Összefoglalás

A mai világban a mindennapi életünk alapvető feltétele az elektromos áram. Az autóiparban egyre nagyobb teret nyer, mint hajtóanyag, otthonaikat már el sem tudnánk képzelni lámpa, mosógép, wifi, TV és egyéb háztartási eszközök nélkül. Ezért nagyon fontos, hogy hogyan és mennyi áramot termelünk.

Az elektromos áramot előállítás szempontjából két csoportba szokás sorolni: megújuló és nem megújuló. Fontos különbség, hogy az előállított áram mennyiségét, milyen pontosan lehet előre meghatározni. Kőolaj és földgáz esetén pontosan ki lehet számolni, hogy adott mennyiségű fűtőanyagból mennyi elektromos áramot tudunk előállítani. Ezzel ellentétben, a szél sebességét és a napsütés mennyiségét nem lehet pontosan megjósolni.

Az energiatermelés fejlettsége mellett viszont, a tárolásra való megoldások le vannak maradva. Ez azért jelent kihívást, mivel a háztartások fogyasztása sem egy konstans érték. Ezen okok, miatt az energiatermelésnek szüksége van egy változó komponensre, amely követni tudja a változó fogyasztást is, ezek jelenleg többségében a gázmotorok, és melléjük csatlakoznak be egyre inkább a megújuló energiaforrások. Ezeknek a termelőknek, hogy mennyit kell termelnie, egy központi szervezet, a MAVIR határozza meg. Ahhoz, hogy ezt a meghatározott célértéket tartani tudják szabályozó központokra van szükség. Mivel ezek fogják tudni feldolgozni és az egyes erőművek között elosztani ezt az értéket.

A szabályozók tesztelése viszont komplex feladat, mivel legegyszerűbben éles környezetben lehetne tesztelni, viszont ez egyértelmű okokból nem a legszerencsésebb. Ezért szakdolgozatom témájaként egy olyan szimulációt valósítottam meg, mely különböző termelő vagy fogyasztó egységek működését másolja le, ez a digitális iker koncepciója. Az általam kialakított tesztkörnyezet célja, hogy segítséget nyújtson a szabályozók későbbi továbbfejlesztése során.

A szakdolgozatban be fogom mutatni a szabályzók működését, a szimulált egységeket ismertetem, valamint bemutatom a megvalósításhoz használt technológiákat, koncepciókat, majd a megoldásomat fogom részletezni és végezetül pár tovább fejlesztési lehetőséget fogok még bemutatni.

# Bevezetés

Napjainkban a fogyasztott áram mennyiségét nem lehet pontosan megjósolni. Példaképpen a háztartásokban nem tudjuk pontosan megmondani, hogy következő nap mennyi lesz a fogyasztás. Emiatt van szükség szabályzókra, melyek dinamikusan változtatják az erőművek energia termelését vagy fogyasztását.

A szakdolgozatom célja, hogy egy ilyen szabályzót lehessen kontrollált környezetben tesztelni egy szimuláción keresztül. A programban lehetőség van a szimuláció befolyásoló különböző tényezők módosítására. Paraméterek megadásával meghatározható, hogy milyen egységek vegyenek részt a szimulációban és ezek milyen konstans értékekkel dolgozzanak.

A második fejezetben a szabályzó feladatát és működését fogom leírni és azt, hogy az általam megvalósított szabályzó milyen logika alapján működik. Ezek mellett bemutatom az egyes szimulált egységek feladatát és működését, mint például az invertert, a terhelőt (loadbank), a gázmotort és az akkumulátort.

A következő szekcióban a szimulációkról fog általánosságban szólni, a digitális iker koncepciójáról és ezek közötti különbségről.

A negyedik fejezetben egy technológiai áttekintő. Kotlin nyelvet választottam a megvalósításhoz ezt fogom majd részletezni, belemerülve a nyelv által nyújtott DSL (Domain Specific Language)-be is. Bemutatom a kiválasztott szimulációs keretrendszert. Ez egy nyílt forráskódú projekt, amely a Kalasim névre hallgat. Technológiák közé fog még tartozni a Gradle, IntelliJ IDEA, Koin és a Kotlin Coroutines.

A következő fejezetben az általam kialakított szimulációt fogom részletezni. Ezt követően pedig értékelem a futási eredményeket, amelyeket diagramok segítségével szemléltetek.

A végén pedig kitérek, hogy milyen fejlődési lehetőségei vannak még a szimulációnak.

# Szabályozó és szimulált egységek

Ebben a fejezetben lesz bemutatni általánosságban a szabályzók feladata működése, és hogy honnét kap parancsot, valamint az irányított egységeket. Ezeket pedig diagramokkal kiegészítve és elmagyarázva. Továbbá még részletezve lesz az általam megvalósított teszt szabályzó is.

## Szabályzó

A szabályzónak alapvetően két oldala van. Az egyik oldalról van a MAVIR, a másik oldalról pedig az egyes napelemparkok, szélturbinák és egyéb erőművek. Ezen két oldal közötti kapcsolatot a szabályzó valósítja meg. Ezt úgy valósítja meg, hogy a kapott célértéket először erőművek között lebontja, majd pedig az egyes egységek között.

### MAVIR

A MAVIR (Magyar Villamosenergia-ipari Átviteli Rendszerirányító) a magyarországi villamosenergia-átvitelhálózat üzemeltetője. Feladatai közé tartozik az energiaellátás biztosítása, valamint a hálózat üzemeltetése karbantartása és fejlesztése1.

A graph showing different colored lines

Description automatically generated

**2.1 Ábra: 11.08-11.23 közötti szükséges árammennyiség2**

A 2.1-es ábra mutatja a MAVIR milyen forrásokból biztosítja az ország energiaellátását. Ezen látható, hogy az országban mennyi volt a fogyasztás. A felső lila réteg az importált mennyiség, ha ezen túl lóg a sárga napelemek által termelt összeg, akkor exportról beszélünk. A többi szín pedig az egyes erőmű típusok alapján van bontva, hogy azok mennyit termeltek. Látható, hogy ebből a napelemparkok a legingadozóbbak.

MAVIR a szabályzással azt akarja elérni, hogy a termelés és a fogyasztás egyenlőek legyenek. Ennek fontossága, hogy a villamosenergia-rendszereknek a frekvenciája állandó legyen. A Magyarországi frekvencia értéke 50 Hz. A termelés és fogyasztás kilengései tudják változtatni, ezt a 2.2-es ábra mutatja be. Annak magyarázata, hogy miért szükséges az 50Hz, túlmutat ezen a szakdolgozaton.

A diagram of a power line

Description automatically generated

**2.2 Ábra: A termelés és fogyasztás hatása a frekvenciára4**

A szabályozásnak alapvetően három fő fajtája van. Az első típus az elsődleges tartalék (FCR), ez harminc másodpercen belüli stabilizálásra szolgál. A következő az automatikus kiegyenlítő szabályozási szolgáltatás (aFRR), ezt az elsődleges tartalék után minél hamarabb elindítják, ennek hét és öt percen belül kell, hogy lehessen aktiválni. A szimuláció feladat egy ilyen szabályozó tesztelése. Az utolsó kategória a manuális kiegyenlítő szabályzási szolgáltatás (mFRR), amit pedig tizenkét és fél percen belül lehet aktiválni.4

A graph of different colored triangles

Description automatically generated

**2.3 Ábra: Hogyan aktiválódik egymáshoz képest a FCR, az aFRR és az mFRR 10**

A graph showing a number of green and yellow bars

Description automatically generated with medium confidence

**2.4 Ábra: 11.17-11.18 közötti országos aFRR szabályzás 15 perces bontásban5**

A 2.4-as ábrán az látható, hogy az automatikus kiegyenlítő szabályozási szolgáltatást hányszor használták egy huszonnégy órás periódusban. Ha a nulla felett van akkor fel szabályzásról beszélünk, tehát az országban többletes rendszerállapot lépett fel. Ha pedig nulla alatt van, ennek az ellentétjéről beszélünk, hiányos rendszerállapotról.

### Szabályzó kliensei

A villamosenergia piacon jelenlevő termelő és fogyasztó szereplőknek, előre jelezniük kell az általuk megtermelt, illetve felhasznált energia mennyiségét.

A termelőknek ezt a predikált értéket kell betartaniuk, ha ezt nem tudják megtenni akár büntetés is járhat érte. A szabályzónak az a feladata, hogy ezeket az eltéréseket minimalizálja. Ezenfelül a hatékonyabb energia elosztását szolgálja, a fogyasztók és termelők különböző mérlegkörökbe vannak besorolása.

A mérlegkör a kiegyenlítő energia igénybevételének okozathelyes megállapítására és elszámolására és a kapcsolódó feladatok végrehajtására a vonatkozó felelősségi viszonyok szabályozása érdekében létrehozott, egy vagy több tagból álló elszámolási szerveződés3.

### Szabályzó feladata

Most, hogy a szabályzó két oldalát ismerjük, ahonnét kapja a bemenetei parancsait, valamint ahova továbbítja a kimeneti szabályzásokat. Így könnyebb lesz megérteni, pontosan, hogy is működik.

Az erőművek alapvetően csoportban vannak és ezek közösen vannak szabályozva. Ezek a csoportok az alapján alakulnak ki, hogy az adott erőmű kit kér fel, hogy elvégezze számára a szabályozást. Ilyen aggregátor fél például a cég, ahol ezt a külsős témát is csinálom. Feladatuk, hogy MAVIR-nál bejelentsék a mérlegkörüket, melybe az erőművek lesznek.

A szabályozás két szinten történik, az első rétegben meghatározásra kerül az egyes erőművekre eső termelési kvóta. A következő réteg pedig, már a parkon belül termelő és fogyasztó egységek szintjén fogja meghatározni, hogy az egy-egy elemnek mennyit kell termelnie. Ezen értékeket olvassa a szabályzó, aminek hatására az első szinten visszacsatolást kap ezzel pontosítva a szabályozást.

## Szimulált egységek

Ebben a fejezetben részletezem a szimulációban megvalósított egységek működését és feladatai.

### Inverter

A rendszerben az inverter felelőssége az előállított energia átalakítása olyan elektromos árammá, amely a magyar villamoshálózat szabványának megfelelő.

A napelemek alacsony feszültségű egyenáramot generálnak, mely nem megfelelő az átlagos fogyasztó eszközök számára. Az inverter két lépcsőben transzformálja az érkező energiát. Először egy DC/DC konverter emelt feszültségű egyenáramot állít elő, majd második lépésben egy DC/AC konverter 50 Hz váltakozó áramot generál. Ezt már közvetlenül betáplálható a közműhálózatba.6.

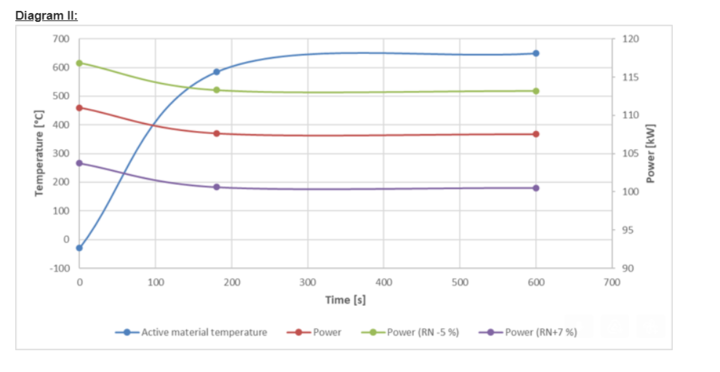


**2.5 Ábra: Napelem melletti inverter7**

### Terhelő (Loadbank)

A terhelőegység egy olyan eszköz, amelyet alapvetően arra terveztek, hogy elektromos terhelést biztosítson a generátorok, akkumulátorok és feszültségmentesítő tápegységek teljesítményének teszteléséhez és validálásához. Fő célja egy elektromos terhelés szimulálása, mivel ennek segítségével precízen beállítható terhelési értékek mellett vizsgálhatóak a korábban felsorolt eszközök viselkedése.

A szabályzás szempontjából, amennyiben nincs szükség a termelő által előállított teljes villamos energiára, akkor mint fogyasztó lehet használni a hálózatra betáplált energia szabályzására. A felesleges energia a terhelő ellenállásain hőként írható le.



**2.6 Ábra: Terhelők ellenállásának nagysága hőmérséklet függvényében**

A 2.6- os ábrán látható a terhelők ellenállása a hőmérséklettel arányosan. Hideg induláskor láthatóan, nagyobb a felvett teljesítmény és ekkor kisebb a generált ellenállás is. Felmelegedés után tud beállni konstans áram felvételre, hőmérsékletre és generált ellenállásra. Ennek következménye, hogy a terhelően belüli komponensek nem indulhatnak egyszerre, mivel túl nagy lenne a felvett teljesítmény. A minél hitelesebb szimuláció elérése érdekében, ezen karakterisztikus jellemzőket is implementálnom kellett a megoldásom során.

### Gázmotor

A gázmotor termelő szerepet tölt be, működési alapja egy belsőégésű motor, amely gáz halmazállapotú szénhidrogén tüzelőanyaggal működik.



**2.7 Ábra: Gázmotor**

A gázmotor a szimuláció szempontjából két karakterisztikus jellemzője is van, amelyeket figyelembe kellett vennem. Az első ilyen, hogy csak akkor indulhat el, ha egy bizonyos százalék fölött fog termelni. A másik pedig, hogy a motornak van felmelegedési ideje, ameddig nem termel áramot.

### Akkumulátor

Az akkumulátor különlegessége, hogy mindkét szerepet fel tudja venni, tud fogyasztó és termelő egység is lenni. Így még az előzőekben bemutatott terhelő csak leszabályzáskor használhatóak, addig ezek az egységek a korábban eltárolt energia segítségével felszabályzást is képesek megvalósítani.

## Saját szabályzó

A szimuláció bemutatására és tesztelésére létre kellett hoznom egy saját szabályzót. Felépítésben követi a 2.1.3-as fejezetben ismertetett felépítést, tehát az első réteg felel, hogy az erőművek célértékét meghatározza, a második szint pedig a parkokon belüli egységek termelési és fogyasztási mennyiségért.

A szabályzó, amire a szimuláció tervezve lett egy aFRR szabályzó, aminek egy tulajdonsága, hogy két másodpercenként lehet vele az erőműveknek parancsot küldeni.

### Erőmű szabályzó

A feladat a MAVIR által adott időközönként küldött fel- és leszabályozási parancs szimulálása. Ezen felül itt valósul meg a célértéke parkokra való bontása is.

Ezt jobb karbantarthatóság miatt és áttekinthetőség miatt külön programban implementáltam, amelyben függőségként használtam fel és futtatom a szimulációt. A szabályozó megvalósítása közbe törekedtem, hogy egyszerűen lehessen a szimulált egységeket definiálni. Valamint arra, hogy könnyen meglehessen határozni, hogy melyik időpillanattól, milyen értékkel szabályozzon. Ez nem dátumhoz, hanem a szimuláció kezdete óta eltelt idővel meghatározva.

A szimulációt pedig külön coroutine-ban indítom el, ezzel megoldva, hogy ne blokkolja a főszálat, amin közben két másodpercenként küldöm az éppen beállított értéket és olvasom ki a szimulált adatokat.

Ez egyértelműen egy leegyszerűsített verzió, melyben a lényeg az, hogy változásokra, hogy reagál a rendszer. Valamint jelenleg nincs visszacsatolás a kiolvasott értékre sem.

### Egység szabályzó

Az egység szabályzó feladata, hogy a parkokra bontott szabályzó értékeket, tovább ossza az inverterek, motorok, terhelők és akkumulátorok szintjére. Ez a valóságban az erőművekhez kihelyezett PLC feladata lenne. Erre, most nincs lehetőségem, hogy a PLC logikáját integráljam a rendszerbe, így egy egyszerűsített verziót implementáltam, amely alkalmas az általam létrehozott szimulációs környezet tesztelésére.

Minden egységre külön kellett megvalósítanom a logikát, hiszen ahogy azt a korábbi fejezetekben kifejtettem nagyon eltérő karakterisztikákkal rendelkeznek.

Az első az inverter volt, itt a legegyszerűbb módot választottam. Az egyes egységek maximális kimenete alapján átlagoltam a célértéket és elosztottam az inverterek között.

A következő a terhelők logikája volt. Ez akkor fog történni, ha a szabályozott érték kisebb, mint a jelenlegi célérték. Az első terhelő után, csak akkor indulhat a következő, ha az már megfelelő mértéig felmelegedett. Ugyan ez igaz a többi egységekre is. Az indítási sorrend az alapján dőlt el, hogy melyik a legnagyobb, amit el lehet indítani.

A gázmotorok esetében, a lehető legnagyobb indul el szabályzáskor, viszont hogyha már van termelő egység, akkor annak termelését figyelembe véve fogja indítani a következő legnagyobbat. Tehát ha húsz a szabályzás mértéke, de már egy ötös egység megy, akkor egy tizenötöset fog elindítani.

Az utolsó egység, aminek modelljét bemutatom az akkumulátor. Ezek az elemek dinamikusan képesek változtatni viselkedésüket. Többlet termelés esetén fogyasztóként viselkedik, és tárolja az energiát, még hiány esetén képesek a tárolt energiájukat visszatáplálni a hálózatra, és termelőként viselkedni.

# Szimulációk

Ebben a fejezetben általánosságban fogok írni a szimulációk felhasználásáról és kifejtem pár fajtáját, majd rátérek az általam használt típusra. Ezek után pedig bemutatom a digitális iker koncepcióját.

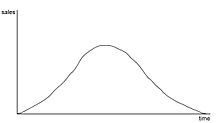
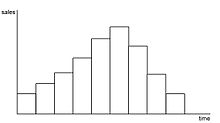
## Szimulációkról általánosságban

A számítógépes szimulációk egyre több és több iparágban nyernek szerepet, gondoljunk csak az egészségügyre, ahol például sebészeti beavatkozások tervezésében tud segíteni. Ezen kívül az autóiparban is alkalmazzák őket a járművek tervezésénél és tesztelésénél. Űr iparra is gondolgatunk, ahol egy szimuláció segítségével le tudják tesztelni, hogy a rakéta sikeresen ki fog-e jutni az űrbe. Esetünket nézve az energiaiparba szabályzó kódunkat tudjuk tesztelni egy olyan teszt környezetben, amelyben semmiféle kárt nem tudunk okozni a valós felhasználóknak.9

A szimuláció alapvetően egy valós világban létező folyamatnak vagy egy rendszernek a mása, melyet meg lehet figyelni és valós adatokkal összehasonlítani. Általában teljesítmény növelésre, optimalizálásra, biztonság fejlesztésre és tesztelésre szokták használni.

A szimulációkat többféleképpen lehet kategorizálni. Egyik besorolási lehetőség a kimenetel kiszámíthatóságán alapuló mód, itt megkülönböztetünk sztochaikus szimulációkat. Sztochaikus szimulációk során konstans paraméterekkel is a végeredmény véletlenszerűen változik. Determinisztikus esetben állandó bemeneti értékek mellett az eredmény minden időpillanatban ugyanaz a marad.

Másik lehetséges megközelítés, hogy milyen módon modellezzük az idő előrehaladását. Ebben a kontextusban két fő kategóriát különböztetünk meg: folytonos és diszkrét szimulációkat. A folytonos szimulációkat általában differenciálegyenletek alkalmazásával vannak modellezve, lehetővé téve a folyamatos, sima időbeli haladást. Ezzel szemben a diszkrét szimulációk időpontokban létrejövő eseményeket kezelik, és a rendszer állapotát csak ezeken a pontokon frissítik.



**2.7 Ábra: Bal oldalon diszkrét, Jobb oldalon folyamatos szimuláció ábrázolása11**

## Digitális iker (Digital Twin) koncepció

A fejezetben áttekintésre kerül a digitális iker koncepciója. Definiálásra kerül a fogalma, fajtái, illetve, hogy hogyan használtam fel a megvalósítás során.12

### Történelme

A digitális iker koncepcióját David Gelernter Mirror Worlds című 1991-es könyve vetítette előre. A könyvében arról beszél, hogy a virtuális környezetek digitálisan tükrözik a valóságos világot, lehetővé téve az interaktív megfigyelését és manipulációját.

A koncepciót először 2002-ben Michael Grieves mutatta be a Society of Manufacturing Engineers konferencián. Ekkor Grieves, mint termékéletciklus-menedzsment modellként javasolta azt.

2010-ig sok néven ismerték, majd ekkor John Vickers a NASA mérnöke Roadmap reportjában „digital twin”-ként hivatkozott rá, ekkortól terjedt el ez a név.

### Digitális iker

A digitális iker egy virtuális reprezentációja egy fizikai rendszernek, amely folyamatosan frissül az aktuális adatokkal. A valóságban megtalálható félt fizikai ikernek is nevezik. Ennek virtuális párja, úgynevezett digitális iker, amelyet a fizikai fél fejlesztésénél, üzemeltetésénél és karbantartásánál használnak.

A digitális iker koncepciónak három fő eleme van. Ebből kettőt már említettem. Az első a fizikai iker, a valós világban felelhető rendszer vagy folyamat. A második a digitális iker, amely az előző szoftver formában. A harmadik pedig az adat, ami összeköti az előző kettőt.

A two airplanes in the sky

Description automatically generated with medium confidence

**2.7 Ábra:Digitális iker elemeinek illusztrációja13**

A digitális ikreknek három fajtája van, amely megmutatja, hogy a koncepciót milyen helyzetekben lehet használni:14

* Digital Twin Prototype - Prototípus (DTP), amelyet akkor használnak mikor még a fizikai termék nem készült el.
* Digital Twin Insctance - Példány (DTI), ezt akkor használják mikor már létezik a termék és azon szeretnének különböző teszteseteket lefuttatni.
* Digital Twin Aggregate - Aggregate (Aggregate - DTA), ez a fajta több DTI-t foglal magában, amelyek adatai és információi felhasználhatóak a fizikai termékkel kapcsolatos lekérdezésekre és prognosztikákra.

Nem csak használat alapján lehet kategóriákra bontani, hanem az alapján is, hogy a rendszernek mekkora részét fogja virtuális térbe átültetni. Ezeknek felsorolás a következő, legkisebb szimulált egységtől az egész rendszerig:15

* Component - Komponens szintű, ezzel a rendszer egy kis részét lehet megfigyelni.
* Asset - Eszköz szintű, amely több komponenst foglal magába.
* System or Unit - Rendszer vagy egység iker, amely több eszköz összevonásával egy működő rendszert lehet létrehozni virtuális térben.
* Process - Folyamat, melyben egész gyárakat tudunk akár létrehozni és megnézni, hogy az egész, hogy fog együtt működni.

A digitális iker technológiáknak vannak tulajdonságaik, amelyek megkülönböztetik, más technológiáktól. A digitális iker koncepciójának előnyei más módszertanokhoz képest:

* Kapcsolódás - a digitális iker a valóság és virtuális világ között kapcsolatot alakít ki. Ez lehet akár két irányú, vagy csak egy irányú. Itt gondolva arra, hogy a létező elem kihatással van a digitálisra és ez visszafele is igaz.
* Adat homogenizáció - Lényege, hogy a valós egységről nagy mennyiségű adat tárolása nehezen megvalósítható, így az ezekkel való munka sem egyszerű feladat. Ezzel szemben viszont a digitális megfelelőjére könnyen lehet nagy mennyiségű adatot tárolni, melyeket közös formára is tudunk hozni, ezzel az adatokat még függetlenítjük is a valós egységtől, ezzel jobban páraméterezhető és tesztelhető lesz.
* Újra Programozhatóság - a fizikai egységet program formájában létrehozzuk, így a módosítás költsége csökken. Ezzel kapcsolatban van a modularitás, amely segít abban, hogy meglehessen figyelni, hogy mely modulok cserélésével a rendszer, hogyan változik.

## Digitál iker és szimulációk összehasonlítása

Habár mind a szimulációk és a digitális ikrek hatékony eszköz egy rendszer bonyolultságának megfigyelésére. Mindkét technológiában digitális modelleket használnak a rendszer funkcióinak replikálására, a fő különbség a méretben rejlik. A szimulációk általában egy folyamatra összpontosítanak, míg a digitális ikrekkel több szimuláció is futtatható a bonyolult műveletek egészének való áttekintés érdekében.

A digitális iker sokkal gazdagabb infromációkészletet biztosít az elemzéshez. A virtuális tér használatával a felhasználók átfogóbb képet kaphatnak, hogy a rendszer hogyan fog viselkedni különböző forgatókönyvek esetén.

Kiemelkedő különbség még, hogy a szimulációkból hiányoznak a valós idejű visszacsatolási adatok a pontosság mérésére, még a digitális iker úgy van megvalósítva, hogy kétirányú kommunikációt folytasson. Ezzel a struktúrával segít pontosabb modellek és válaszok generálásában, lehetővé téve a felhasználók számára, hogy azonnal hozzáférhessenek az intelligenciához.

Konklúzióként a digitális iker is egy szimuláció, tehát tulajdonságaival jellemezhető, viszont összetettebb, ez által komplexebb feladatok elvégzésére alkalmasabb.

## Megvalósítás

Itt szeretném leírni milyen jellemzői vannak megoldásomnak, szimulációs és digitális iker szempontból is.

Szimulációt nézve egy determinisztikus diszkrét esemény vezérelt rendszerről lesz szó. Ebben a kontextusban az események az egyes egységeknek az állapot változását fogja jelenti, amely során a szabályozási parancs fog végrehajtódni.

Digitális iker felől pedig egy DTA-ról beszélünk, melyben az egyes DTI-k a szimulált inverterek, terhelők, gázmotorok és akkumulátorok. Valamint egy rendszer szintű ikerről tudunk beszélni, mivel a termelő egységek virtuális másolatát valósítottam meg, így például a teljes szabályzó nem került bele.

Az egyes egységek olvasása által létrejövő adatokkal pedig lehetővé teszik a korábban már tárgyalt kétirányú kommunikációt, ahol az olvasott adatok befolyásolják a szabályozási értékeket.

# Felhasznált technológiák

Ebben a fejezetben a azon technológiák lesznek részletezve, amelyeket felhasználtam a szimuláció megvalósítására. Kitérve a Kotlin-ra és nyelvi elemeire, alkalmazott könyvtárakra, a fejlesztő környezetre, a build eszközre, valamint a keretrendszerre. A végén pedig kifejtve a Koin keretrendszerrel, milyen nehézség adódott.

## Kotlin

A Kotlin egy modern, expresszív programozási nyelv, melyet a JetBrains fejlesztett ki. Nyilvánosságra 2011-ben hozták, 2012-ben pedig nyílt forráskódú lett, 2017-ben az Android hivatalos nyelve lett.16

A Kotlin olyan nyelvként szolgál, amely támogatja mind szerveroldali, mind a kliensoldali fejlesztést. JVM felett fut, ami lehetővé teszi a Java kódokkal való kompatibilitást, ennek köszönhetően rendelkezik a Java minden tulajdonságával, de szebb szintaxissal rendelkezik és több olyan beépített funkció van mely segíti a fejlesztők munkáját. Kliens oldali felhasználásában pedig, a Kotlin tud fordulni Javascript-re is.

### Kotlin DSL

A Kotlin egyes funkcióit közösen használva tudunk létrehozni Domain Specific Language-ket (DSL). Ezek akkor lehetnek hasznosak mikor komplex objektumokat szeretnénk létrehozni, vagy konfigurációk létrehozására. Nem könnyű őket létrehozni, de olvashatóság szempontjából mindenféleképpen megéri, mivel elrejti a kódunk komplexitását és sok sablon kódot.17

A DSL-hez szükséges Kotlin funkciók közé tartozik a függvény típusok. Ez a típus egy olyan objektumot reprezentál, melyet lehet függvényként használni. a deklarációja példaként, a Kotlin filter függvényét hozom.

inline fun <T> Iterable<T>.filter(

predicate: (T) -> Boolean

): List<T>

A példában a predicate függvény típusú. A zárójelen belüli a függvény paraméter, a nyíl utáni pedig, hogy milyen típussal térjen vissza. Ha csak egy bemeneti paraméter van arra it-ként lehet hivatkozni. Használatára is hozok példát.

val numbers = listOf(0,1,2,3,4,5,6,7,8,9)

val evenNumbers = lnumbers.filter{ it % 2 == 0}

// evenNumbers = [0, 2, 4, 6 ,8]

Másik szükséges Kotlin funkció az extension function (kiterjesztési függvény), melynek segítségével ki tudunk egészíteni osztályokat függvényekkel anélkül, hogy le kellene származni az osztályból, vagy valamilyen Dekorátor mintát kellene használni. Extension function tudunk függvény típusként is meghatározni, ez fontos eleme a DSL-nek.18

Ezt a 2 funkciót használva, valamint kiegészítve Builder mintával tudunk létrehozni DSL-t.

class Sample(private val value) // Sample osztály

class SampleBuilder{ // SampleBuilder builder Sample classnak var value: Int? = null

fun build(): Sample{

return Sample(value)

}

}

fun sample(init: SampleBuilder.() -> Unit): Sample{ // DSL függvény

return this.init().build()

}

val newSample = samlpe{ //DSL függvény teszt

value = 1

}

Ezt a szimuláció konfigurációs objektumához használtam.

### Kotlin Coroutine

A Kotlin coroutine-ok egy hatékony aszinkron programozási eszköz, amely lehetővé teszi a szálak nélküli, non-blocking kód írását. A coroutine-ok segítségével könnyedén kezelhetjük az aszinkron műveleteket, például hálózati hívásokat vagy adatbázis műveleteket, anélkül, hogy blokkolnánk a főszálat. A suspend kulcsszó segítségével felfüggeszthetjük a futást egy függvényen belül. A a launch vagy async függvények segítségével hozhatunk létre coroutine-okat.

A feladatban fontos szempont volt, hogy a szabályozó és a szimuláció között gyors kommunikáció legyen. Ennek megoldásaképpen a szimulációt a szabályozón belül egy coroutine-ba futtatom, ami így nem blokkolja a főszálat, ahol a szabályzás történik.

### Gradle

A Gradle egy erőteljes, rugalmas nyílt forráskódú build-automatizált rendszer és projekt építő eszköz, amely széles körben használt a szoftver tervezési és fejlesztési folyamatokban. A Gradle támogatja a többnyelvű projekteket, például Java-t, Kotlin-t, és még más programozási nyelveket, és lehetővé teszi a felhasználók számára, hogy hatékonyan építsenek, teszteljenek és telepítsenek alkalmazásokat.19

Az eszköz erőssége a moduláris és konvencionális projektstruktúrák könnyű támogatása, valamint a jól konfigurálható és testre szabható felépítési folyamatok lehetősége. A Gradle továbbá képes automatizálni a projekt függőségek kezelését, valamint támogatja az inkrementális fodítást, ami gyorsítja a fejlesztési folyamatokat.

Alternatívája lehetne a Maven, de mivel a Gradle újabb verziója Kotlin-t használ így ez adta magát.

### Kotlin Data Frame

A Data Frame egy absztrakció a strukturált adatokkal való munkához. Alapvetően ez egy kétdimenziós tábla különböző típusú és elnevezett oszlopokkal. Hasonlóan, mint egy SQL tábla.20

Legnagyobb erősége nem az absztrakcióban rejlik, hanem a Data Frame könyvtár által megvalósított műveleteken, amik megkönnyítik a munkát az ilyen típusú adatokkal.

A megvalósítás során törekedtem a visszakövethetőségre, valamint a későbbi elemzési lehetőségek megteremtésére, így ezzel az API-val valósítottam meg a logolást .csv fájlokba.

### IntelliJ IDEA

Az IntelliJ egy integrált fejlesztő környezet (IDE), amit Java-ban írtak, abból a célból, hogy számítógépes programokat fejlesszenek benne Java, Kotlin és egyéb JVM alapú nyelveken. Kotlin-hoz hasonlóan ez is a JetBrains fejlesztése.

Ebben a fejlesztőkörnyezetben sok hasznos funkció található, melyek segítik a fejlesztő munkáját. Ezen funkciók közé tartozik az integrált adatbáziskezelő és git.

IntelliJ IDEA-t használtam a fejlesztéshez, ami egyértelmű választás volt, hiszen a Kotlin-nal, ez a legkompatibilisebb, valamint ezt használtam, az egyetem és munka közben, Java és Kotlin fejlesztéshez is.

## Kalasim

Ebben a részben bemutatom a keretrendszert, melyet a szimuláció elkészítéséhez használtam. Először általánosan beszélek róla, majd bemutatom, az általam használt elemeket.

### Általánosságban

A Kalasim egy Kotlin-ban írt nyílt forráskódú keretrendszer, mely egy diszkrét eseményszimulátor. Ennek a v0.11.5-ös verzióját használtam. Ezt Holger Brandl fejlesztette, akinek a Kotlin Data Frame elődjét is köszönhetjük. A keretrendszer nagyon jól kidolgozott dokumentációval rendelkezik.21

A diszkrét eseményszimulátor olyan eszköz, mely lehetővé teszi a sztochasztikus, dinamikus és diszkrét módon változó rendszerek dinamikus viselkedésének tanulmányozását. Ilyen rendszerek például gyárak, kikötők, repterek és vezérlés/szabályzás. Ezen kívül tágabb felhasználási körei közé tartoznak a termék tervezés, folyamat automatizálás és vizualizáció, projekt menedzsment és a digitális iker fejlesztés.22

A Kalasim azon szimulációs szakemberek, folyamatelemzők és ipari mérnökök számára készült, akiknek túl kell lépniük a meglévő szimulációs eszközök korlátain, hogy modellezés és optimalizálják üzleti szempontból kritikus felhasználási eseteiket.

Összehasonlításképpen más szimulációs eszközökhöz képest a Kalasim nem low-code és nem is no-code modell alapján működik, hanem code-first mentalitásban. Ez lehetővé teszi a változás követést, skálázhatósságot , refaktorálást, CI/CD-t, unit tesztelést és még sok mást.

A Kalasim, mint már említettem Kotlin nyelven íródott. Egyik fő jellemzőjét használja ki a nyelvnek a Coroutine-et, mellyel a folyamatokat definiálják. JVM-en fut a teljesítmény és skálázhatóság miatt. Koin függőség injektálási keretrendszert használ. Valamint használja az Apache Common Math könyvtárat.

Azért ezt a keretrendszert választottam, mivel a fejlesztéshez választott nyelvem a Kotlin volt, valamint nagyon sok olyan funkciója volt, melyet feltudtam használni és miközben kerestem, ezt találtam a legjobb választásnak.

### Fő funciók

Kalasim egy generikus folyamat orientált diszkrét eseményszimulátor, így ehhez kapcsolódóan sok olyan funkció található benne, mely megkönnyíti a feladatok definiálását.

Az első fő funkció a szimulációs entitások. Ezek a Component névre hallgatnak. Bennük lehet folyamatokat definiálni, amelyek vagy aktiválásra vagy pedig folyamatosan időközönként futnak le.

object: Component() {

override fun process() = sequence {} // aktiválásra használt

override fun repeatedProcess() = sequence{} // ismétlődő folyamat

}

Másik fontos előny a gazdag függvény paletta, mellyel a folyamatokkal tudunk interaktálni. Négy ilyen függvény van, ezek közül nekem egyre volt szükségem, a hold-ra ez egy meghatározott szimulációs időre felfüggeszti a folyamatot. Valamint a repeatedProcces()-t ennek a függvény segítségével lehet ütemezni, milyen gyakorisággal ismétlődjön.

object: Component() {

override fun process() = sequence {

hold(10.minutes) // felfüggeszti szimulációs időben 10 percre

}

Az említett négy függvény közé tartozik még, a request(), amely a erőforráshoz (Resource) kapcsolódó feltétel ellenőrzésére használt. A következő a wait(), ez adott állapothoz (State) tartozó feltételt vár. Az utolsó pedig a passivate, ezzel lehet komponenst passzív állapotba helyezni. Ezeket megvalósítás során nem használtam, így csak említés szintjén írtam róluk.

Másik fontos funkciója a Kalasimnak az események időzítése, melyek úgy tűnnek, hogy egyszerre futnak le viszont az egyszálú működés miatt a folyamatok egymás után egyesével fognak lefutni. A Kalasim egy priorityQueue-t használ ehhez.

A priorityQueue lényege, hogy benne az elemek prioritás alapján vannak sorban, és ebben a sorrendben hajtódnak végre. Az egyes elemek fontossága alapesetben nincs különbség, ezt kódban lehet explicit megadni. Erre a Kalasim nyújt metrikát, melyben a LOWEST, LOW, NORMAL, IMPORTANT és CRITICAL a beállítható értékek, de ezeken kívül a Priority() osztály segítségével sajátot is meghatározhatunk.

A screenshot of a computer

Description automatically generated

**4.1 Ábra: Kalasim végrehajtási modellje**

Valamint a Kalasim beépített montoring és statisztika gyűjtővel rendelkezik, ezeket jelenleg nem használtam fel, de későbbiekben hasznos elem lehet, amellyel tisztább képet lehet előállítani az egységek belső állapotáról futás közben.

A Kalasim nyújt ezen felül beépített esemény naplózási lehetőséget, ezt felhasználva történik a .csv fájlok létrehozása, melyekből diagramok készülnek.

### Szimuláció konfiguráció

Szimulációk konfigurálásához is sok elemet nyújt a keretrendszer. Erre az Enviroment osztályt használja. Ezen belül belehet állítani, hogy egy Tick (szimulációs idő egy egysége), mennyinek felejen meg a valóságban. Valamint azt is állítható elem, hogy a komponenseket logolja-e.

ClockSync(tickDuration = 1.seconds) // 1 tick mennyi idővel egyenlő valós időben

addEventListener { it : Event -> println(it) } // ha elkap egy eseményt kiírja konzolra

Az alapvető szimuláció létrehozásához van lehetőségünk DSL-t használni viszont, ha összetettebb, számunkra megfelelő szimulációs környezetet szeretnénk, akkor az Enviroment osztályból kell leszármaznunk.

Az Enviromenten belül lehet állítani a véletlenszám-generátor kezdeti állapotát is, ezzel azonos paraméterű futatásokkal a random értékek eltérhetnek.

### Nehézség

Ez volt az első, hogy úgy használtam nyílt forráskódú rendszert, hogy értelmeztem annak működését, és abból leágazva próbáltam megoldást hozni a saját problémámra. Ez a probléma a Koin függőség injektáló rendszer köré összpontosult.

Ez abból adódott, hogy a Koin minor verzió váltások között nem tart bináris és viselkedésbeli kompatibilitást. Számomra a Koin 3.3.3-as verziója kellett még a keretrendszer a 3.1.6-at használta. Ezt a magyarázatot egy github issue-ban találtam meg.23

A különbség a verziók között a lazy loading-ban volt. Régi verzióban a module-ban való felsoroláskor megtörtént az inicializálás, az újabb verziókban viszont, első használatkor fog.

Megoldásképpen próbálkoztam a verziók és a forráskód átírásával, de ezek nem váltak be. Végső megoldás az lett, hogy .jar fájllá alakítottam a függőségei nélkül, majd a szükséges függőségeit utólag adtam, meg a saját projektemben.

# Implementáció

Ebben a fejezetben mutatom be a megvalósított kódot és mutatok szimulációs eredményeket.

A feladatott két program formájában oldottam meg. Az egyik maga a szimuláció, a másik pedig az első rétegben található szabályozó. Azért választottam szét őket, mivel mikor a szimuláció éles rendszerben lesz, akkor is hasonlóan fog működni, valamint így azt is megtudtam nézni, hogy hogyan tudom függőségként használni a szimulációt.

Először a szimuláció kódbázisát mutatom be, majd a szabályozó kódját, aminek része lesz a szimulációs eredmények is.

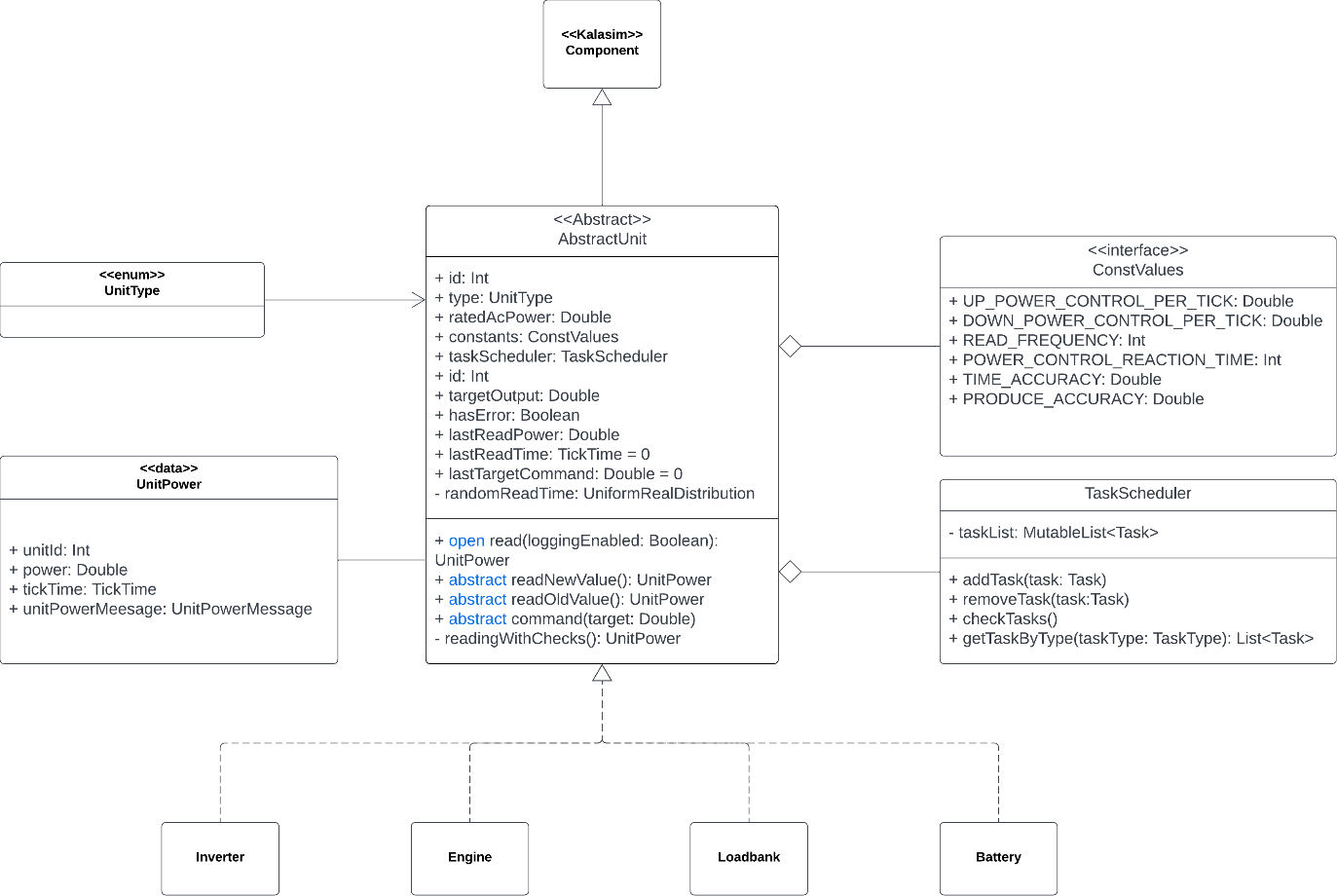
## Szimuláció

A kód kifejtésében lentről felfelé fogok haladni. Így felépítve az egyes elemek közötti kapcsolatot. Közben pedig kifejtem, hogy a Kalasim milyen funkcióját használtam fel.

Azt is kifejtem az egyes komponenseknél, hogy milyen megfontoltságból választottam az adott megoldást.

### Egységek

Az elemek kifejtését az egységeknél (Units) kezdem. Felépítése az 5.1-es ábrán látható.



**5.1 Ábra: Egységek felépítésének osztálydiagramja**

A diagrammon látható elemek közül először a függőségeket fejteném ki.

Az első ilyen a UnitType. Ez egy enum, amelyben az egység típusok vannak meghatározva. Erre az esetleges típus ellenőrzés és castoláskor van szükségem.

A UnitPower egy egységre megmondja, hogy mennyit termelt az adott időpillanatban. A UnitPowerMessage segítségével pedig azt is megmondja, hogy akkor éppen termelt (PRODUCE), fogyasztott (CONSUME) vagy éppen hiba lépett fel (ERROR). Ennek az események naplózásakor van jelentősége, ez összegyűjti, az ahhoz szükséges adatokat.

#### Konstans értékek

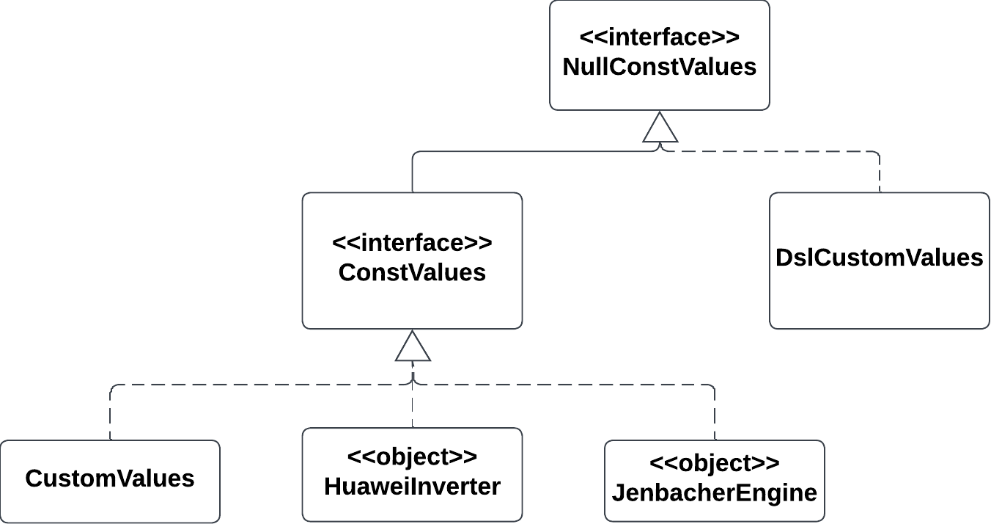
Ezek olyan értékek, melyek előre meghatározottak az egység típusától függően. Az UP/DOWN\_POWER\_CONTROL\_PER\_TICK, azért felel, hogy az akkumulátor, milyen gyorsan tud töltődni vagy milyen gyorsan tudja azt leadni. Nagyobb parkok esetén az is lehet, hogy milyen gyorsan tudja elérni a szabályozó által adott célértéket.

A READ\_FREQUENCY, azért felel, hogy milyen időközönként lehet friss adatot olvasni az egységből. Alapvetően a szabályozó két másodpercenként olvas, viszont olyankor nem biztos, hogy friss adatot olvas, hanem még az előző értéket olvashatja.

A POWER\_CONTROL\_REACTION\_TIME, azért felel, hogy milyen gyorsan reagál az új célértékre. Hány másodperc elteltével fogja az új értéket termelni.

Az egyes egységek olvasáskor nem fix, hogy pontosan a READ\_FREQUENCY ideje alatt fog olvasni, hanem lehet, hogy kicsit több vagy kevesebb, ezt az ingadozást határozza meg a TIME\_ACCURACY. A TARGET\_ACCURCY pedig, azt, hogy az egységek, nem tudják mindig pontosan követni a célértéket valamelyik irányba kitérnek, és ezt a kitérést határozza meg.

Ezen konstans értékeknek összetett hierarchiája van, mely az 5.2-es képen látható.



**5.2 Ábra: ConstValues hierarchia**

Ennek a hierarchiának köszönhetően, minden leszármazottban az előzőleg felsorolt értékek lesznek. A legfelső szinten a NullConstValues-ban az értékek nullable változata található. Erre azért van szükség, mivel a feladat része volt, hogy legyenek alapértelmezett értékek, de lehessen ezeket az értékeket változtatni, így ezzel a megoldással meg lehet tenni, hogy nem az összes értéket határozzuk meg, hanem csak egy részét.

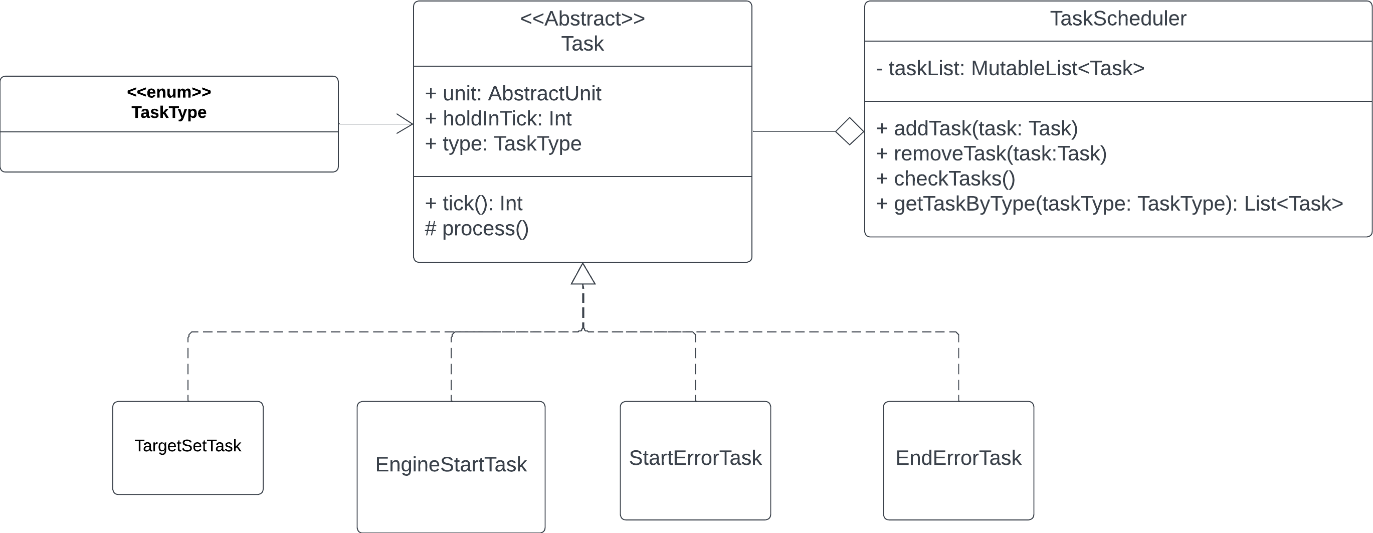
Az értékek konfigurálását DSL-ben oldom meg, ezért is van a DslCustomValues, melynek értékei lehetnek null-ok és később ez példányosul CustomValues-ra, aminek már minden értéke konstans.

A másik ágon a ConstValues-ban az értékek már a nem nullable formában vannak. Erre azért volt szükség, mivel kellett egy absztrakciós szint, amit, ha használok, nem kell null ellenőrzést használnom, hiszen a az alsóbb szinteken már nem változhatnak az értékek.

Az alapértelmezett értékeket pedig, mint Példány (Object) határozom meg, így ezekből csak egy példány van csak.

#### A feladatok és ütemező

A taszkok és az ütemezők segítségével értem el, hogy tudjak időzíteni feladatokat. Például, ha egymás után érkezik szabályzási parancs különböző célértékkel, akkor így mindegyik egy adott időpillanatban megfog történni, vagy még ezzel időzítem a hibákat is melyeket a szimuláció használója előre meghatároz. Az ütemező és a taszkok felépítését a 5.3-as ábra mutatja.



**5.3 Ábra: Taszkok és ütemező kapcsolata**

A TaskType-ra ebben az esetben típus ellenőrzés miatt kell. Főleg a gázmotor estén szükséges, ahol ameddig indul a gázmotor kaphat szabályzási parancsot, viszont azok csak indulás után történhetnek meg, azért először a motor indítási feladatnak kell, hogy mennyi idő van belőle vissza és csak az után indulhat. Valamint, ha visszavonják a motor indítását, de a szimulációba vannak előre meghatározott taszkok, akkor azok ne törlődjenek ki.

A Task működése a tick() függvényt ha meghívják levon egyet a visszalévő időből, ha az idő letelik, akkor meghívja a process() függvényt, ami végrehajtja az ütemezett feladatot.

fun tick(): Int{

holdIntTick -= 1

if(holdInTick <= 0)

process()

return holdInTick

}

A process() pedig a leszármazott osztályokban van implementálva, így működése tesztre szabható. Itt például a StartErrorTask függvénye.

override fun process(){

unit.hasError = true

}

Ezen felül még az egyes leszármazott osztályokba új attribútumot is fellehet venni, amit például a TargetSetTask-ban használok, hogy beállítsam a célértéket.

A TaskScheduler checkTasks() függvénye pedig meghív minden benne lévő feladatra a tick()-et és ha ez nullával tér vissza, akkor kiveszi az adott taszkot a listájából.

fun checkTasks(){

val iterator = taskList.iterator()

while (iterator.hasNext()){

val task = iterator.next()

if(task.tick() <= 0)

iterator.remove()

}

}

#### Absztrakt egység

Ahhoz, hogy az egyes szimulált egységeket tudjam közösen kezelni bekellett vezetnem egy absztrakciós szintet, ez lett az AbstractUnit. Ebben vannak kiemelve a funkciók melyek, minden egységben előfordul. Fő elemek ként a read() és command().

A command() különbözhet az egyes egységeknél, így ez csak egy absztrakt függvény lett.

A read() függvény viszont, már jobban összevonható. Olvasáskor két fő eset van, mikor az új értéket lehet olvasni és mikor még a régi értéket lehet csak. Erre a két lehetőségre vannak absztrakt függvények, melyekben meglehet határozni mit és hogyan adjanak vissza. Ez a két metódus a readNewValue() és a readOldValue(). Ezek be vannak csomagolva a readingWithChecks() függvénybe, ami ellenőrzi, hogy adott egységnek van-e hibája. A read() pedig ezt hívja meg.

A read() függvénynek van egy bemenő paramétere a loggingEnabled, mely az események naplózásáért felel. Ezt akkor használom, mikor a felül írom ezt a függvényt és belerakom az event-ek létrehozását. Itt a Kalasim által nyújtott log rendszert használom.

override fun read(loggingEnabled: Boolean): UnitPower{

val power = super.read()

eventLogging(loggingEnabled) { … }

return power

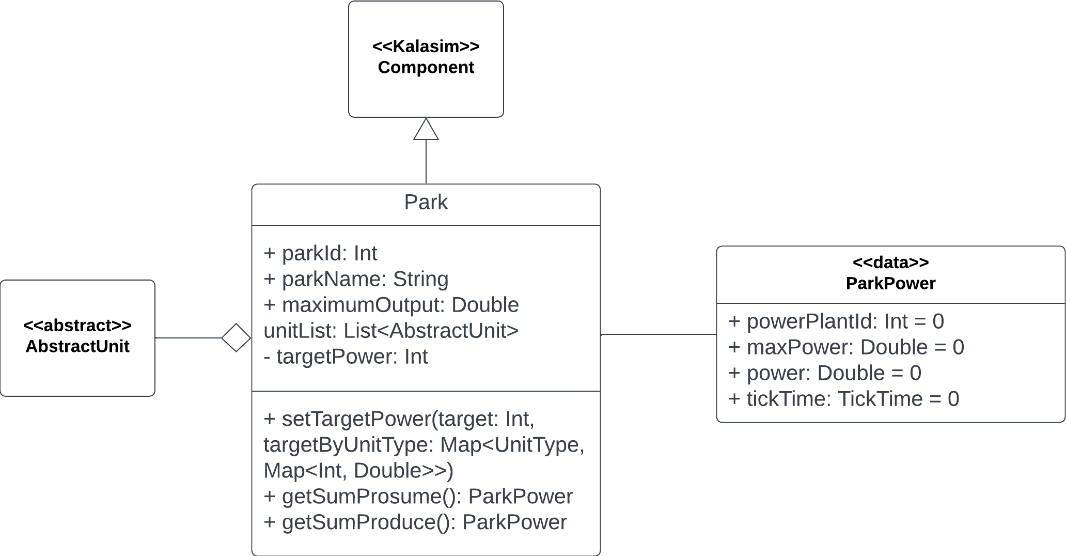
}

Azért volt szükség a bool bemenő paraméterre, hogy úgy is lehessen használni, a read() függvényt, hogy az nem logol.

Valamint ebben az absztrakt osztályban származok le a Kalasim Component osztályból is.

### Parkok

A park egy erőműnek a megfelelője és feladat, hogy a termelő egységeket összefogja és azoknak továbbítsa a szabályozó által meghatározott célértékeket. Ennek felépítése a 5.4-es ábrán látható.



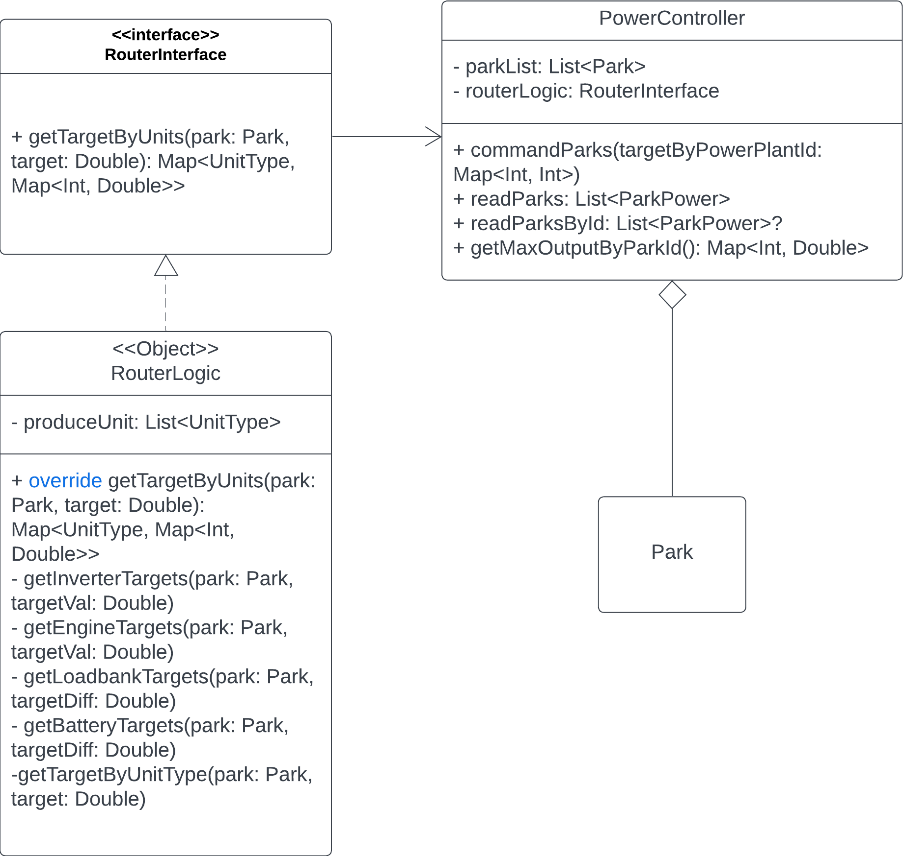
**5.4 Ábra: Taszkok és ütemező kapcsolata**

A ParkPower gyűjti össze, hogy adott parkban az egységet mennyit termeltek és fogyasztottak összesen az adott időpillanatban és hogy a park mennyit tud maximálisan termelni. Ez a esemény naplózás miatt lesz fontos, ahol megint a Kalasim logolását használom. Az osztály ezért valósítja meg a Component-t.

A Park osztálynak két függvénye van, amit a termelés olvasására lehet használni. Az első getSumProsume(), ezt a függvényt hívja meg a szabályozó mikor beolvassa a jelenlegi termelést, valamint eseményekhez is ezt használom. A másik pedig a getSumProduce(), amelyet számoláshoz használok. Az utóbbi függvényben az egységek nem naplóznak.

### Egység szabályozó

Ez a második fele a szabályzásnak, itt történik az erőmű szintű parancs lebontása, termelő egység szintű célértékre. Ennek felépítése az 5.5-ös képen látható.



**5.5 Ábra: Szabályozó és hozzá kapcsolódó elemek**

A PowerController használja fel a RouterInterface-t valamint ez a komponens ismeri az összes parkot és tudja szabályozni és olvasni azokat. Ezen kívül még visszatudja adni az egyes parkokhoz tartozó maximális kimenetet.

#### Router helyettesítő

A Router jelen helyzetben a PLC-t jelenti, ami kint van a napelemparkokban és felelős azért, hogy a napelemparkba bejövő célértéket lebontsa az inverterek, gázmotorok és többi egység szintjére.

Jelen helyzetben még egy tényleges PLC-nek a kódját nem lehetett bele integrálni a szimulációba, így nekem kellett egy ideiglenes implementálnom.

Annak érdekében, hogy ez a logika könnyen cserélhető legyen a PowerController-ben egy interfész segítségével van függőségként. Ez a RouterInterface, melyben egy függvény található a getTargetByUnits(). Ennek bemenő paramétere az adott Park, valamint a hozzá tartozó célérték.

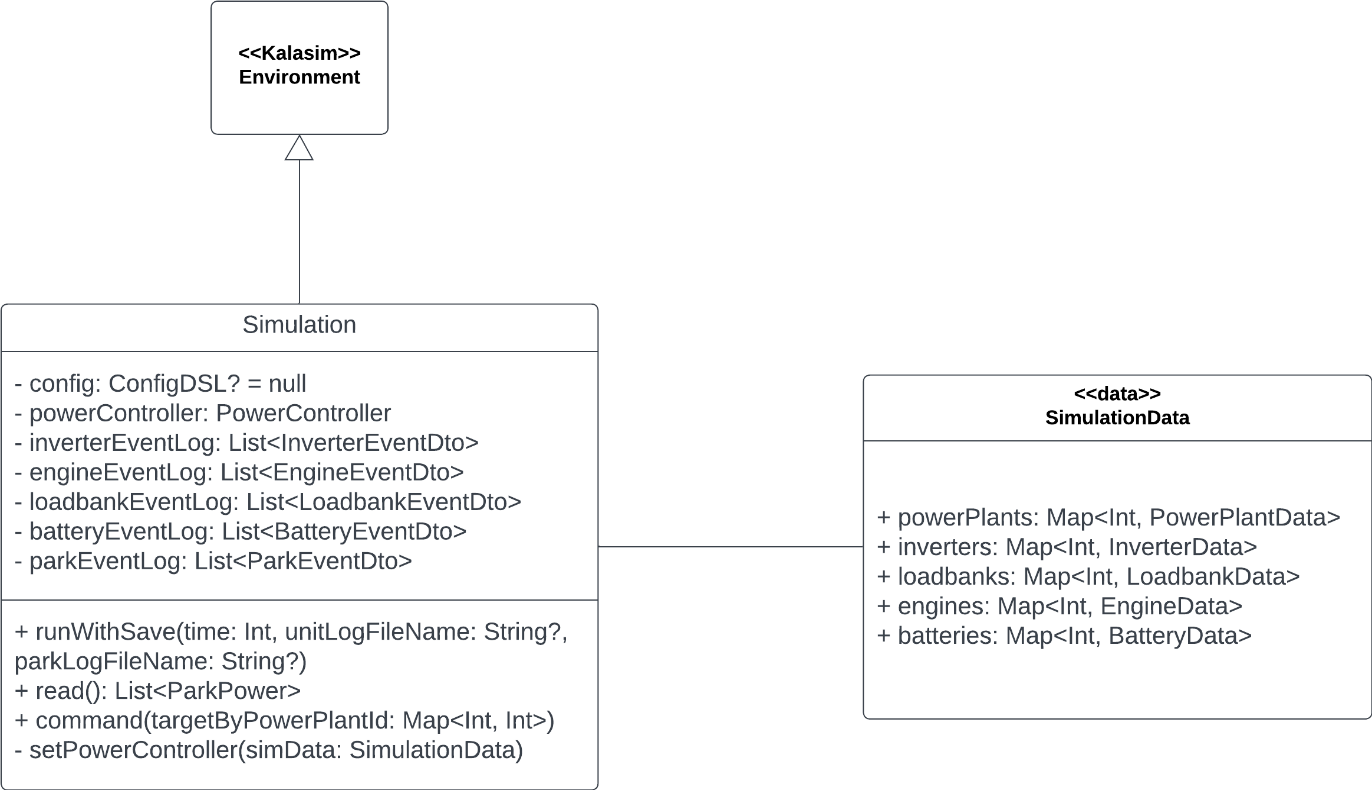
Ennek az interfésznek a jelenlegi megvalósítása a RouterLogic objektum. Ennek a működésének részletezése jön. Jelenleg feladat alapján két egység típus van. Az első a csak a termeléssel foglalkozó egységek, ide tartozik az inverter és a gázmotor. A másik csoportba azok az elemek tartoznak melyek a jelenlegi termelés és a célérték közötti különbséget csökkentik, ezek a terhelők és akkumulátorok.

Az inverter és a gázmotorok között elosztott célértékből számolja ki, hogy egyes egységeknek mennyit kell termelnie. A terhelők és akkumulátorok logikája pedig a különbséget kapja meg és az alapján fog szabályozni.

### Szimuláció osztály

A Simulation osztály feladata a szimulációs objektumok és köztük lévő kapcsolatok létrehozása, az események kiírása, valamint a szimuláció indítása és kommunikáció megvalósítása. Az osztály feladatainak elvégzésére megvalósítja a Kalasim Environment osztályát.

A következőkben ezen osztály részletezése fog következni. Fontosabb attributumok, függvények és osztályok az 5.6-os ábrán láthatóak.



**5.6 Ábra: Simulation osztály**

#### Szimuláció felépítése

A környezet és az objektumok létrehozásában fontos szerepet játszik a Simulation osztály konstruktora, így ennek bemenő paraméterei lesznek először kifejtve.

class Simulation(

simData: SimulationData,

randomSeed: Int,

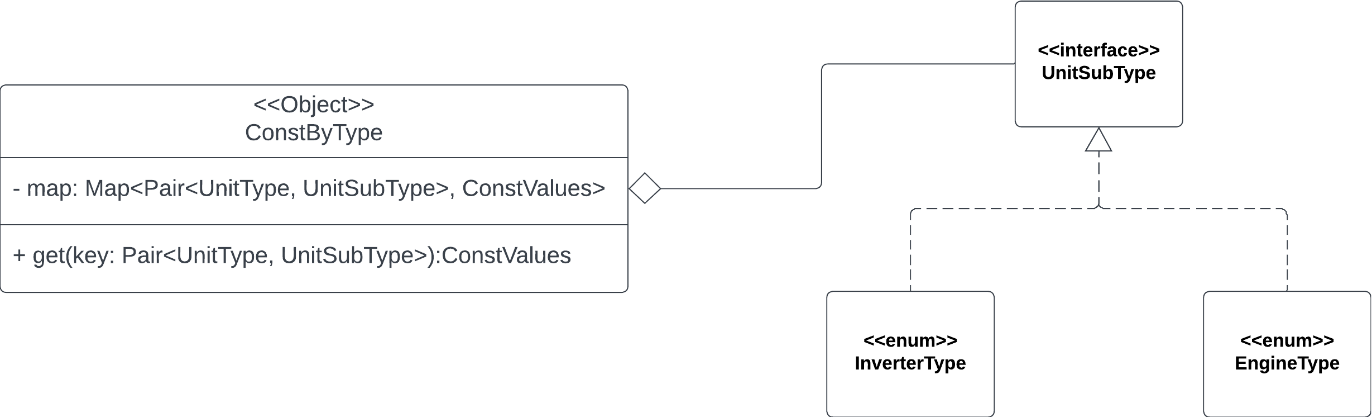
inRealTime: Boolean,

private val config: ConfigDsl:?

): Environment(randomSeed = randomSeed)

A simData tárolja el, hogy milyen egységek fognak a szimulációban szerepelni. Ezeknek külön adat osztályaik vannak. Ezeken belül az adott egységre specifikus adatok vannak például id, az erőműnek az id-ja amiben elhelyezkedik, maximális teljesítmény kimenet, valamint az egységek altípusai.

Az altípusok segítségével oldottam meg, hogy egyszerűen eltudjam dönteni, hogy adott egységhez melyik konstans értékek tartoznak. Ennek szemléltetése az 5.7-es ábrán látható.



**5.7 Ábra: Konstans értékek összekötve típus párokkal**

Konkrét példát hozva a megoldásra.

val constValue = ConstByType.get(Pair(UnitType.INVERTER, InverteryType.HUAWEI))

// constValue => HuaweiInverter

val constValue = ConstByType.get(Pair(UnitType.ENGINE, EngineType.TEST))

// constValue => TestEngine

Az egységeknek a típusát pedig abból lehet megállapítani, hogy a simData-ban külön vannak tárolva, így ez magától értetődő.

Következő paraméter a randomSeed. Ezzel a paraméterrel lehet a véletlenszám generátort kezedeti állapotát beállítani. A random generátor főleg az AbstractUnit-ban jelenik meg, ahol ezzel szimulálom a parancsfeldolgozás, olvasás és célérték követés pontatlanságát.

Az inRealTime változó a tovább fejlesztéskor lesz fontos, mikor esetleg a szimuláció gyorsabban tud, majd futni, mint valós idő. Jelenleg igaz állapotban kell használni.

if(inRealTime) ClockSync(tickDuration = 1.seconds)

A config változóban található azok az értékek, lehet konfigurálni a szimuláció adatait. Köztük az, hogy az egyes egységeknek mikor legyen hibája vagy mi legyen a kezdő termelése, konstans értékek eltérjenek-e az alapértelmezettől. Ennek megvalósítására DSL-t használtam.

Példa a DSL használatára:

val testConf = config{

addDefaultProducing(UnitType.INVERTER, 0.5)

addTypeConfig(UnitType.INVERTER, InverterType.Test){

UP\_POWER\_CONTROL\_PER\_TICK = 2.0

READ\_FREQUENCY = 6

}

addUnitConfig(UnitType.INVERTER, 1){

addDefValues{

hasError = true

}

addTask{

listOf(

DslStartErrorTask(10)

DslEndErrorTask(20)

)

}

}

}

A szimuláció építésekor, az egyes egységek adat osztályain végig megy és azokat a megfelelő park inicializálásakor létrehozza, figyelembe véve a konfigurációs objektumot.

#### Szimuláció futtatása és esemény kiírás

A szimuláció tud határozott és határozatlan ideig futni, viszont csak az elöbbi esetben lehet az eseményeket kiírni, ennek megoldására az Environment run() függvényét becsomagoltam egy runWithSave() függvénybe. Ennek a függvénynek a bemenő paraméterei, hogy meddig fusson a szimuláció, hogy milyen fájlnév alá mentse az adatokat és hogy hova mentse azokat. Alapértelmezetten a main-en belül a kotlin mappa mellé kell létrehozni egy LogResults mappát és ide menti el a .csv fájlokat.

Az egyes események a Kalasim Event osztályából származtak le. Ez viszont a Kotlin Data Frame könyvtárat használva kiíráskor felesleges adatokat is kiírt, emiatt hoztam létre az eseményeknek megfelelő DTO-kat. Így az Event-ek elkapása után DTO konvertált formát mentettem el és írattam ki.

addEventListener{it:InverterReadEvent

->inverterEventLog.add(InverterEventDto(it))}

Az, hogy a program naplózza-e az eseményeket lehet állítani bool változók segítségével. Viszont ezek nem szimulációhoz kötöttek, hanem globális változók, amiket felhasznál a program. Ezek a LogFlag objektumban találhatóak.

Egy példa a DataFrame használatára:

inverterEventLog.toDataFrame().writeCSV(fileName)

A parkohoz és az egyes egység fajtákhoz tartozó események külön .csv fájlba menti. Az egységekhez tartozó fájlnevek végére oda kerül, hogy melyik fajtához tartozik. Inverter esetén INV, gázmotor esetén ENG, Terhelő esetén LD és akkumulátor esetén pedig BT.

Fájlok neveit lehet állítani a runWithSave() bemeneti paramétereinek segítségével, vagy van alapértelmezett név is, amely a parkLog vagy unitLog és a szimuláció indulási dátumjából áll.

val name = unitLogFileName ?: „unitLog${fileNameDateFormater.print(startDate)}”

Ezekből a fájlokban lévő adatokból lehet diagramokat csinálni. A diagramok megalkotására egy Python kódot használtam, ami nem az én kezem műve így ezt csak megemlítés szintjén írom bele szakdolgozatomba.

### Segéd eszközök

Ezek az eszközök segítenek az események logolásában és a formázásban.

A programban két fajta logolás van. Az egyik az események .csv-be való kiírása. Ennek van egy segéd függvénye, ami igazából csak egy feltétel hordoz magában. De ennek a függvény segítségével könnyem megtalálhatóak a helyek, ahol esemény logolás történik.

fun eventLogging(isLoggingEnabled: Boolean, block: () -> Unit){

if(isLoggingEnabled)

block()

}

Ez a függvény a EventLoggingUtil-ban található, ahol mellette van még a PATH változó, melyben meglehet határozni, hova kerüljenek a .csv fájlok. Valamint itt található a LogFlags objektum is.

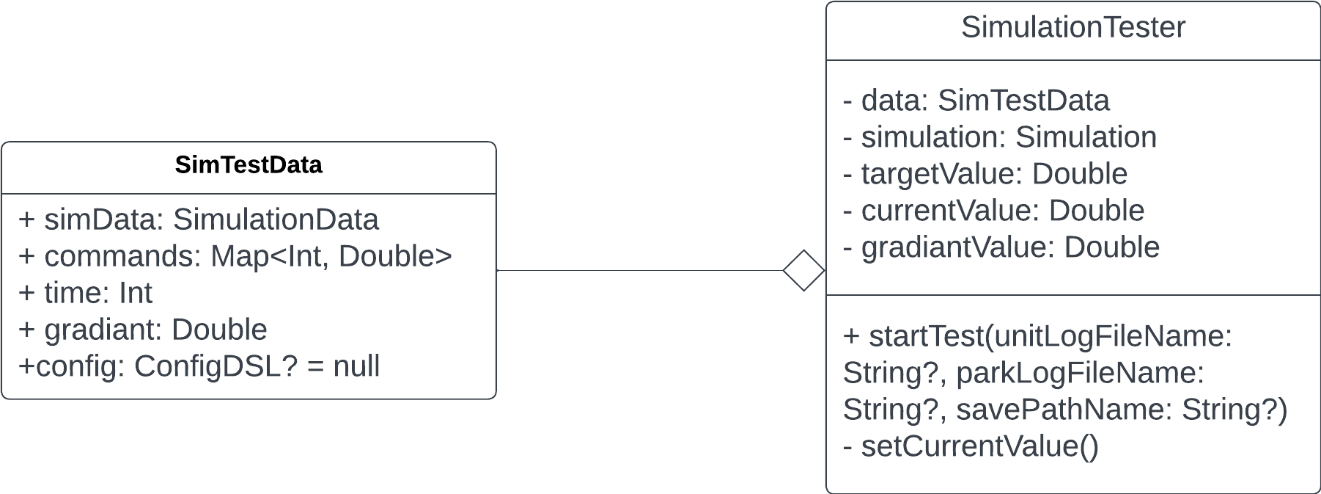
A másik fajta naplózás, a konzolba való információ megjelenítése. Erre a kotlin-logging github-os24 könyvtárat használom, amely a Java-s slf4j-t csomagolja be és nyújt mellé kényelmesebb használatot.

Az utolsó segédeszköz a dátumok formázásáért felel. Egyik forma, ami az eseményel fájlba való kiírásánál használt ez a hónapot-napot-órát-percet jeleníti meg. A másik pedig ISO Date formátum a másodpercig.

## Erőmű szabályzó és tesztelő

Ennek a programnak a feladata, hogy helyettesítse a MAVIR-t és a szimulációnak tudjak szabályzásra parancsot adni. A szimuláció a libs mappában van .jar formájában. A fájl neve Simulation-0.0.1-fat.jar. Ez egy fat JAR25 vagy, más néven uber JAR. Lényege, hogy ebben nem csak az osztályok vannak benne, hanem a függőségek is.

A program két részből áll. Ezek között soros kapcsolattal. Ez az 5.8-as ábrán látható.



**5.8 Ábra: Program osztályai és kapcsolatuk**

Az első Data osztály, melyben a tesztesetekhez szükséges adatok találhatóak. Ezeket a SimTestData osztályban gyűjtöm össze. Az első attribútum a SimulationData, ezt már előzőekben részleteztem, ebből épülnek fel a szimuláció objektumai. A második a commands, ebben szám párok vannak eltárolva. A pár első fele azért felel, hogy hányadik másodperctől kezdődöen használja szabályzási értékként a pár második számát. A time, azt határozza meg meddig tartson a szimuláció másodperceben. A gradiant, azért felel, hogy milyen gyorsan mehet az erőművek szabályzása, ez részletesebben a SimulationTester-nél lesz kifejtve. A config pedig a konfigurációs objektum, amelyről már eset szó.

A második része a SimulationTester, ez a SimTestData segítségével először felépíti a szimulációt, majd futatja azt és vezérli a commands értékeivel. Ennek megoldása coroutine-ok segítségével lett megoldva.

A szimuláció külön coroutine-ban van elindítva ameddig a főszálon történik az szabályzás és az olvasás. Futás végén pedig a coroutine scope tér vissza a szimulált adatokkal. Ennek illusztrációja a következő leegyszerűsített kódrészlet.

suspend fun startTest(): Map<Int, List<ParkPower>>{

return coroutineScope {

launch{ simulation.run() }

while(!isTimeUp()){

read and command

}

return@coroutineScope result

}

}

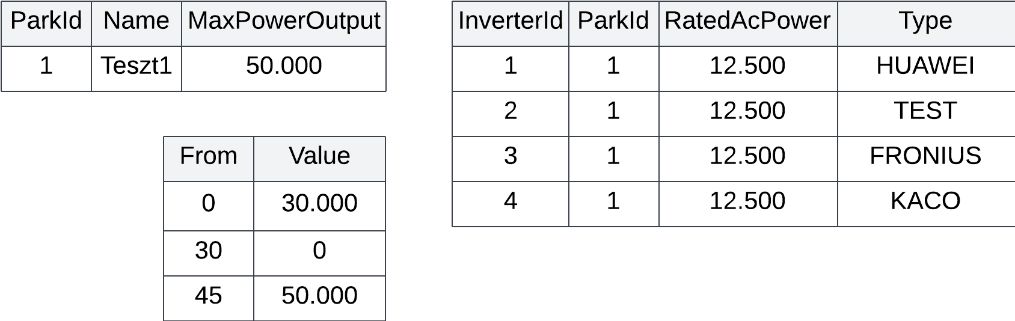
## Szimulációs eredmények

A szimulációt több tesztesettel teszteltem, melyek a program egyes aspektusait kipróbáljak. Így külön készítettem teszteket az egyes egységek tesztelésére, a konfiguráció tesztelésére, valamint egy komplex tesztet is. Ezen felül a valós rendszerben való működéséről is mutatok, majd diagramot összehasonlítva egy éles szabályzással.

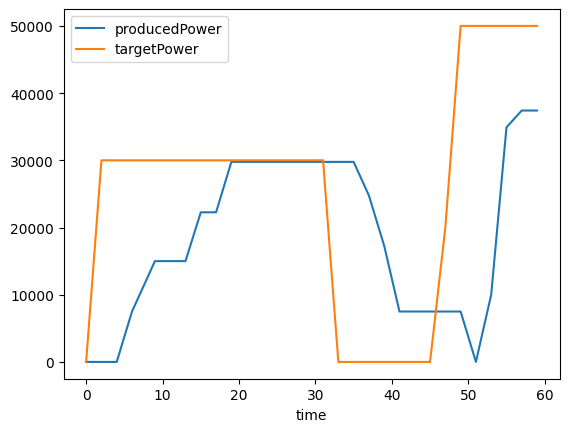
A szimulációkról az több eredmény is lesz. Az első az erőművek termelési diagramját, amelyen azt mutatja, hogy mennyit termel a park és mennyi a célérték. A többi pedig az egységekhez köthető. Ezeken látható a maximum lehetséges termelés, a minimum lehetséges termelés, a célérték és a tényleges termelés. Valamint az akkumulátornál van egy olyan, ami azt mutatja mennyi a töltöttsége.

### Inverter teszt

Ebben a tesztben négy inverter van, amelyek kimenő maximum teljesítménye egyenlő, viszont típusuk eltér. Ez befogja folyásolni, hogy milyen időközönként lehet friss adatokat olvasni és milyen gyorsan fogja a szabályzási parancsot végrehajtani.



**5.9 Ábra: Test 1 szimuláció adatai**

****

**5.10 Ábra: Test 1 park termelési diagram**

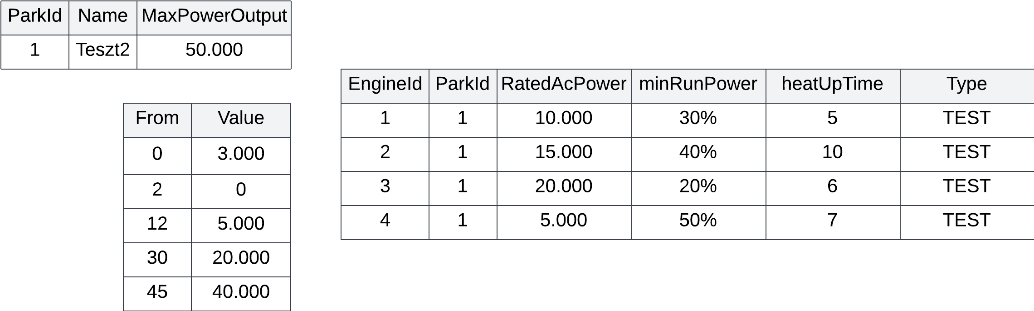
|  |  |
| --- | --- |
|  |  |
|  |  |

**5.11 Ábra: Test 1 egységek termelési diagramok**

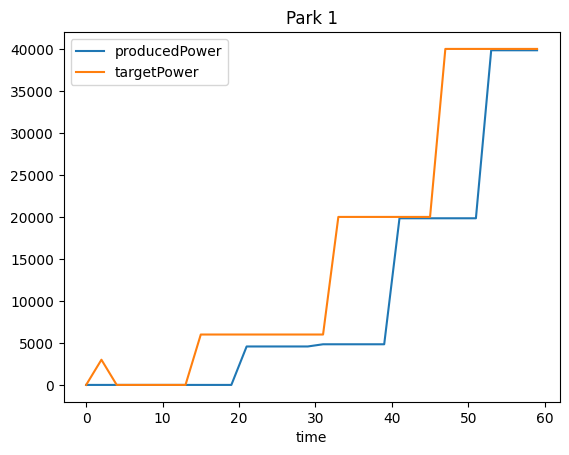
A teszteken látható, hogy a Huawei és a Test típusú inverterek eredménye hasonló, ez abból adódik, hogy ennek a két típusnak a konstans értékei nagyon hasonlóak. Ellentétben a Fronius és a Kaco-val. A Fronius típusúnak az olvasási gyakorisága tizennégy másodperc, ez a digramon látható is, hogy a az új értékeket kb ennyi időközönként olvassa. A Kaco típusúnak viszont a parancs feldolgozási ideje tizenhat másodperc az olvasásának gyakorisága viszont viszonylag gyors. Az ábrán látható is, hogy nagyobb elcsúszással kezdte el termelni az első célértéket.

### Engine teszt

Ebben a teszteben a motorok indulási logikáját tesztelem. Ehhez 4 gázmotort használok, különböző maximum termeléssel ugyan olyan típussal. Azt lesz fontos megfigyelni, hogy melyik és mikor fog elindulni.



**5.12 Ábra: Test 2 szimuláció adatai**



**5.13 Ábra: Test 2 park termelési diagram**

|  |  |
| --- | --- |
|  |  |
|  |  |

**5.14 Ábra: Test 2 egységek termelési diagramok**

Látható, hogy az első szabályzási parancsra, a legkisebb gázmotor indult el. A következő szabályzásoknál pedig, ahogy egyre nőt a célérték, úgy egyre több indult el, de nem a legnagyobb, mivel a logika figyelembe vette az addig elindult gázmotorokat is.

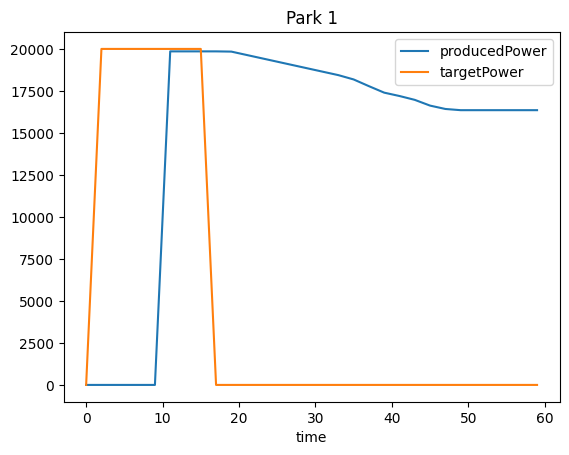
### Terhelő teszt

Ebben a teszteben a terhelők logikájának bemutatásán lesz a hangsúly. Ennek megvalósítására felhasználtam egy gázmotort, amely fixen termel egy adott értkét és beállítottam, hogy parancs teljesítési ideje fél perc legyen, ezzel a termelési különbsége nagysága sokáig állandó lesz és minden terhelő aktiválódni fog.

A screenshot of a computer

Description automatically generated

**5.15 Ábra: Test 3 szimuláció adatai**

****

**5.16 Ábra: Test 3 park termelési diagram**

|  |  |
| --- | --- |
|  |  |
|  |  |

**5.17 Ábra: Test 2 egységek termelési diagramok**

Az egységek termelési diagramján látszik, hogy időben az egyes terhelők egymás után indultak el, attól függően, hogy melyik tudja a legtöbb áramot felvenni. Ez a park termelési ábráján is megmutatkozik, ahol a termelés láthatóan csökken.

### Akkumulátor teszt

Ebben a tesztben az akkumulátorok működése lesz e lényeg, ahogy feltöltenek, majd felszabályzás esetén pedig a tárolt energiát leadják. Ebben az esetben is konfiguráció során a parancs végrehajtási idő fél percre lett növelve.

A table with numbers and text

Description automatically generated

**5.18 Ábra: Test 4 szimuláció adatai**

|  |  |
| --- | --- |
|  |  |

**5.19 Ábra: Test 4 park és a benne lévő gázmotor termelési diagramja**

|  |  |
| --- | --- |
|  |  |
|  |  |

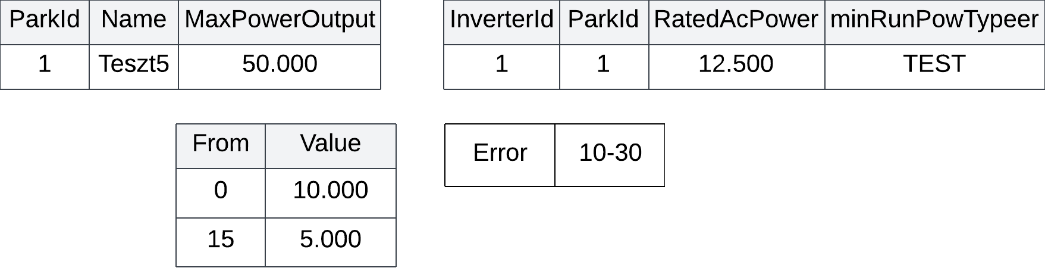
**5.20 Ábra: Test 2 egységek termelési diagramok**

A parkban látható, hogy az egyetlen termelő egység a gázmotor és összehasonlítva a park diagramjával látható, hogy tizenkettedik másodperctől elkezd töltődni az akkumulátor, majd mikor megnő a célérték és az akkumulátor is feltöltődött azt kiadja magából. Ez megjelenik az egységek töltöttségi diagramján is.

Az akkumulátorok nem teljesen töltődnek fel teljesen és azt se hagyja a rendszer, hogy teljesen lemerüljön.

### Meghatározott hiba teszt

Ebben a teszteben a konfigurációban meghatározható hibát fogom tesztelni egy inverteren. Teszt negyven másodpercig fut, a hiba pedig a tizedik másodperctől tart a harmincadikig.



**5.21 Ábra: Test 5 szimuláció adatai**

A graph of power consumption

Description automatically generated with medium confidence

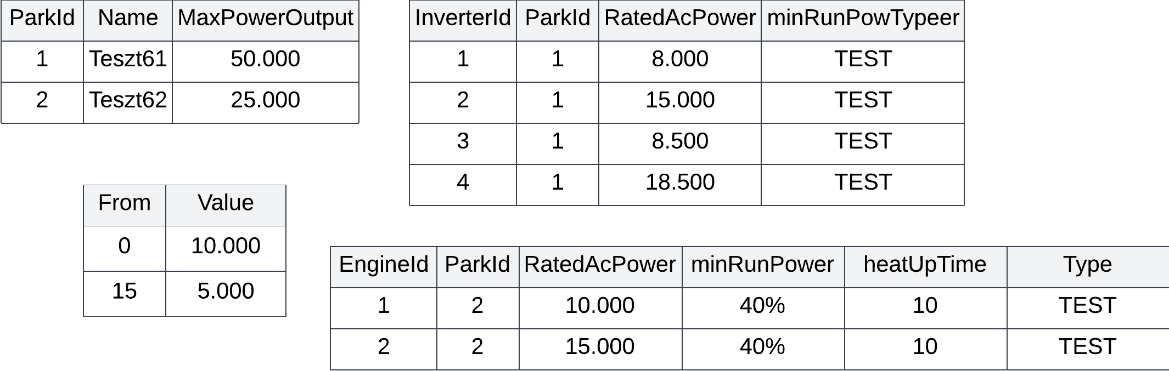
**5.22 Ábra: Test 5 inverter termelési hiba közben**

A teszt során a hiba megjelén, azt fogja jelenteni, hogy az invertert nem lehet olvasni, viszont közben ugyan úgy termel. A célértéken látszik, hogy azt közben ugyan úgy megkapta, így a hiba csak az olvasást befolyásolja.

### Konfigurációs teszt

Itt két tesztet fog futni, melyek között a különbség, hogy a második konfigurációjában több paraméter is megváltozik ezzel eltérnek az eredmények és ezek lesznek összehasonlítva.

Valamint ez a teszt mutatja azt is, hogy két park között, hogy osztja el a szabályzó a célértéket.



**5.21 Ábra: Test 6 szimuláció adatai**

|  |  |
| --- | --- |
|  |  |
|  |  |

**5.22 Ábra: Test 6 tesztek parkjainak termelési diagramjai**

|  |  |
| --- | --- |
|  |  |
|  |  |

**5.23 Ábra: Test 6 tesztek invertereinek termelési diagramjai**

|  |  |
| --- | --- |
|  |  |
|  |  |

**5.24 Ábra: Test 6 tesztek gázmotorjainak termelési diagramjai**

A konfiguráció változtatások közé a következők tartoznak:

Az inverterek számára az, hogy 30% termeljenek alapból, szabályzás nélkül. Ez az 5.23-as ábrán látható is, ha összehasonlítjuk a diagramokat.

A gázmotorok számára pedig 50%, viszont az 1-es id-val rendelkező számára ez felül lett írva 8.000-el. Ezt az 5.24-es ábrán lehet látni.

Ezen kívül az inverter TEST típusának egy re lett állítva az olvasási és parancs reagálási ideje, ez az 5.23-as ábrán jól is látszik, hogy a késleltetés eltűnt.

A konfigurációs DSL pedig a következő:

config {

addDefaultProduceConfig(UnitType.INVERTER, 0.3)

addDefaultProduceConfig(UnitType.ENGINE, 0.5)

addTypeConfig(UnitType.INVERTER, InverterType.TEST){

READ\_FREQUENCY = 1

POWER\_CONTROL\_REACTION\_TIME = 1

TIME\_ACCURACY = 0.0

}

addUnitConfig(UnitType.ENGINE, 1){

addDefVales {

targetOutput = 8\_000.0

}

}

}

# Fejlesztési lehetőségek

A szimulációban még sok fejlesztési lehetőség van, mellyel pontosabb szimulált adatok érthetőek és könnyeb tesztelési lehetőséget ad.

Az első, amellyel pontosabb adatok érhetőek el az időjárás beintegrálása a rendszerben. Ennek megoldására a szimulációba megjelennének az, hogy az inverter milyen egység mellett van, hogy nap elem vagy szélturbina.

A másik ide tartozó fejlesztés, hogy többféle hiba megjelenjen. például, hogy az adott egység nem működik így nem is termel, vagy hogy időjárásból adódóan kisseb hatásfokkal tudnak működni.

Az utolsó fejlesztési lehetőség, amelyet meg is kell majd még valósítanom, az az lesz, hogy a szimuláció nem valós időbe fusson, hanem mondjuk egy napot pár másodperc vagy perc alatt lehessen lefuttatni.

# Összegzés

A feladat során számos értékes tapasztalatot szereztem, amelyek lehetővé tették számomra, hogy elmélyítsem ismereteimet olyan technológiákba, amelyekről az egyetemi tanulmányaim során hallottam, de eddig nem volt lehetőségem mélyebben foglalkozni velük, Például, a JAR fájlok kezelése és a többszálúság alkalmazása Kotlin nyelven.

Ezenkívül új tapasztalat volt az is, hogy nyílt forráskódú keretrendszert használtam, mivel így tudtam tanulmányozni és keresni a belső rendszer működésében.

Valamint maga a feladat is új kihívás elé állított, mivel még nem kellet szimulációkkal foglalkoznom. A megvalósítás közben sokat fejlődött az, hogy hogyan kezdek neki egy ilyen feladatnak és hogyan gondolkozok közben.

Összességében a projekt számos új és értékes tapasztalattal gazdagította a szakmai repertoáromat, és lehetőséget adott arra, hogy szélesebb körű ismereteket szerezhessek a programozás és fejlesztés területén.

1 - [Gyakran ismételt kérdések - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt.](https://www.mavir.hu/web/mavir/gyakran-ismetelt-kerdesek)

2 - [Szabályozási Adatok Kiegyenlítő és Nem Kiegyenlítő Szabályozás céljából - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt.](https://www.mavir.hu/web/mavir/szabalyozasi-adatok-kiegyenlito-es-nem-kiegyenlito-szabalyozas-celjabol) Erőművi termelés primmer források szerinti megoszlás és az import-exposrt szaldó – bruttó üzemirányítási mérések alapján

3 - [partner.mvm.hu/hu-HU/Nagykereskedelem/Merlegkor-menedzsment](https://www.partner.mvm.hu/hu-HU/Nagykereskedelem/Merlegkor-menedzsment)

4 - [Manuális Kiegyenlítő Szabályozási Szolgáltatás (mFRR) | Nano Energies: Tegye eredményesebbé vállalkozást az energia hálózat hatékonyabb kihasználásával](https://nanoenergies.hu/tudastar/manualis-kiegyenlito-szabalyozasi-szolgaltatas-mfrr)

5 - [Szabályozási Adatok Kiegyenlítő és Nem Kiegyenlítő Szabályozás céljából - MAVIR - Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt.](https://www.mavir.hu/web/mavir/szabalyozasi-adatok-kiegyenlito-es-nem-kiegyenlito-szabalyozas-celjabol) – Aktiválás kiegyenlítő szabályzás céljából

6 - [Az inverter jelentése és felhasználási területei (eon.hu)](https://www.eon.hu/hu/blog/napenergia/inverter_jelentese.html)

7 - [Müllberg Speyer - 2 - Power inverter - Wikipedia](https://en.wikipedia.org/wiki/Power_inverter#/media/File:M%C3%BCllberg_Speyer_-_2.JPG)

8 - [Simplex: Load Bank Fundamentals (simplexdirect.com)](https://www.simplexdirect.com/loadbanksEducation.aspx)

9 - <https://en.wikipedia.org/wiki/Simulation>

10 - <https://nanoenergies.hu/tudastar/mari-picasso-es-terre>

11 - <https://en.wikipedia.org/wiki/Continuous_simulation>

12 - <https://en.wikipedia.org/wiki/Digital_twin>

13 - <https://www.techtarget.com/searcherp/definition/digital-twin>

14 - <https://www.twi-global.com/technical-knowledge/faqs/what-is-digital-twin>

15 - <https://www.ibm.com/topics/what-is-a-digital-twin>

16 - <https://en.wikipedia.org/wiki/Kotlin_(programming_language)>

17 – Marcin Moskala: Effective Kotlin BestPractices :: Item 35: Consider defining a DSL for complex object creation

18 - <https://www.baeldung.com/kotlin/extension-methods>

19 - <https://kotlinlang.org/docs/gradle.html#what-s-next>

20 - <https://kotlin.github.io/dataframe/overview.html#what-is-data-frame>

21 - <https://www.kalasim.org/>

22 - <https://www.kalasim.org/theory/> ???

23 - <https://github.com/InsertKoinIO/koin/issues/1369>

24 - <https://github.com/oshai/kotlin-logging>

25 - <https://www.baeldung.com/gradle-fat-jar>

26 - <https://www.unite.ai/hu/mi-az-a-digit%C3%A1lis-iker/>