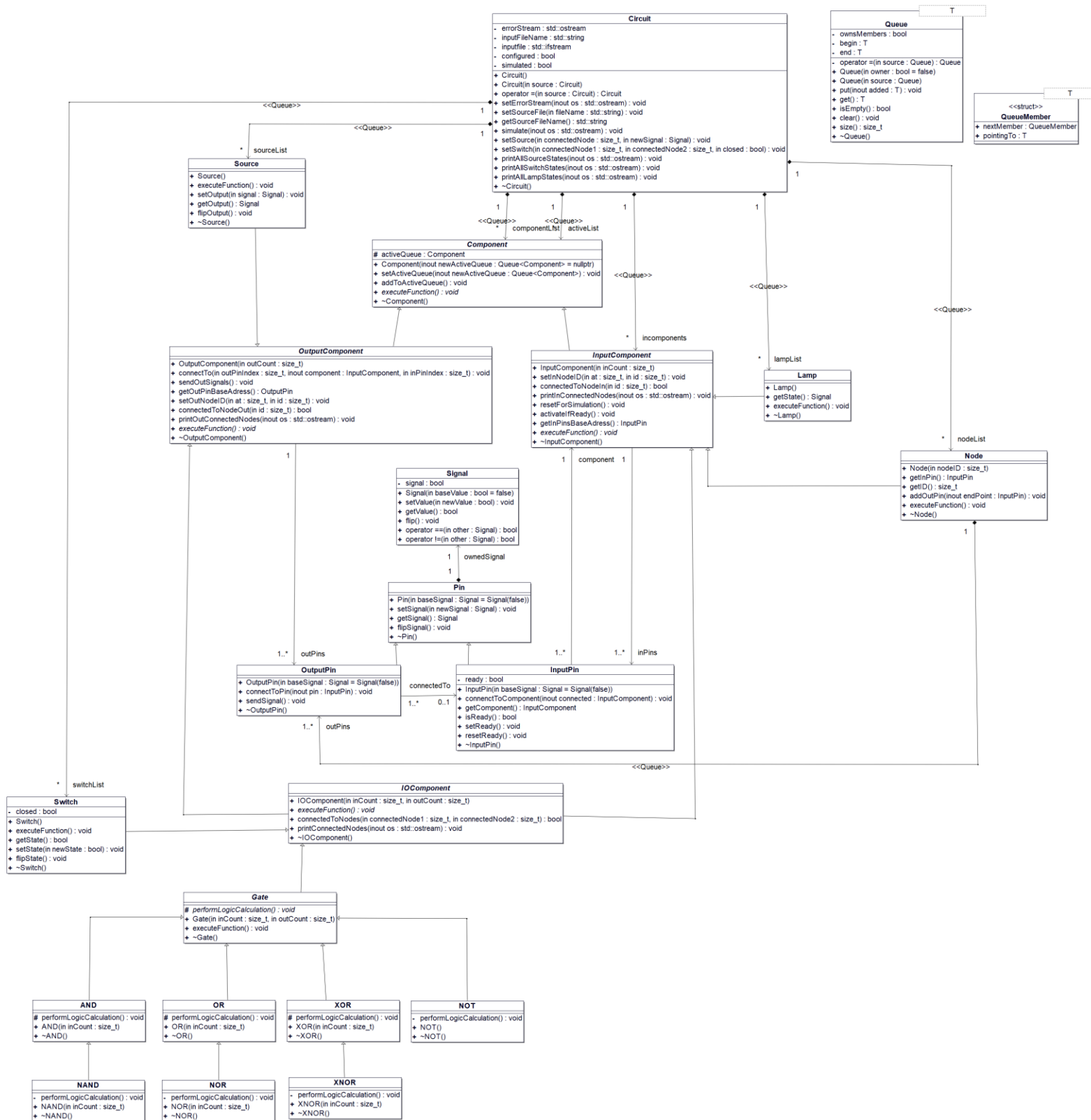
Alap konstruktorja üresen hozza létre. Felkonfigurálás automatán történik (általában első szimuláció során), de csak akkor konfigurál fel, ha szükséges, feleslegesen nem konfigurál újra.

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



## 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, funkcionális haszna valójában nincsen. Mivel nem esszenciális a feladathoz, emiatt itt úgy ítéltem meg, hogy itt értelmes megemlíteni, mivel nincs meghatározó a funkcionális 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 találkozunk 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 kiírni 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 felkonfiguráláskor az, hogy nincsen-e elszigetelt elem-e, avagy nincs helytelen működéshez vezető, leszimulálatlan elemcsoport. Emiatt kell futtatni egy ellenőrző tesztet felkonfiguráláskor. 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 függvénnyel 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 még egyszer 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:

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

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 használni lehet a jövőben, ezért nincsen tiltva).

Tagváltozók:

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

Publikus tagfüggvények:

```
/**
 * @brief Létrehoz egy pin-t kezdő jelértékkal.
 *
 * @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;

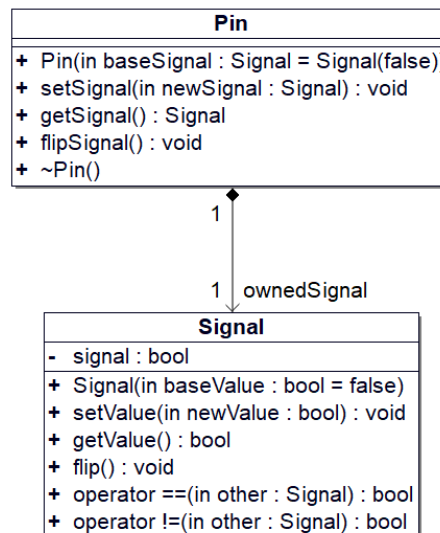
/**
 * @brief Megfordítja a pin jelét.
 *
 */
void flipSignal();
```

```

/**
 * @brief Virtuális a destruktork 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 connectToComponent(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();

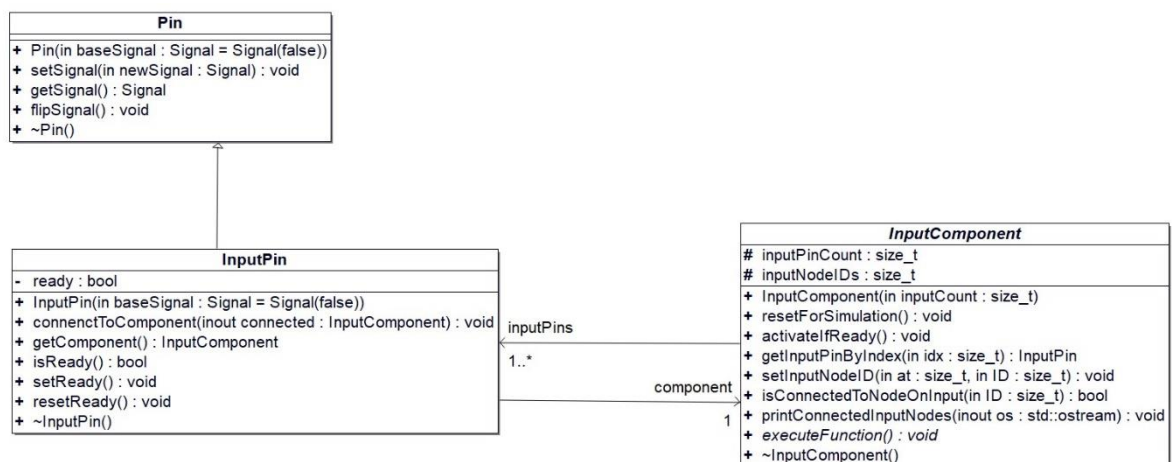
/**
 * @brief Jelzi az áramköri elem felé, hogy ezen a lábán rendelkezésre áll
 *        a jel.
 *
 * @exception NonExistentConnection = Nincs kapcsolt InputComponent eleme.
 */
void setReady();

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

/**
 * @brief Virtuális a destruktork 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:

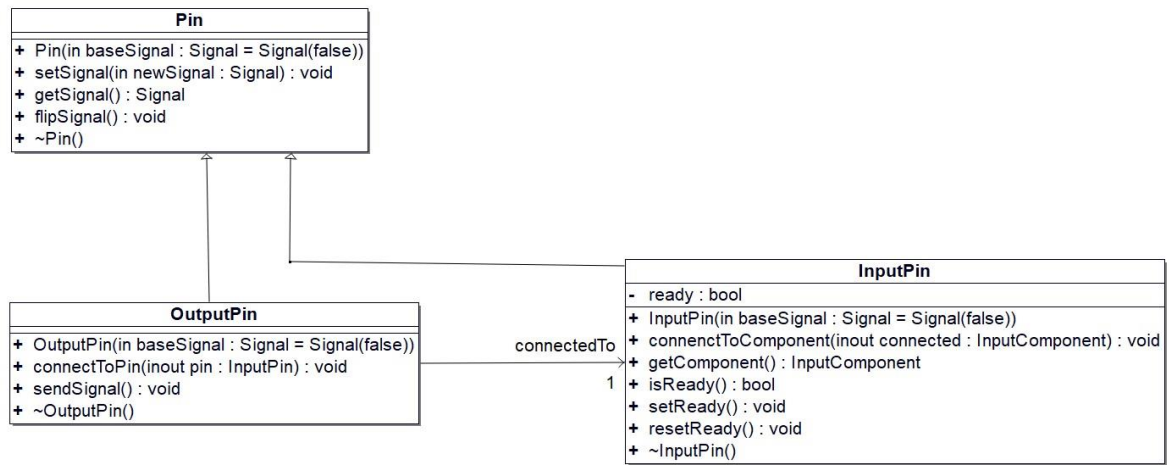
```
/**
 * @brief Létrehoz egy kimeneti pin-t, kapcsolt bemeneti pin-jét NULL-ra
        állítva.
 *
 * @param baseSignal Az alapjel, ha van megadva.
 */
OutputPin(Signal baseSignal = Signal(false));

/**
 * @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 destruktork az öröklés miatt.
 *
 */
virtual ~OutputPin();
```

UML ábra:



## 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 őson 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:

<b>Component</b>
# activeQueue : Component # simulated : bool
+ Component(inout newActiveQueue : Queue<Component> = nullptr) + setActiveQueue(inout newActiveQueue : Queue<Component>) : void + addToActiveQueue() : void + executeFunction() : void + simulated() : bool + setSimulated() : void + resetSimulted() : void + ~Component()

## 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 csomópont 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 bemeneten.
 *
 * @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;

```

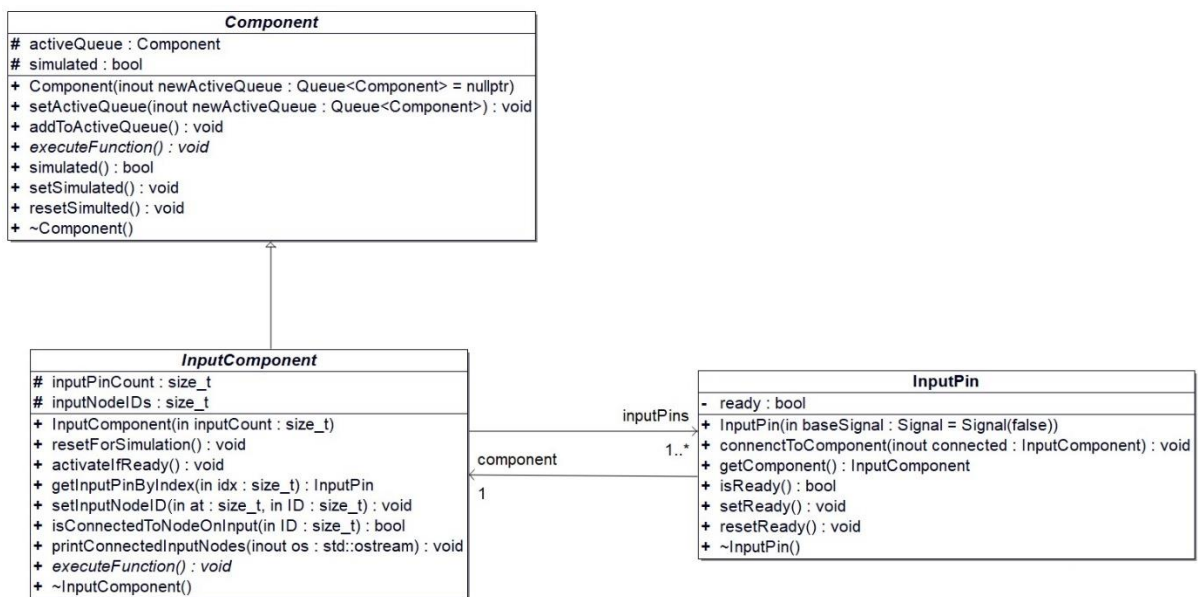
```

/**
 * @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 funkcionalításban 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);

```

```

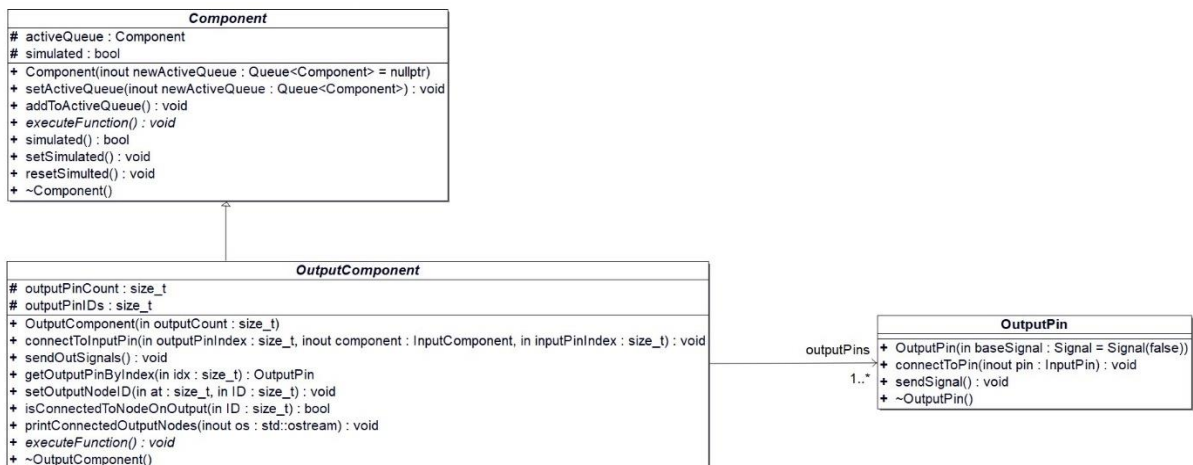
/**
 * @brief Kiírja a kimeneten csatlakoztatott csomópontok ID-ját.
 *
 * @param os A kimeneti stream, ahova akarjuk kiírni.
 */
void printConnectedOutputNodes(std::ostream& os) const;

/**
 * @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

Ahogy 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;

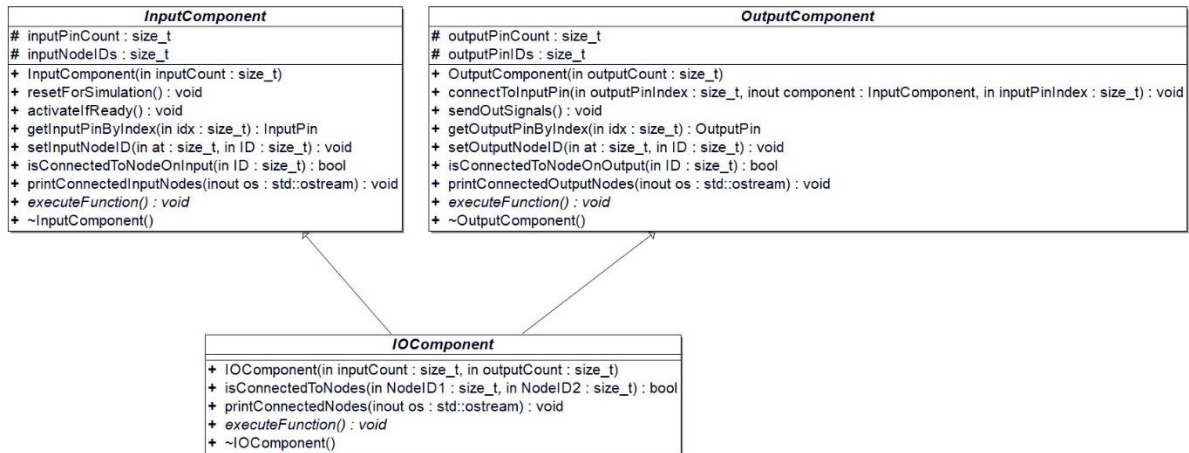
/**
 * @brief Végrehajtja a funkcióját az áramköri elemnek. Leszármazottban
 *        konkretizálva.
 *
 */
virtual void executeFunction() = 0;
```

```

/**
 * @brief Virtuális destruktork, 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 legyünk 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();

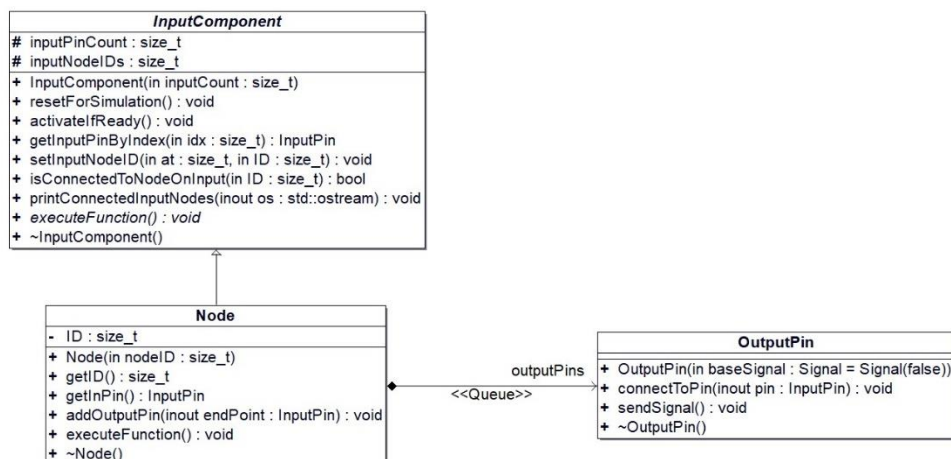
/**
 * @brief Hozzáad a kimeneti pin-ek láncolt listájához egy új pin-t, a
        kívánt végponttal.
 *
 * @param endPoint Ahova küldi majd a jelet a csomópont ezen a kimeneti
        pin-en.
 */
void addOutputPin(InputPin* endPoint);

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

/**
 * @brief Virtuális destruktork 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 destruktorköré miatt.
 *
 */
virtual ~Source() {}

```

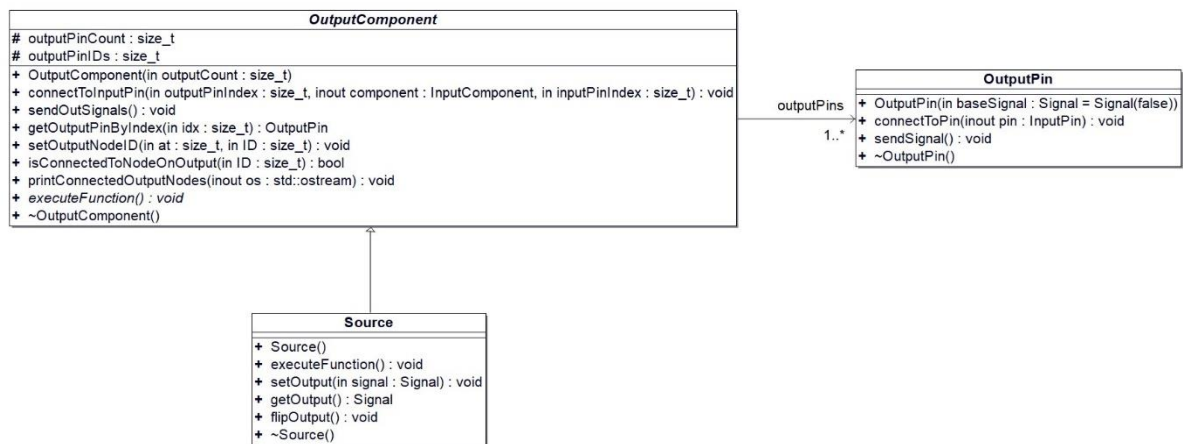
Globális függvények:

```

/**
 * @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 destruktork az öröklés miatt.
 *
 */
virtual ~Switch();

```

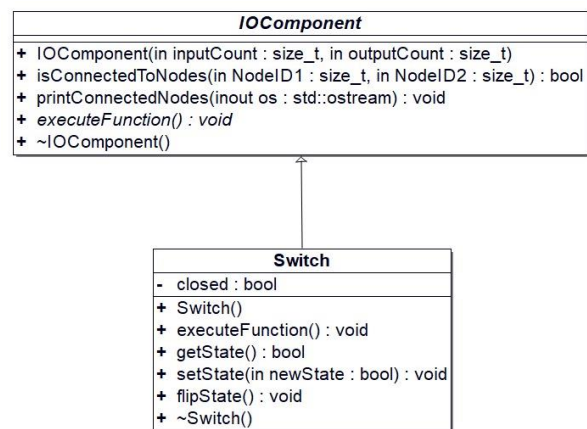
Globális függvények:

```

/**
 * @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;

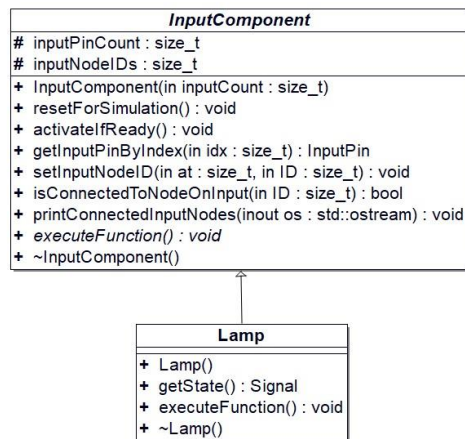
/**
 * @brief Itt igazából haszontalan, lámpának nincs végezni valója.
 *
 */
virtual void executeFunction();

/**
 * @brief Virtuális destruktork 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:



## 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 folyamatot 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 Végrehajtja a kapu funkcióját, 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” vezetéket a logikus ellentéte. Itt volt a legkönnyebb implementálni.

Tagváltozók:

*Saját nincsen, csak amit Gate-től örökölt.*

Védett tagfüggvények:

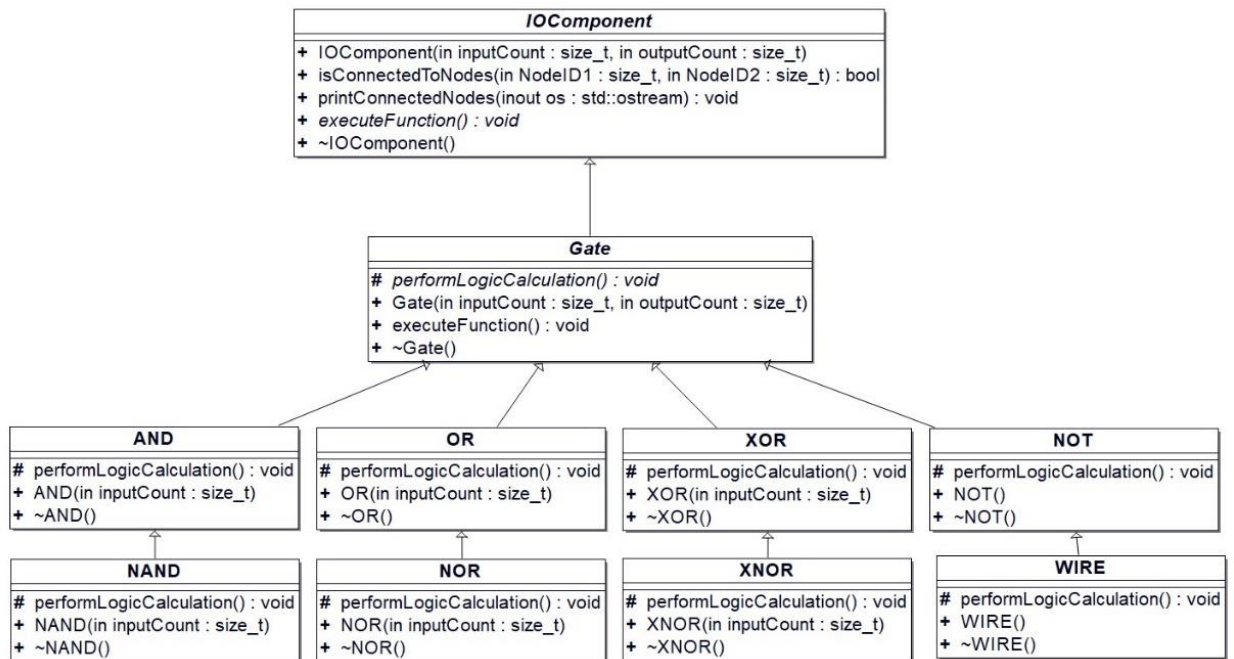
```
/**
 * @brief Vezetéket jelvezetését hajtja végre, nem csinál semmit a jellel.
 *
 */
virtual void performLogicCalculation();
```

Publikus tagfüggvények:

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

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

## Közös UML ábra





## Queue

Ez az osztály specifikusan a feladathoz készült, egy FIFO-t megvalósító osztály, csak saját cél szerint kitalált funkciókkal bővítve.

A tervben működését már részleteztük, ezért ezt itt nem ismételném meg, nem történt eltérés, csak iterátorral bővült a funkcionalitása, illetve kapott egy kiíró függvényt is.

A használatnál kapcsolatban fontos, hogy pointereket tárol, tehát minden kivett adat egy mutató. A felszabadításért a tervben említett "gazda-rendszer" felelős, de kivett memóriát nekünk kell kezelni.

Ezen túl STL-es List/Deque-hez hasonló, néhány helyen korlátozott funkciókkal rendelkezik. (pl. iterátor)

Fontos, hogy csak másolni lehet, de ekkor csak shallow copy készül, hiszen azonos memóriaterületre fog mutatni, emiatt ekkor sosem lesz tulajdonos az újonnan készült FIFO.

Belsejében egy QueueMember struktúra segítségével alkotja a láncot, ennek a deklarációja:

```
struct QueueMember {  
    /**  
     * @brief A következő elem címe a FIFO-ban.  
     */  
    QueueMember* nextMember;  
  
    /**  
     * @brief A mutatott elem.  
     */  
    ComponentType* storedPointer;  
  
    /**  
     * @brief Default konstruktor, ami NULL-ra állítja a tárolt pointert, amit  
     * lehet átállítani.  
     */  
    QueueMember() : storedPointer(nullptr) {}  
};
```

Emellett az iterátor osztályt is tartalmaz, ennek elérése azonban a megszokott módon publikus. Itt nem mutatom be, de a szokásos implementációja és metódusai vannak, megtalálható ez a forráskódban.

Tagváltozók:

```
/**
 * @brief Birtokolja-e az elemeit, azaz destruktorban fel kell-e
 *        szabadítani a tárolt tagokat is a FIFO tárolóival.
 *        (true = igen, false = nem)
 */
bool ownsMembers;

/**
 * @brief A FIFO elejére mutató. (kiszedés és felszabadításhoz)
 */
QueueMember* first;

/**
 * @brief A FIFO végére mutató. (könnyű beszúráshoz)
 */
QueueMember* last;

/**
 * @brief A tárolt elemek száma, könnyű méret olvasáshoz.
 */
size_t siz;
```

Publikus tagfüggvények:

```
/**
 * @brief Beállítja az üres FIFO állapotát.
 */
* @param owner Tulaja-e az elemeknek, azaz fel kell majd szabadítani a
 *        mutatott objektumokat-e.
 */
Queue(bool owner = false);

/**
 * @brief Másolást teszi lehetővé, hogy ideiglenesen valamit tudjunk
 *        futtatni a FIFO-n, a tagokat nem birtokolja.
 */
* @param source A másolás forrása.
 */
Queue(const Queue<ComponentType>& source);

/**
 * @brief Berak egy tagot a sor végére.
 */
* @param added A hozzáadott tag.
 */
void put(ComponentType* added);
```

```

/**
 * @brief Kivesz a sor elejéről egy tagot.
 *
 * @return T* A soron következő tag.
 */
ComponentType* get();

/**
 * @brief Megmondja, hogy üres-e a FIFO.
 *
 * @return true = üres a FIFO ;
 * @return false = nem üres a FIFO
 */
bool isEmpty();

/**
 * @brief Kiüríti a FIFO-t, ha tulaj, akkor törli a memóriát is.
 */
void clear();

/**
 * @brief Visszaadja a FIFO elemszámát.
 *
 * @return size_t A FIFO mérete.
 */
size_t size() const;

/**
 * @brief Detruktor, mely felszabadítja a tárolt elemeket, attól függően,
    hogy birtokolja-e őket.
 */
~Queue();

/**
 * @brief A FIFO elejére ad iterátort.
 *
 * @return iterator Az elejére iterátor.
 */
iterator begin() const;

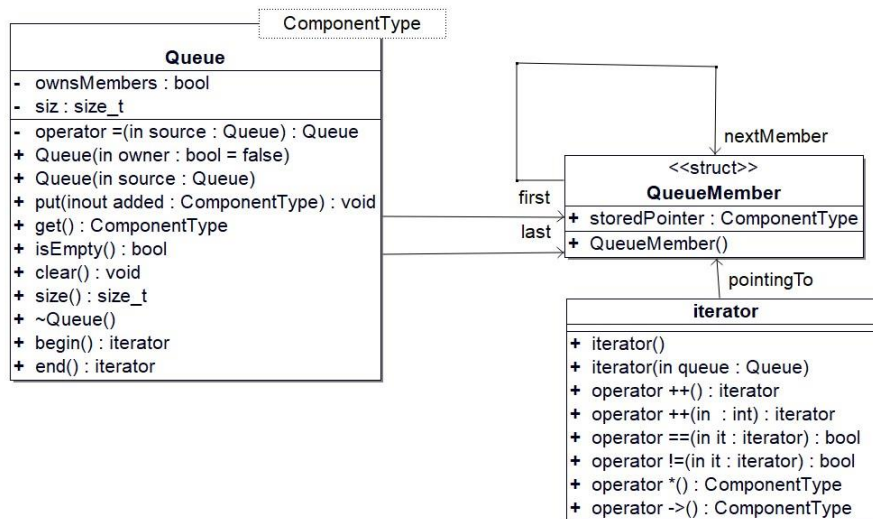
/**
 * @brief A FIFO végére ad iterátort.
 *
 * @return iterator A végére iterátor.
 */
iterator end() const;

```

Globális függvényei:

```
/**
 * @brief Kiírja egy kimeneti stream-re a list összes elemét. (Feltételezett,
 *        hogy létezik << operátor a ComponentType típusra)
 *
 * @tparam ComponentType A listában tárolt elemek típusa.
 * @param printed A kiírt lista.
 * @param os A kimeneti stream.
 */
template<class ComponentType>
void printQueue(const Queue<ComponentType>& printed, std::ostream& os);
```

UML ábra:



## Exceptions

Ahogy a bővítések részben is már említettem, a kivételkezelés kifinomítására saját kivételosztályokat adtam a modellhez, hogy ezek jobban tükrözzék a program struktúráját.

A következőekben ezeket röviden mutatnám be, mivel a legtöbbjük viszonylag egyszerű, emiatt nem szentelek minden egyesnek külön szekciót, itt lehet megtalálni az összes leírását.

Mindegyik a `std::exception`-ből származik, emiatt azzal bármelyik elkapható.

```
/**
 * @brief Üzenetet hordozó kivételek őssosztálya.
 *
 */
class MessageException : public std::exception {
    /**
     * @brief A hordozott üzenet.
     *
     */
    std::string message;
public:
    /**
     * @brief Konstruktor, átadható üzenettel.
     *
     * @param msg Az átadott üzenet.
     */
    MessageException(const std::string& msg);
    /**
     * @brief Az üzenete adja vissza, amit tárol.
     *
     * @return std::string Az üzenet.
     */
    std::string exception_message();
};
```

Azon programban használt kivételek, melyeket a felhasználónak el is kell kapnia, mert a `Circuit` osztály dobhatja:

```
/**
 * @brief Jelölvasás és állítás során dobja az áramkör, ha nincs a megfelelő
 *        csomópontokra kapcsolódó elem. Ezt dobhatja a felhasználó felé.
 *
 */
class MatchingComponentNotFound : public MessageException {
public:
    MatchingComponentNotFound(const std::string& msg);
};
```

```

/**
 * @brief Konfiguráció során történt hiba, ezt dobhatja a felhasználó felé.
 *
 */
class ConfigurationError : public MessagingException {
public:
    ConfigurationError(const std::string msg);
};

```

Azon kivételek, melyek a belső működésben játszanak szerepet (nem a Circuit osztályban, mert más osztályok is használják):

```

/**
 * @brief Kimeneti vagy bemeneti pin a szükséges kapcsolat hiánya esetén
        dobja.
 *
 */
class NonExistentConnection : public MessagingException {
public:
    NonExistentConnection(const std::string& msg) : MessagingException(msg) {}
};

```

```

/**
 * @brief Rövidzár esetén dobja egy kimeneti pin. El van tárolva, hogy melyik
        áramköri elemet próbált újraszimulálni.
 *
 */
class ShortCircuit : public MessagingException {
    InputComponent* resimulated;
public:
    ShortCircuit(const std::string& msg, InputComponent* resimmed);
    InputComponent* getResimulated();
};

```

```

/**
 * @brief A file-hoz nem tudott hozzáférni a program (nem létezik, stb.)
 *
 */
class FileAccessError : public MessagingException {
public:
    FileAccessError(const std::string& msg) : MessagingException(msg) {}
};

```

```

/**
 * @brief Az áramkör felkonfigurálásakor nem elfogadott sortípust tartalmaz a
 * konfigurációs file.
 *
 */
class NonExistentLineType : public MessageedException {
public:
    NonExistentLineType(const std::string& msg) : MessageedException(msg) {}
};

/**
 * @brief A sor komment, csak ennek jelzésére van. Lekezelése = ignorálás és
 * kövi sorra váltás.
 *
 */
class CommentLine : public MessageedException {
public:
    CommentLine(const std::string& msg) : MessageedException(msg) {}
};

/**
 * @brief Hibás szintaxisa a konfigurációs file-nak valahol.
 *
 */
class IncorrectSyntax : public MessageedException {
public:
    IncorrectSyntax(const std::string& msg) : MessageedException(msg) {}
};

/**
 * @brief Hibás számú csomópont mennyiséget adott meg a konfigurációs file-ban
 * egy áramköri elemhez.
 *
 */
class IncorrectPinCount : public MessageedException {
public:
    IncorrectPinCount(const std::string& msg) : MessageedException(msg) {}
};

/**
 * @brief Nem Node-ot dobott a rövidzár (elméletileg nem kéne ilyet dobnia).
 *
 */
class PointerConversionError : public MessageedException {
public:
    PointerConversionError(const std::string& msg) : MessageedException(msg) {}
};

```

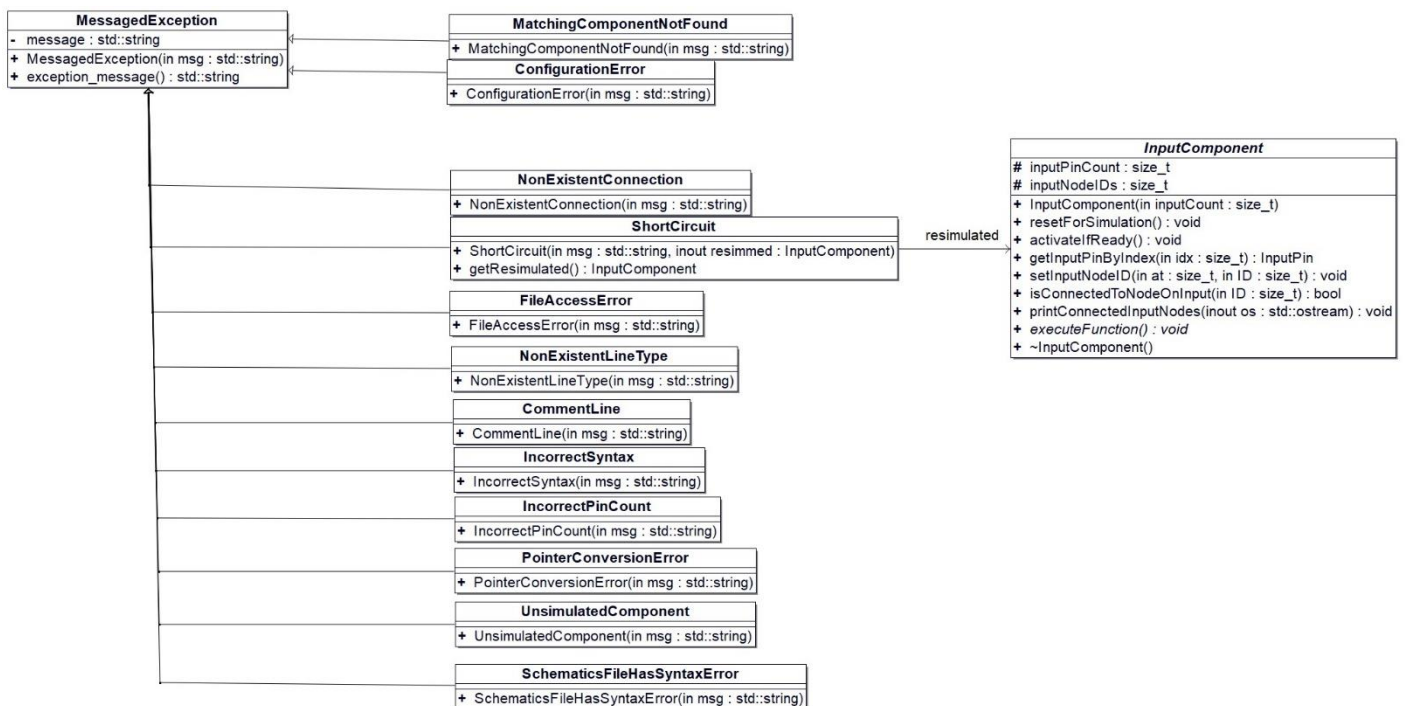
```

/**
 * @brief Szimulálatlan elem van az áramkörben, valahol szakadás van benne.
 *
 */
class UnsimulatedComponent : public MessageedException {
public:
    UnsimulatedComponent(const std::string& msg) : MessageedException(msg) {}
};

/**
 * @brief Szintaktikai error jelzésének továbbítására szolgál.
 *
 */
class SchematicsFileHasSyntaxError : public MessageedException {
public:
    SchematicsFileHasSyntaxError(const std::string msg) :
    MessageedException(msg) {}
};

```

UML ábra:





## Circuit

A feladat szempontjából ez a legfontosabb osztály, ez felelős a megfelelő interakciót biztosítani az áramkörrel. Ő végzi a felkonfigurálás egyes lépéseit.

A tervhez képest a bővítések szekcióban mutatott funkciókkal bővült csak az interface-e, ennek is a tesztelés volt a fő oka, illetve lehetőség adhat áramkörök összekapcsolására is esetleg.

Tagváltozókat három részre tudjuk osztani. Az első csoport kezeli a külvilággal a kommunikációt, ezek tárolják a megfelelő stream-eket: (A sima kimeneti stream nincs megadva, mert ezt a szimulációnak kell megadni, mivel nem tűnt célszerűnek ezt belül tárolni.)

```
/**
 * @brief A hiba stream, ahova ki írjuk az esetleges hibákat (pointer, nem
 *        referencia, mert változtatható).
 *
 */
std::ostream* errorStream;
/**
 * @brief A felhasznált bemeneti file helye.
 *
 */
std::string inputFilePath;
/**
 * @brief A bemeneti file, ahonnan olvassuk felkonfiguráláskor.
 *
 */
std::ifstream inputfile;
```

A második csoport a Queue tárolók, melyekkel tudja számontartani az egyes áramköri elemeket, illetve különféle interface-eken keresztül elérni őket:

```
/**
 * @brief Összes áramkör elem listája, felszabadításkor ezen keresztül
 *        biztosítja, hogy minden memóriát töröl;
 *
 */
Queue<Component> componentList;
/**
 * @brief A bemeneti pin-el rendelkező pin-ek listája. Reseteléshez kell,
 *        ezen keresztül tudjuk az összes bemeneti pin-jeit resetelni.
 *
 */
Queue<InputComponent> inputComponentList;
```

```

/**
 * @brief Az aktív FIFO, amiből szimuláció során kiértékejük elemeket.
 *        (Azért kell itt, mert be kell az elemekre állítani az aktív FIFO-t)
 *
 */
Queue<Component> activeList;
/**
 * @brief A jelforrások listája, kiolvasáshoz és beállításhoz.
 *
 */
Queue<Source> sourceList;
/**
 * @brief A kapcsolók listája, kiolvasáshoz és beállításhoz.
 *
 */
Queue<Switch> switchList;
/**
 * @brief A lámpák listája, kiolvasáshoz.
 *
 */
Queue<Lamp> lampList;
/**
 * @brief A csomópontok listája, melyeken keresztül kapcsolódnak az egyes
 *        áramköri elemek.
 *
 */
Queue<Node> nodeList;

```

A harmadik csoport boole változók, melyekkel tudja elkerülni a felesleges újrakonfigurálást és újraszimulálást, ezeket szimulációk és konfigurálások előtt ellenőrzik:

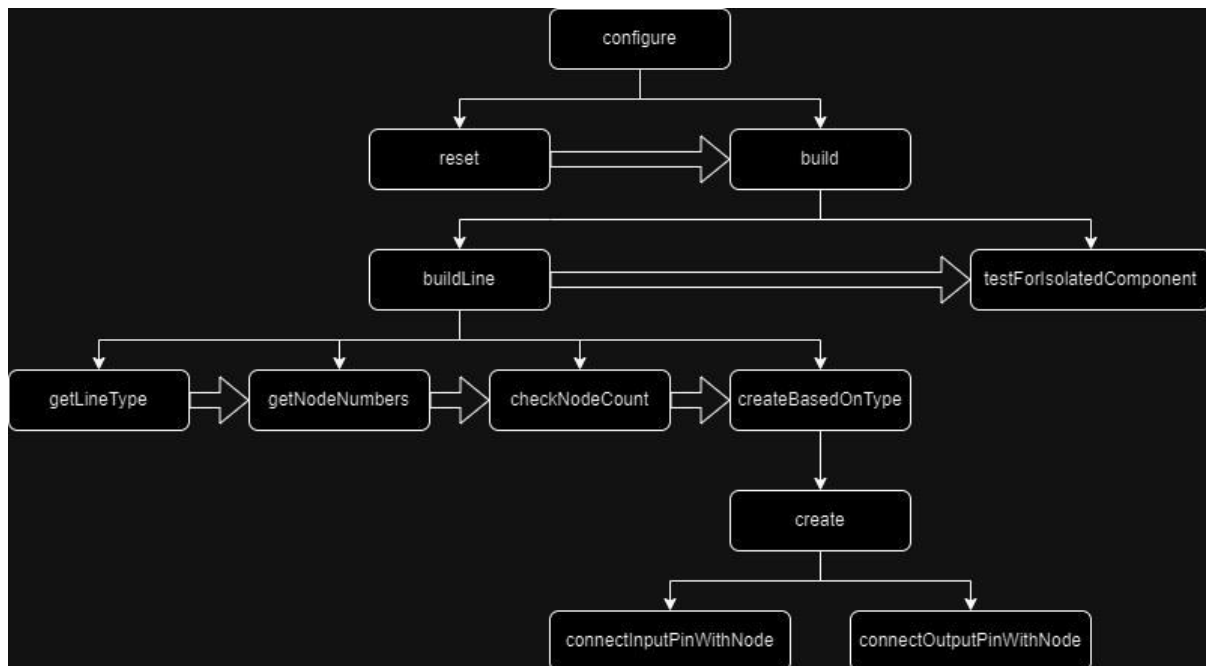
```

/**
 * @brief Ez a változó jelzi, hogy fel van-e konfigurálva az áramkör. (Ha
 *        igen, akkor nem konfigurál újra feleslegesen.)
 *
 */
bool configured;
/**
 * @brief Ez jelzi, hogy le lett-e szimulálva az áramkör. (Ha igen, akkor
 *        nem szimuláljuk újra feleslegesen.)
 *
 */
bool simulated;

```

A belső függvények szinte teljes mértékben a konfigurációt kezelik. Ezekből számos van, főleg a dekompozíció érdekében, és könnyen el lehet bennük veszni, emiatt itt szeretném bemutatni érthetően a

működésüket. Ennek érdekében az alábbi ábrával szemléltetném a konfigurációs függvényhívásokat, mert ez jól szemlélteti a lefolyását:



Itt a függőleges vonalak a függvényhívásokat, a vízszintes vastag vonalak a sorrendet jelölik, ha van.

Ki tudjuk tehát olvasni, hogy körülbelül alábbi módon konfigurál:

- Először reseteli áramkörben tárolt információt, töröl minden korábbi elemet és adatot.
- Ezután soronként haladva építi fel.
  - Kiszedi a sor típusát, majd minden ()-es egységre kiszedi a csomópontok számait.
  - Leellenőrzi, hogy ez helyes mennyiség (létezik ilyen pin darabszámmal rendelkező elem, csak ennyi szemantikai ellenőrzés van), majd típusa alapján létrehozza őt.
  - Ezt ismétli, amíg talál ()-es egységet.
- A legvégén, ahogy a tervben is láttuk, futtat egy ellenőrzést, hogy nincs-e szimulálatlan elem-e benne.

Ha bárhol hibába fut (pl. szintaktikai hiba), akkor bár végig feldolgozza a file-t (ha hozzá tudott férni), de lesz a folyamat végén egy ConfigurationError dobva, és a konfiguráció sikertelen lesz.

A konkrét függvény deklarációkat itt nem részletezném, mivel főleg az interface-re fókuszálok, ezt is csak a belső működés megértése miatt fejtettem ki. (A belső működésben részt vesz egy privát struktúra és egy enum, de ezeket itt nem fejtem ki szintén, mert a kódban megtalálható a dokumentációjuk, bár egyik sem bonyolult konstrukció.)

Publikus tagfüggvények:

```
/**
 * @brief Létrehoz egy üres áramkört, amit fel tudunk konfigurálni.
 */
Circuit();

/**
 * @brief Másolással hoz létre egy áramkört egy másikkól.
 * @attention Feltételezzük, hogy nem változott a forrás file-ja a
             másoltnak!
 *
 * @param source A másolt áramköri elem.
 */
Circuit(const Circuit& source);

/**
 * @brief Átmásol egy áramkört egy másikba.
 * @attention Feltételezzük, hogy nem változott a forrás file-ja a
             másoltnak!
 *
 * @param source Ahonnan másolunk.
 * @return Circuit& A láncolás miatt kell csak magára referenciát
             visszaadnunk.
 */
Circuit& operator=(const Circuit& source);

/**
 * @brief Beállítja az hiba stream-et, ami lehet vagy std::cerr vagy file
        vagy bármilyen ostream objektum.
 *
 * @param os Ahová fogja kijelezni a hibaüzeneteket.
 */
void setErrorMessageStream(std::ostream& os);

/**
 * @brief Beállítja forrás file-t. (Nem konfigurálja fel!)
 *
 * @param path A forrás file helye.
 */
void setSchematicFile(const std::string& path);

/**
 * @brief Visszaadja a forrás file nevét (esetlegesen a helyével, ha úgy
        lett megadva).
 *
 * @return const std::string& A forrás file neve.
 */
const std::string& getSourceFileName() const;
```

```

/**
 * @brief Leszimulálja az áramkört, és az eredményeket kiírja a megadott
        kimeneti stream-re.
 *
 * @param os A kimeneti stream, ahová írjuk.
 *
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
void simulate(std::ostream& os);

/**
 * @brief Beállítja egy forrás jelszintjét a megadott értékre, amennyiben
        létezik. (Ha nem, akkor hiba stream-re jelzi ezt.)
 *
 * @param connectedNode A csomópont, amihez a forrás kapcsolódik.
 * @param newSignal A beállítani kívánt jelszint.
 *
 * @exception MatchingComponentNotFound = ha nincs az adott csomóponthoz
        kapcsolódó forrás.
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
void setSource(size_t connectedNode, Signal newSignal);

/**
 * @brief Beállítja egy kapcsoló állapotát, amennyiben létezik. (Ha nem,
        akkor hiba stream-re jelzi ezt.)
 *
 * @param connectedNode1 Az egyik csomópont, amihez kapcsolódik a kapcsoló.
 * @param connectedNode2 A másik csomópont, amihez kapcsolódik a kapcsoló.
 * @param closed Az új állapota. (true = zárt, false = nyitott)
 *
 * @exception MatchingComponentNotFound = ha nincs az adott csomóponthoz
        kapcsolódó forrás.
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
void setSwitch(size_t connectedNode1, size_t connectedNode2, bool closed);

```

```

/**
 * @brief Visszaadja egy megadott csomópontra csatlakozó forrás által
        kiadott jelszintet.
 *
 * @param connectedNode A csomópont, amihez kapcsolódik a forrás.
 * @return Signal A jelszint, amit kiad a forrás.
 *
 * @exception MatchingComponentNotFound = ha nincs az adott csomóponthoz
        kapcsolódó forrás.
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
Signal getSourceSignal(size_t connectedNode);

/**
 * @brief Visszaadja két megadott csomópontra csatlakozó kapcsoló állapotát.
 *
 * @param connectedNode1 Az egyik csomópont, amire kell csatlakoznia.
 * @param connectedNode2 A másik csomópont, amire kell csatlakoznia.
 * @return true = zárt a kapcsoló ;
 * @return false = nyitott a kapcsoló
 *
 * @exception MatchingComponentNotFound = ha nincs az adott
        csomópontokhoz kapcsolódó kapcsoló.
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
bool isSwitchClosed(size_t connectedNode1, size_t connectedNode2);

/**
 * @brief Visszaadja egy megadott csomópontra csatlakozó lámpa állapotát,
        azaz a tárolt állapotát.
 *
 * @param connectedNode A csomópont, amihez kapcsolódik a lámpa.
 * @return Signal A jelszint, amivel a lámpa rendelkezik.
 *
 * @exception MatchingComponentNotFound = ha nincs az adott
        csomópontokhoz kapcsolódó kapcsoló.
 * @exception ConfigurationError = felkonfiguráláskor volt probléma
        errorstream-re kiírja, hogy milyen baj történt.
 */
Signal getLampSignal(size_t connectedNode);

```

```

/**
 * @brief Kiírja egy kimeneti stream-re az összes forrás jeleit.
 *
 * @param os A kimeneti stream.
 */
void printAllSourceStates(std::ostream& os) const;

/**
 * @brief Kiírja egy kimeneti stream-re az összes kapcsoló státuszát.
 *
 * @param os A kimeneti stream.
 */
void printAllSwitchStates(std::ostream& os) const;

/**
 * @brief Kiírja egy kimeneti stream-re az összes lámpa állapotát.
 *
 * @param os A kimeneti stream.
 */
void printAllLampStates(std::ostream& os) const;

/**
 * @brief Törli az objektumot, minden memóriával együtt, illetve bezárja a
 *        használt forrás file-t.
 *
 */
~Circuit();

```

Globális függvények: (főleg segéd funkciókat töltenek be, vagy inserterek)

```

/**
 * @brief Kiírja a kimeneti stream-re az áramkör összes periféria elemének
 *        (forrás, kapcsoló, lámpa) állapotát, illetve a forrás file nevét
 *        és a kiolvasás idejét is.
 *
 * @param os A kimeneti stream, ahova írunk.
 * @param circuit Az áramkör amit kiírunk.
 * @return std::ostream& Referencia a kimeneti stream-re a láncoláshoz.
 */
std::ostream& operator<<(std::ostream& os, const Circuit& circuit);

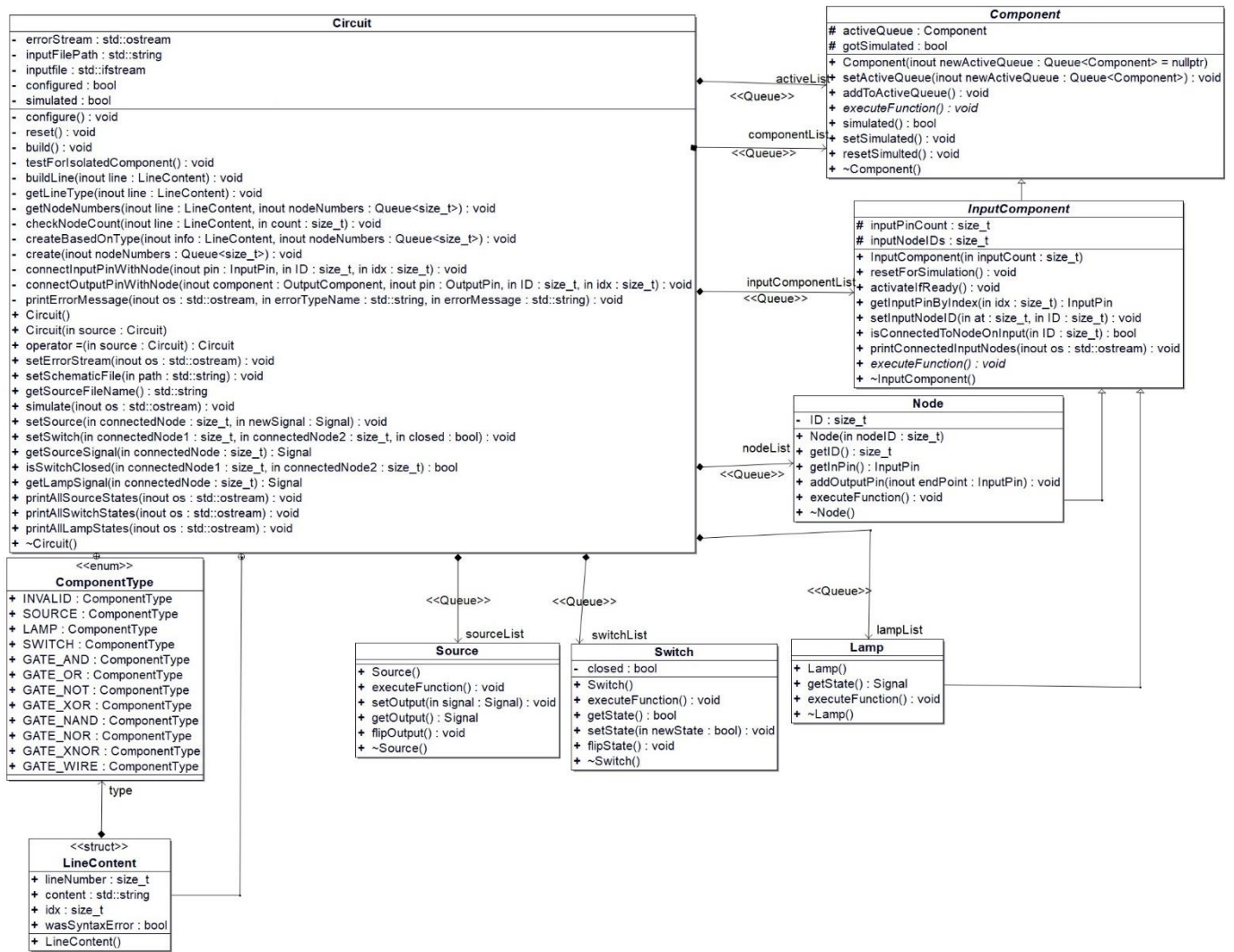
/**
 * @brief Kiír egy elválasztóvonalat egy kimeneti stream-re.
 *
 * @param os A kimeneti stream.
 * @param c A karakter amiből épül az elválasztóvonal.
 * @param times Ailyen hosszú az elválasztóvonal.

```

```
*/  
void printSeparatorLine(std::ostream& os, char c, int times);  
  
/**  
 * @brief String-é konvertál egy unsigned intigert.  
 *  
 * @param converted A konvertálandó szám.  
 * @return std::string A szám, string-ként reprezentálva.  
 */  
std::string size_tToString(size_t converted);
```



## UML ábra:



App

## 6. Tesztelés

### Tesztelés menete

Az osztályok helyes működésének a tesztelést alapvetően a `gtest_lite` segítségével 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. (pl. építeni egy grafikus felületet rá.)

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, meg bizonyos esetekben körülményesebb is lenne.

A megadott tesztesetek mellett szimpla menüvel vezérelt programmal is kipróbálhatjuk a működést, melyen keresztül saját file-okat is be tudunk tölteni, és saját bemeneti kombinációval tudunk szimulálni. Ehhez a felhasználói dokumentáció az utolsó szekcióban található.

## Tesztesetek

A tesztesetekben az egyes API funkciókat teszteljük a Circuit osztálynak.

A teszteseteket csoportosítjuk aszerint, hogy hogyan ellenőrzi a Circuit osztály helyes működését:

- SANITY:
  - Ezek az alapvető tesztek, melyek a nagyon alapvető funkciót ellenőrző műveleteket tartalmazznak (konstruktorok, másolás, konfigurációs 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.)

## Memóriakezelés tesztje

A program és osztályok memóriaszivárgásának ellenőrzése a memtrace segítségével történik, ez garantálja hogy a végső termékünkben nincsen semmilyen fajta memóriaszivárgás.

A teszteseteket vagy a menüs programot futtatva sem tapasztalunk bármilyen fajta memória szivárgást, ezért valóban nincs helytelenül kezelve a memória.

## 7. Felhasználói dokumentáció

Program használata

Szintaxis