

TDK-dolgozat

Tóth Bálint

DOMÉNSPECIFIKUS PROGRAMNYELVEK BEÁGYAZÁSA ÁLTALÁNOS CÉLÚ PROGRAMNYELVEKBE EGY GRAFIKONKÉSZÍTŐ JAVA MODUL PÉLDÁJÁN KERESZTÜL

Szerző: Tóth Bálint

Témavezető: Dr. Juhász Zoltán, egyetemi docens

Pannon Egyetem

Műszaki Informatikai Kar, Villamosmérnöki és Információs
Rendszerek Tanszék

Lezárás: 2022. december

Doménspecifikus programnyelvek beágyazása általános célú programnyelvekbe egy grafikonkészítő Java modul példáján keresztül

Tóth Bálint

Mérnökinformatikus BSc.

*Pannon Egyetem, Műszaki Informatikai Kar, Villamosmérnöki és
Információs Rendszerek Tanszék*

Témavezető: Dr. Juhász Zoltán

Napjainkban szinte végtelen számú programnyelv áll rendelkezésre, melyek bármely feladat megoldására alkalmasak. Természetüknél fogva azonban nem rendelkeznek nyelvi elemekkel, melyek lefednék a világon létező minden felhasználási terület (domén) jellemző koncepcióit és műveleteit. A valóság és a program közti absztrakció megértése, kialakítása és folyamatos kezelése a programozó feladata, ami komoly tudást és tapasztalatot igényel. Léteznek emellett speciális programnyelvek (DSL), melyek egy bizonyos domén problémakörére épülnek, így jelentősen leegyszerűsödik az absztrakció a valóság és a program között, azonban ebben az esetben felmerül a keretrendszerek hiánya és programok többi komponenseivel való együttműködés nehézsége. E két eset közötti átmeneti állapot a beágyazott doménspecifikus programnyelv, amely valamely DSL alapján egy általános nyelven készített programozói interfész.

A fejlesztés során egy olyan Java modul készült el, amely képes alapvető grafikontípusok létrehozására és konfigurálható megjelenítésére, elemi komponensként más alkalmazásokba egyszerűen beágyazható, valamint önállóan parancssori környezetből is használható. A grafikonokat konfiguráló interfész egy beágyazott doménspecifikus programnyelvként lett kialakítva a MATLAB programnyelv grafikon készítő alrendszerének programozói interfészéhez hasonló módon a gazdanyelv által biztosított lehetőségekkel, hogy annak használata minél intuitívebb és egyszerűbb legyen.

Tartalom

1. Bevezetés.....	1
2. A beágyazott doménspecifikus nyelvek elméleti háttere	3
2.1. Doménspecifikus programnyelvek.....	3
2.2. Általános módszerek DSL-k beágyazására Java nyelven.....	5
2.3. MATLAB grafikon programozási interfésze	8
3. A beágyazott doménspecifikus nyelv kialakítása.....	10
4. Az alacsonyszintű Java grafikus rendszer implementációja.....	15
4.1. Követelmények.....	15
4.2. Komponensgráf	16
4.3. Keret és tengelyek	18
4.4. A különböző diagramok jellemző grafikus elemei.....	20
5. A beágyazott nyelvet megvalósító interfész.....	22
5.1. Több diagram egy ábrában	22
5.2. Általános formázás	23
5.3. Vonaldiagram	23
5.4. Scatter diagram.....	24
5.5. Oszlopdiaagram.....	24
6. Eredmények.....	25
6.1. Az elkészült diagramok példákkal.....	25
6.2. Shell támogatás.....	31
7. Összegzés	33
8. Irodalomjegyzék.....	34

1. Bevezetés

Az adatvizualizáció minden tudományos folyamat szerves része, napjainkban számos keretrendszer és programkönyvtár támogatja. Az ismertető fejlesztés eredetileg egy EEG jelfeldolgozó rendszerhez készült hagyományos Java könyvtárként indult, speciális grafikonokkal. Ugyan a rendszer elkészítése során alkalmazott JavaFX keretrendszer biztosít beépített grafikonokat, azok formázási és személyre szabási lehetőségei nem bizonyultak megfelelőnek, szükségessé vált egy saját rendszer elkészítése. Emellett komoly érv az egyéni rendszer mellett, hogy a keretrendszer beépített grafikontípusai, valamint a rendelkezésre álló könyvtárak által biztosított lehetőségek főképp önálló grafikus programok és üzleti alkalmazásokra készültek.

A folyamat kezdeti szakaszában eldőlt, hogy az EEG jelfeldolgozáshoz szükséges grafikonok mellett ideális lenne általános grafikontípusok (vonaldiagram, oszlopdiagram) támogatása, így az megnövelné a lehetséges használati esetek számát. A kinézet és a beállítások mintája a MATLAB grafikus rendszere lett, mivel szinte bármilyen grafikon kinézetet el lehet érni vele, kinézetében letisztult és a gyakorlatban általánosan használt szoftver. Az első prototípusok elkészülte után merült fel az igény, hogy az objektum könyvtárát vezérlő API valamelyest hasonló legyen a MATLAB-hoz, hogy koncepciók, formázások könnyebben érthetőek legyenek azon fejlesztők számára, akik nem rendelkeznek komoly Java tapasztalattal, azonban ismerik, netán használták már a MATLAB programot.

Az interfész ilyen irányú fejlesztésekor merült fel a Fluent Interfész használatának lehetősége. Miután a Java programnyelvet általános célokra fejlesztették ki, nem feltétlen teszi lehetővé, hogy minden felhasználási területet lefedő kulcsszavakkal, szintaxissal és műveletekkel rendelkezzen. A nyelv objektum orientált, amelyben a valódi világot objektumok és azok közti kapcsolatok írják le, melynek segítségével tetszőleges koncepció modellezhető. Ugyan az absztrakció lehetővé teszi a grafikonok objektumként való létrehozását és kezelését, sokkal intuitívabb lehetne, ha a grafikonokat valóban grafikonokra jellemző módon lehetne leírni. Erre a célra lenne ideális egy doménspecifikus programnyelv, amely kizárólag az adott felhasználási területen, ez esetben grafikonok formázásában, lenne használható, azonban hiányozna belőle minden olyan interaktív tulajdonság, ami egy Turing-teljes programnyelvben megtalálható.

Egy doménspecifikus programnyelv (DSL) kifejlesztése a jelen esetben nem volt célravezető, mivel egyrészt a meglevő funkcionalitást biztosító kóddal kompatibilisnek kell, hogy legyen, másrészt továbbra sem oldja meg az egyszerűen használható interfész problémáját sem, miután egy külön rendszer elkészítését eredményezné. A harmadik jelentős indok egy ilyen nyelv létrehozása ellen az egyszerű tény, hogy ez a nyelv létezik, a fent említett MATLAB rendszer scriptnyelve. A teljesen független leíró programnyelv és az objektum orientált API közötti átmeneti állapot a belső doménspecifikus nyelv, amely egy olyan interfész, melynek függvényei egy DSL kulcsszóit imitálják.

A beágyazott doménspecifikus nyelv koncepciója viszonylag régóta él a köztudatban, azonban viszonylag kevés jelentős példát látni rá ismertebb kódbázisokban. Ennek oka főképp az beágyazott doménspecifikus nyelvek implementációs nehézségei, valamint az azt megvalósító kód karbantartásának nehézsége.

A következőkben a grafikonkészítő könyvtár példáján keresztül demonstrálom az interfész kialakításának lépései során alkalmazott módszereit, a mögöttes grafikai rendszert, a kész rendszer használatát, valamint az egyezőségeket a MATLAB programnyelvvvel.

2. A beágyazott doménspecifikus nyelvek elméleti háttere

2.1. Doménspecifikus programnyelvek

Azokat a programnyelveket, melyek nem általános célra, hanem egy felhasználási terület számítási feladatainak megoldására hozták létre, doménspecifikus programnyelveknek (Domain-Specific Language - DSL) nevezzük. Szintaxisuk és sok esetben szemantikájuk is a doménjük sajátosságait tükrözik, alapvető elemei a szoftverfejlesztésnek. Létrehozásuk mögötti fő motiváció a valós világ és az általános célú programnyelvek közti absztrakciós folyamat egyszerűbbé tétele. Napjainkban az objektum orientált tervezés játssza a fő szerepet a felhasználási területből a feladat objektum orientált programnyelvre történő leképezésére, azonban hosszadalmas folyamat és komoly szoftverfejlesztői tudást igényel, ami a további fejleszthetőséget nem feltétlen segíti. Ezzel szemben egy DSL áthidalja a szakadékot egy tudományterület zsargonja és egy tetszőleges objektum orientált programnyelv között, miután annak kulcsszói az adott doménből származnak.

Doménspecifikus programnyelvek két csoportját különböztetjük meg: külső és belső. A külső DSL-k saját, jól definiált szintaxissal bírnak és valamely más nyelven írt fordítóprogrammal, vagy értelmezőprogrammal használhatók. Ilyen nyelvek az SQL, HTML, CSS stb., melyek adatbázis lekérdezésekre, webfejlesztésre használhatók, azonban általános célú programot semmilyen, vagy túlzottan komplikált módon lehet velük készíteni. Másik csoportjuk a belső (Internal DSL - IDSL), vagy beágyazott DSL-ek (Embedded DSL - EDSL), melyek egy gazdanyelvbe kerülnek beépítésbe a nyelv által biztosított lehetőségekkel. Ezek a nyelvek általában egy API formájában jelennek meg, ezért szokás őket a Fluent Interfész néven is említeni [1]. A későbbiekben főképp ezekről a beágyazott nyelvekről lesz szó.

A beágyazott DSL-k kialakításuk kezdeti nehézségeitől eltekintve számos előnnyel bírnak a hagyományos doménspecifikus nyelvekkel szemben. Egy programnyelv megtervezése és a hozzá tartozó fordító, vagy értelmező készítése jelentősen több időt, erőforrást és személyes tudást igényel, mint egy interfész megtervezése, azonban a szintaxis minden eleme fölött teljes kontrollja van a fejlesztőnek. Egy jól kivitelezett DSL a kifejlesztése után azonban sokkal könnyebbé teszi egy célszoftver elkészítését. A Fluent Interfészek esetében azonban jelen van az állandó korlátozó tényező, hogy a gazdanyelv szintaxisával kompatibilis kell, hogy legyen. Erre a problémára lehetséges megoldás a programnyelv fordítójának kiegészítése, ezt számos nyelv támogatja előfeldolgozó direktívákkal, annotációkkal és interpreter bővítményekkel. Ebben az esetben azonban a kliens oldali program fejlesztőinek is folyamatosan rendelkezniük kell a különféle bővítményekkel és azok frissítéseivel. (Cseppet sem ideális.) Egyes modern, illetve kísérleti fázisban lévő programnyelvek makró szemantikája lehetőséget biztosít az Absztrakt Szintaxisfa direkt eléréséhez (Rust [2]), melyekkel a gazdanyelv viselkedése tetszőlegesen megváltoztatható. Egyéb programnyelvek minimalista szintaxisukkal könnyítik meg az EDSL fejlesztést. Példának okáért a JVM alapú Groovy

programnyelvben a függvényhíváskor a paraméterek zárójelbe tétele opcionális, így mondatszerű módon írhatók utasítások [3]. Könnyebben olvasható interfész alakítható ki olyan nyelvekben is, amelyek támogatják az operátorok túlterhelését (C++, C#, Object Pascal), esetleg lehetővé teszik egyéni operátorok létrehozását (Haskell). A saját objektumokon értelmezett operátorok segítségével könnyebb elkerülni a függvények paraméterezéséből fakadó emberi hibákat.

A LISP programnyelv 1958-ban jelent meg, a második magasszintű programnyelv a FORTRAN után. Jellegzetes S-kifejezés alapú szintaxisa, makrói, a program adatként kezelésének lehetősége, valamint a definiálható kulcsszói kiváló platformot biztosít doménspecifikus nyelvek beágyazására. A Common LISP dialektus biztosít is példát erre a beépített loop makróval, ami lehetővé az alapvetően funkcionális nyelvben az imperatív nyelvekből ismert különféle ciklusfajták használatát [4].

Az alábbi példa a Common LISP loop makróját illusztrálja, a for szintaxis ugyan nem része a nyelvnek, azonban a loop makró értelmezhető LISP kóddá alakítja, így a művelet pontosan úgy fog lefutni, mint egy számláló ciklus.

```
(loop
  for i from 0 to 10 do
    (some-operation i))
```

A nyelv szintaxisának alapját a listák adják (innen a név: LIST Processor), ezeket a listákat a nyelv képes adatszerkezetként kezelni és bizonyos elemeit kiértékelni, többi elemét azokhoz a műveletekhez adatként használni. Ezeket a makrókat fordítás, vagy feldolgozás előtt létező nyelvi elemekké konvertálja, amelyeket a feldolgozóprogram már képes kezelni.

Általános célú programnyelvekbe beágyazott doménspecifikus nyelvek nem minden esetben gondos tervezés és körülményes implementáció eredményei. A C++ programnyelv sablonosztály-rendszere a rekurzív sablon specializáció következtében véletlenül egy Turing-teljes beépített programnyelvet alkot [5]. A sablonosztály a C++ egy alapvető mechanizmusa, melynek segítségével olyan struktúrák hozhatók létre, amelyekben bizonyos elemek típusa paraméteres, azt példányosítás esetén a fejlesztőnek kell megadnia. Ezeket a típusokat a fordítóprogram kiértékeli, futási időben a típusok nem változhatnak, ezért statikus linkelésnek, vagy sablon polimorfizmusnak nevezik. A C++ programnyelv lehetőséget biztosít sablonosztályok általános paraméteres esete mellett specializált esetek megadására, abban az esetben a fordítóprogram példányosításkor a megfelelő specializációt használja. A DSL működését az a kulcsfontosságú tervezési döntés adja, hogy a sablonparaméterek bármilyen típusúak lehetnek (típusnév helyett például egész szám), így a specializációk keresése közben a fordítóprogram fordítási időben rekurzívan előállíthat köztes struktúrákat, így gyakorlatilag egy funkcionális paradigmának megfelelő nyelv lehetséges utasításait hajtja végre. Ezt a jelenséget C++ Template Metaprogram néven jegyzik és köztes adatszerkezetek generálásához, valamint olyan fordítási idejű számításokhoz használják, amelyeket a C++ makró rendszere és fordítási idejű függvényei nem

tesznek lehetővé [5]. Az ezen a „nyelven” írt programok megértése, tervezése és karbantartása jelentősen nehezebb, mivel a függvények sablonos struktúrák formájában vannak jelen a programban.

Az alábbi példakód egész számok faktoriálisának meghatározásához használható. A sablon struktúra szám paramétere az így keletkezett „függvény” bemeneti paramétere, a struktúra adattagja pedig a bármikor kiolvasható visszatérési érték.

```
#include <iostream>

template <int n>
struct factorial
{
    static const int value = factorial<n-1>::value * n;
};

template <>
struct factorial<0>
{
    static const int value = 1;
};

auto main() -> int
{
    std::cout << factorial<5>::value << std::endl; // 120
}
```

2.2. Általános módszerek DSL-k beágyazására Java nyelven

Miután a később tárgyalt fejlesztés egy Java könyvtár programozói interfészének különféle eszközökkel történő feljavítását foglalja magába, ismertetek néhány általánosan alkalmazott Java specifikus módszert doménspecifikus nyelvek beágyazására, illetve ahhoz hasonló interfészek kialakítására. Ezek a módszerek ugyan emberileg könnyebben olvasható kódot eredményeznek, azonban számos, a Java fejlesztők által irányadónak tartott stilisztikai konvenciót megszegnek, függvények és osztályok elnevezésével. Az alábbi módszerek bizonyos része bármely magas szintű objektum orientált programnyelv esetén alkalmazható.

A Metóduslánc (Method Chain) a legalapvetőbb módszer, objektumok módosító függvényeinél alkalmazható, a közismert tervezési mintaként ismert Builder pattern esetében is ezt a módszer javasolt [6]. Lényege a módosított objektum példányának visszatérítése, amelyre a Java nyelv a beépített `this` kulcsszót használja. Ezzel a módszerrel egy objektumra meghívott függvények egymás után szerepelnek, nem szükséges folyamatosan kiírni a változó nevét, mint a hagyományos objektum orientált kód esetén.

Az alábbi példakódban egy objektum létrehozása és két adattag beállítása látható hagyományos Java objektum orientált programkód esetén:

```
Person p = new Person();  
p.setName("Steve");  
p.setAge(21);
```

A metóduslánc kialakítható olyan módon, hogy logikailag összefüggő metódusok hívhatók legyenek egymás után [7], így már egy kezdetleges DSL-re hasonlító interfész is kialakítható, illetve eddig a pontig, mint hasznos gyakorlat, szinte bármilyen Java projektben használható. A metódusok nevei megválaszthatók a beállított paraméter nevének, esetleg a kialakítandó nyelv kulcsszóinak. Ezt a Java konvencióknak megfelelő kód `get` és `set` kezdetű metódusokkal végezne, azonban az egymás után láncolás egyértelműsíti az adatbeállítási, vagy lekérési szándékot, így az interfész tisztasága végett az előtagokat rendszerint elhagyják.

A Builder tervezési minta alkalmazásával és annak metódusláncának segítségével a következőképpen írható át az előbbi példa:

```
Person p = new Person.Builder()  
    .name("Steve")  
    .age(21)  
    .build();
```

Java nyelven bármely típusú objektum létrehozásához a `new` kulcsszó használható, a dinamikus memóriakezelés miatt manuális törlés nem szükséges. Amennyiben egy interfészt úgy szeretnénk kialakítani, hogy a létrehozási logikában a kulcsszó ne látszódjon, a Factory tervezési minta és annak különféle változatainak használata ajánlott [7]. A Factory minta alapja a konstrukciós folyamat kiszervezése valamely osztályba, azonban Fluent interfészek esetén előfordulhat, hogy az adott osztály tartalmazza azt a statikus függvényt, ami a létrehozásáért felelős. Ez a „factory metódus” felülírható, így több változat definiálható hozzá.

A konstruktor elrejtésével és a statikus létrehozófüggvénnyel teljesen eltüntethető a gazdanyelv létrehozási szintaktikája. Java nyelven a következő példán látható módon működik:

```
Address a = Address.address(8200, "Egyetem utca 12.");
```

A Java programnyelv lehetőséget ad osztályok statikus függvényeinek nevének importálására, ekkor az adott függvényt az osztálynév nélkül is használhatjuk. Komoly előnye, hogy lerövidíti a függvényhívást, azonban több osztálynak megegyező nevű metódusát ezzel a statikus importálás módszerrel nem lehet végrehajtani.

Metódusláncokkal drasztikusan tömöríthetők programkódok, azonban azok paraméterezése okozhat nehézségeket. Ideális lenne a paraméterezést is könnyen olvasható metódusfüzerekkel elvégezni, ehhez azonban szükségesek köztes osztályok és azok objektumai. Ez a módszer jelentősen

hozzájárul az így kialakított interfész komplexitásához, miután az összetett paraméterrel rendelkező függvények esetén szükségessé válik egy új osztály létrehozása.

A korábbi statikus függvény importjának köszönhetően nem szükséges a létrehozó metódus elé kiírni az osztály nevét, azonban a paraméterek továbbra is tisztázatlanok:

```
Time t = time(13,45,12);
```

Erre megoldásul bevezethető egy olyan köztes objektum, amely a `hour()` indítófüggvénnyel hozható létre, az interfész tükrözi a paramétereket az alábbi példa alapján:

```
Time t = time(hour(13).minute(45).second(12));
```

Köztes osztályok használhatóak visszatérési érték formájában is. Amennyiben a `this` objektum helyett egy másik köztes példányt adunk vissza, az interfészt használó programozót kötelezhetjük valamiféle üzleti logikai, vagy DSL szintaktikai szabály betartására [7]. Másrésről komoly előnye, hogy amennyiben a köztes objektumot absztrakt osztályként vagy interfészként definiáljuk, biztosítható magasabb szintű modularitás. Emellett az interfész megoldás esetén hierarchia is könnyebben kialakítható, ami az általános szintaxisú EDSL-k esetén jobb absztrakciós lehetőségeket biztosít a szintaxis definiálásához.

A Java SE nyolcadik verziójától kezdve elérhető a nyelvben, lehetőséget biztosít anonim függvények paraméterként történő megadására és kiválóan alkalmazható Fuent Interfészek belső adatszerkezeteinek módosítására. A lambda függvények lehetővé teszik, hogy a korábban függvénylánccal létre hozott és konfigurált belső objektumot módosítsuk. Ez úgy hajtható végre, hogy az interfész valamely függvénye paraméterül vár egy lambda kifejezést, melynek paraméterül szolgáltatja a módosítandó objektumot. A lambdán belül az objektum tetszőleges módon, akár egy újabb függvénylánccal módosítható.

Amennyiben a beágyazni kívánt DSL segítségével bonyolultabb műveleteket szeretnénk végezni, célszerű a szintaxist egy interfészrendszerrel definiálni, a mögöttes logikát pedig a szintaxis objektumain értelmezve automatizáltan végrehajtani. Ezzel elkerülhető, hogy az interfész és az implementáció túlzottan egymásba legyen ágyazva, mivel az minden szempontból előnytelen a továbbfejlesztésre nézve. Ebben az esetben az értelmező réteg a Builder tervezési minta alapján kialakítható és a függvénylánc paramétereivel üzemeltethető [8]. Erre a műveletre nem minden esetben van szükség, mivel a beágyazott nyelvek általában deklaratívak, míg a legtöbbször használt gazdanyelvek imperatívak [8].

A Java SE tizedik verziójától elérhető a `var` kulcsszó, amelynek segítségével nem szükséges a programkódban megadni egy változó típusát, amennyiben az adott értékadási kifejezésből a fordítóprogram egyértelműen meg tudja állapítani a változó típusát. Általános esetben használható

nyelvi elem, azonban érdemes EDSL-k tervezésekor figyelembe venni, miután teljes mértékben eliminálhatja a nyelv implementációja során létrehozott köztes osztályok nevének kiírását.

2.3. MATLAB grafikon programozási interfésze

A MATLAB rendszert kiterjedt használata miatt választottuk a grafikonkezelő programunk mintapéldájának, mind a készítendő ábrák kinézete, mind az azokat létrehozó API tekintetében. A grafikus rendszerben minden diagramtípusra szerepel egy-egy függvény, azok különböző alváltozataiból állíthatók elő tetszőlegesen formázott ábrák. Miután a nyelv típusrendszere nem kötött, a függvények paramétereinek sorrendje alapján az feldolgozóprogram futás közben választja ki a megfelelő függvényváltozatot. A diagramtípusok közül a legtöbbet használt példa a vonaldiagram, annak példáján mutatom be a rendszer működését.

Vonaldiagram a `plot()` függvénnyel és annak változataival hozható létre. Első két paraméter minden esetben az értelmezési tartomány értelmezési tartományát és értékkészletét tartalmazó vektorok, utánuk opcionális formázó paraméterek következnek. A formázó paraméterek megadása szöveges kulcsokkal, utánuk a paraméter értékével történik, ezt a nyelv dinamikus típusrendszere teszi lehetővé, akár csak azt, hogy több vonal is megadható legyen, ugyanis a paraméterlistában felsorolva két egymást követő vektor és az utána következő formázószöveg egy újabb vonalat definiál. Fontos megjegyezni, hogy ebben az esetben bármely kulcs-érték páros paraméter minden abban a függvényben definiált vonalra vonatkozik.

A diagram minden része paraméteresen módosítható, vonaldiagram esetén megadható a vonal színe, vastagsága, stílusa, valamint az adatpontokat jelölő markerek típusa és színe. Típus, szín és marker egy szintaktikai rövidítésen keresztül megadható minden vonal után. Ebben az esetben a formázószövegben tetszőlegesen felsorolható a különböző paraméterek rövidítése. Példa: a `'--or'` formázás szaggatott vonalú, kör markerekkel piros színű vonalnak fele meg. A színek angol nevükkel, vagy annak kezdőbetűjével adhatók meg, a leggyakrabban használt vonalstílus és marker formázó karaktereket az alábbi táblázatok tartalmazzák:

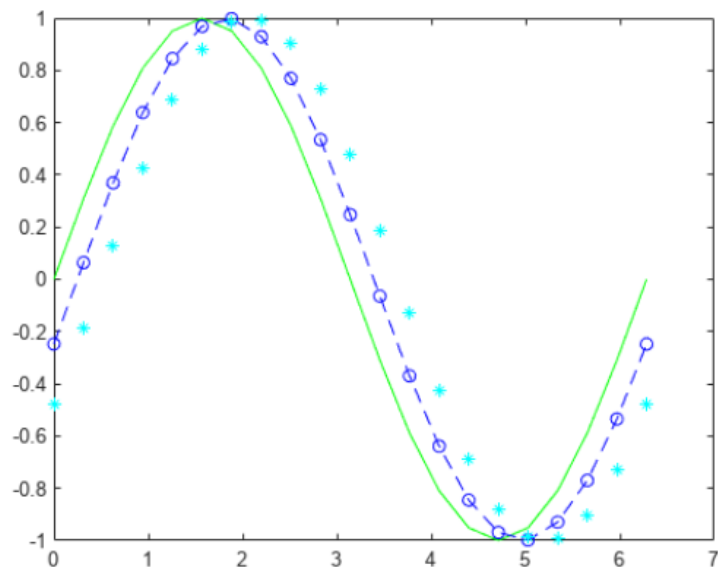
-	normál vonal (alapértelmezett érték)
--	szaggatott vonal
:	pontozott vonal
-.	pont-vonal vonal

'o'	kör
'+'	pluszjel
'*'	csillag
'x'	kereszt
'.'	pont

Formázószövegek minden vonal esetén megadhatók az értelmezési tartomány és értékkészlet után a következő módon:

```
plot(x,y1,'g',x,y2,'b--o',x,y3,'c*')
```

A példakód egy vonaldiagramot generál három adatsorral, az elsőt a formázó zöld színűre állítja, a második vonalat szaggatott kék színűre, a harmadik vonal esetén pedig kikapcsolja a vonalat és csak a markert jeleníti meg. Az eredmény az alábbi ábrán látható:



1. ábra A példakód által generált MATLAB grafikon

3. A beágyazott doménspecifikus nyelv kialakítása

A következőkben az elkészült interfész megtervezését szeretném bemutatni, hogyan jutottam el a közismert Builder tervezési mintából kiindulva fokozatos fejlesztéssel a grafikonokat vezérlő EDSL-ig. Alapjaiban véve a Builder tervezési minta a fluent interfész egy alkalmazási területe, lényegét tekintve összetett objektumok létrehozásának egy módja. Alapja a létrehozási logika konstruktorból való kiszervezése, valamint az egyes adattagok egyéni függvényekkel való beállítása. Ezzel arra a problémára hivatott megoldást adni, miszerint egy konstruktor-függvényhívásban az előre megadott paramétereket az előre megadott sorrendben kell a paraméterlistában feltüntetni. Ebből adódik, hogy minden beállítandó adattagot szerepeltetni kell benne és az egyébiránt alapértelmezett értékkel ellátható adattagokat is be kell állítani, ellenkező esetben a program nem fog lefordulni. Emellett a paraméterek megvizsgálása, a létrehozási logika is nehezen lesz skálázható, mert mindennek egy függvényben kéne szerepelnie. A Builder tervezési minta segítségével a paramétervizsgálat minden esetben az adattagot beállító függvényben megvalósítható, a logika pedig az építőmetódusban szerepelhet.

Míg egy általános célú fluent interfész és egy Builder első ránézésre nagyon is hasonlít egymásra, a felhasználásuk célja és környezete is jelentősen eltér egymástól. A Builder célja objektumok létrehozása, ezt neve is sugallja, de hogy működése után az általa létrehozott objektumból mi lesz, már nem tartozik a hatáskörébe. A fluent interfész, bővebb kategória lévén, egyéb célokra is felhasználható, de elsősorban objektumok konfigurálása a fő használati területe. Itt megjegyeznénk, hogy a létrehozás és a konfigurálás két külön tevékenységnek tekintendő. A létrehozás során a Builder egy köztes adattároló osztály példányát tárolja, általában a sajátját, és egy másik célosztály példányát hozza létre (erre szolgál az általában `build()`-nek nevezett metódus). Ennek megfelelően a Builder lefutása után egy elemi művelettel létrehozott példány keletkezik egy másik osztálynak. Fluent interfész az adott osztályhoz készül, ezért a példányainak is állandóan változtathatóknak kell lenniük. Pontosan evégett nem lehetséges futás közben paramétervizsgálatot csinálni egy általános fluent interfészen belül, mivel a metódusai bármilyen sorrendben meghívhatóak, lehetetlen függvényen belüli validációt végezni. Ez a konvencionális, függvényláncot lezáró függvény hiánya miatt lép fel, ugyanis ez a függvény valósítja meg a Builder tervezési mintánál tárgyalt konstrukciós logikát. Mivel abban az esetben minden művelet az utolsó függvényben kerül implementálásra, akkor lehetséges kötelező paraméterek meglétének vizsgálata, vagy a megadott értékek hitelessége. A fluent interfész mellőzi ezt a függvényt, ettől nem csak szabadabb lesz a felhasználási módja, de semmi nem garantálja, hogy egy függvény meghívásakor annak futásának összes előfeltétele teljesült már. Ez a jelenség nehezen kivédhető, nem megfelelő használat esetén futási hibákhoz vezet.

Az alábbi MATLAB kód kulcs érték párokkal állít be formázási paramétereket egy vonaldiagramnak. Ez a kód szolgál alapul a későbbi interfész kifejlesztéséhez, a továbbiakban minden megvalósításnak ehhez kell igazodnia:

```
figure;
plot(xs,ys,'LineStyle','-','LineWidth',2,'Color','red');
```

A megfelelő funkcionalitás objektum orientált modellezésével meghatározhatók osztályok, melyek a szükséges adatokat tárolják, valamint azok kapcsolata. Minden lehetséges paraméter szem előtt tartása esetén a következő interfész alakítható ki:

```
Figure figure = new Figure();
// xs, ys : double[]
figure.add(new Plot(new Line(xs,ys,new LineStyle(StrokeStyle.NORMAL, 2,
                                                    Color.RED))));
figure.display();
```

A megoldás teljes mértékben igazodik az objektum orientált programozás elveinek, minden paraméter a megfelelő reprezentációs osztály egy példánya, vagy primitív adattípus. Az interfész könnyebb kezelése érdekében a felsorolt típusok helyettesíthetők azok szöveges megfelelőivel a következő módon:

```
Figure figure = new Figure();
// xs, ys : double[]
figure.add(new Plot(new Line(xs,ys,new LineStyle("-",2,"red"))));
figure.display();
```

Ez a megoldás az objektum orientált programozás elveinek megfelel, azonban ismét felmerül a probléma, hogy első ránézésre nehezen állapítható meg, hogy a konstruktorok paraméterei pontosan mik. Példának okáért feltételezhető, hogy a `LineStyle` konstruktorban a "red" a vonal színét jelenti, de a második paraméter a dokumentáció állandó szem előtt tartása nélkül csak egy értelmezhetetlen kettes szám. Ez a programrészlet adja az alapját a megvalósítandó grafikonrajzoló könyvtárnak, a cél ennek továbbfejlesztése egy Beágyazott Doménspecifikus programnyelv irányába.

A továbbfejlesztés első lépése a `Plot` osztály konstruktorának értelmezhetővé tétele és a későbbi konstrukciós logika kiszervezése a saját metódusába.

```
Figure figure = new Figure();
figure.add(new PlotBuilder()
    .withXs(xs)
    .withYs(ys)
    .withStyle(new LineStyleBuilder()
        .withStyle("-")
        .withWidth(2)
        .withColor("red")
        .build()
    ).build()
);
figure.display();
```

Ez a megoldási már lehetővé teszi, hogy a programozó (aki ebben az esetben a végfelhasználó is) függvényekkel állítsa be a létrehozandó grafikon paramétereit, de még mindig egymásba ágyazott zárójelezett blokkok alkotják a létrehozó függvényhívást.

Ezt kiküszöbölendő, a függvények visszatérési értékeinek módosításával és köztes adattípusok bevezetésével létrehozható egy olyan megoldás, ami teljesen kizárja a paraméterlistán belüli függvényhívást. Erre egy lehetséges példa:

```
Figure figure = new Figure()
    .withPlot()
    .withLine()
        .withXs(xs)
        .withXs(ys)
        .withStyle()
            .withStyle("-")
            .withWidth(2)
            .withColor("red")
        .addToLine()
    .addToPlot()
.addToFigure()
.display();
```

A fenti példakód a fluent interfészt használja, minden függvény egymás után hívható, így kiküszöböli az egymásba ágyazott zárójelek adta problémát, azonban felvet számos másikat. Elsősorban az `addToLine()`, `addToPlot()` és `addToFigure()` függvényeket szükséges megvizsgálni. Ezek a függvények kód szintjén felelősek a visszatérési típusok egyezéseért. A megoldás lényegében a korábban ismertetett Builder olyan átírása, hogy a függvények a következő Builder típusát adják vissza, így zárójeles blokk nélkül lehet egyik osztályból egy abba beágyazott osztályba lépni. Felmerül ilyenkor a probléma, hogy miként lehet lezárni egy Builder-t és visszalépni az eredeti hívó osztályba. Ennek megfelelően a beágyazott osztályok tárolnak egy referenciát arról az osztályról, ami létrehozta őket és az `addTo...` függvények ezt a példányt adják vissza, hogy tovább láncolható legyen a függvény. Ez jelentősen megnehezíti a köztes objektumok és Bilder-ek több területen való felhasználását, mivel egy adott típusról tudnak csak referenciát tárolni. Másodsorban a függvények nagyszámú egymás után láncolása eredményezhet átláthatatlan, nehezen azonosítható futási hibákat, mivel abban az esetben a kivételt a futási környezet végig terjeszti az összes függvényen, és a stack trace alapján csak hosszas értelmezés után lehetne azonosítani és javítani a hibát. Egy ilyen jelenség komoly hátulütő lehet egy könyvtár felhasználói megítélésekor.

Ezek után a lehetséges objektum orientált megoldások után vizsgáljuk meg az alapprobléma átültetését egy DSL-be. Szerencsére ez nem bonyolult, mivel a MATLAB programnyelv fejlesztői már megtették helyettünk. A MATLAB programnyelv két módot ad tulajdonságok beállítására. Az első a szöveges típusú kulcsok után a megfelelő típusú érték megadása. Ez a megoldás egyszerűen kivitelezhető, mivel a nyelv nem típusos, a függvények paramétereit nem kötik. A második lehetőség egy rövidített formázólista megadása, mely a parancsértelmező által kerül feldolgozásra.

Ezek alapján a cél olyan osztályok és metódusok létrehozása, amik minél jobban utánozzák az eredeti MATLAB kódot, emellett logikus felépítésű programot eredményez. A korábban áttekintett módszerek alkalmazásával létrehozható egy olyan köztes osztály, amely statikus indítófüggvénnyel konstruálható, valamint a paraméterek karakteresen beállíthatók:

```
figure()
    .plot(line(xs,ys,style("-").width(2).color("red")))
    .display();
```

Ebben a megoldásban a `LineStyle`, mint vonalstílus leíró látszólag nincs is jelen, mert a `style()` függvénnyel került létrehozásra. Ebben az esetben köztes objektum átadása történik a háttérben, a végfelhasználó elől azonban rejtve van. Ez a módszer a Static Factory Method, lényege a konstruktor lecserélése egy statikus függvényre, ami létrehoz egy objektumot és visszatér vele. Ahhoz, hogy ebben a formában működjék, szükséges a megvalósító nyelvben a statikus import létezése. (Példa: Java programnyelv, `import static package.LineStyle.*`;) Amennyiben a gazdanyelv ezt nem támogatja, szükséges az osztály megadása is:

```
...
    .plot(LineData.line(xs,ys,LineStyle.style("-").width(2).color("red")))
    ...
```

A MATLAB specifikációja szerint lehetséges a paraméterek megadása kulcs-érték párokkal. Mivel a MATLAB nem típusos, esetünkben azonban egy szigorúan típusos programnyelvben történő megvalósítás a feladat, a paraméterezést is típusosan kell megoldani. Ennek következménye, hogy minden formázóparamétert egységesen, `String` típussal fog kezelni a program. A kulcs érték párok egy darab paraméterben fognak szerepelni egy előre definiált karakterrel elválasztva.

```
Figure figure = figure()
    .plot(line(xs,ys,"Style:-","Width;2","Color;red"))
    .display();
```

Ebben az esetben minden paraméter egy `String` típusú "KULCS;ÉRTÉK" párban kerül bevitelre, így a programkód érthetőbb, tisztább lett. A fenti módszer azonban számos problémát vet fel, amik a programozói felületet nem befolyásolják, annál inkább a működést. Elsősorban a paraméterek visszaalakítása egy kisebb fordítót igényel, ami szétválasztja a kulcs-érték párt és egy attribútummá alakítja. A problémát a típusok feletti fejlesztői kontroll elvesztése jelenti. Amíg függvények kerülnek alkalmazásra a paraméterek megadásakor, hibás típusú paraméter megadásakor a fordítóprogram észleli és a program nem fordul le. Minthogy ebben a verzióban minden paraméter `String`, semmi lehetőség nincs ellenőrizni, hogy egy kulcs olyan értéket kap, ami értelmezhető a program által. Ez nagyban megnöveli a futási hibák valószínűségét, mivel minden lehetséges programozói hiba akkor a futtatási környezetben fog megjelenni.

A natív JavaFX integrálhatóság okán a `display()` függvény elhagyható. Ilyenkor az objektumra meghívott `show()` függvény használható. Ebben az esetben létrejön egy ablak, amiben megjelenik a kívánt grafikon.

```
Figure figure = figure()
    .plot(line(xs,ys,style(":").color("#ff0000")));
figure.show();
```

A fenti rendszerrel kezelhető a teljes grafikon, illetve a vonaldiagram egyes vonalai, azonban a legtöbb esetben a grafikon egészére is szükségesek beállítások, amiket nem lenne célszerű a vonalakhoz rendelni. Ebben az esetben a vonal interfészéhez hasonlóan egy köztes objektum alkalmazható a vonaldiagram személyre szabásához. Ennek a köztes objektumnak szintúgy definiálható statikus indítófüggvény és fluent interfész, így beépíthetővé válik a meglevő rendszerbe. Ezzel az elemmel általános eset építhető fel, amely minden későbbi grafikontípusra is működik, három szinten biztosít lehetőséget a teljeskörű konfigurációra az ablaktól a grafikonon át az egyes adatsorokig. Az interfész emellett tükrözi az elemek logikai egymásba ágyazottságát, mivel a függvények paramétereiben meghívott köztes objektumok megfeleltethetők a grafikon egyes részeinek. A végső, általános eset a következőképpen néz ki az előző példák alapján:

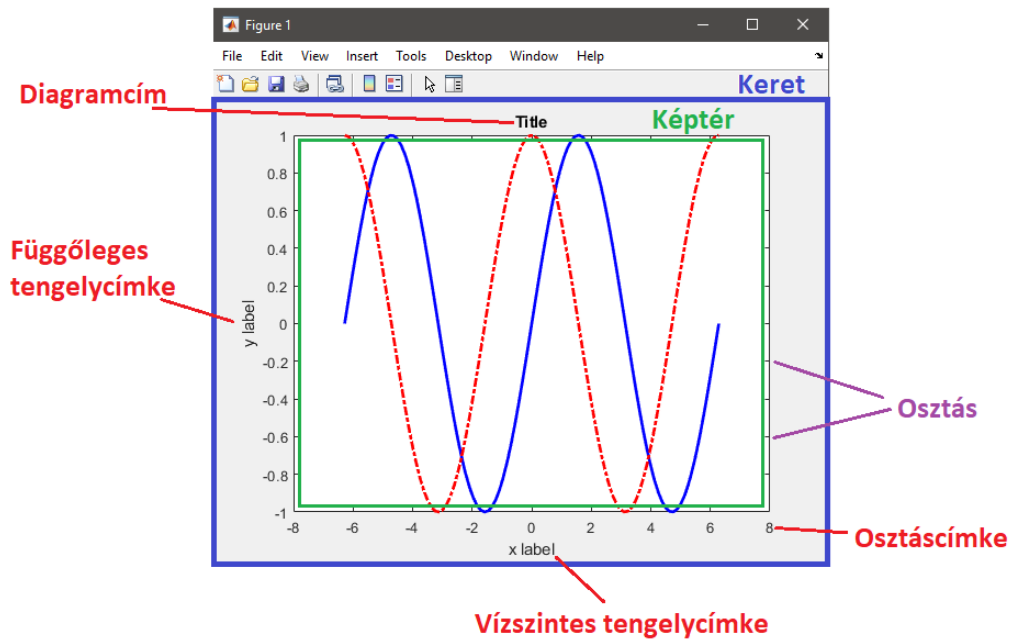
```
Figure figure = figure()
    .plot(title("test").grid("on"),line(xs,ys,style(":").color("#ff0000")));
figure.show();
```

A `plot()` függvényhívásba beépített átmeneti `FrameStyle` objektum (`title()` indítófüggvénnyel) beállítja a grafikon címét és bekapcsolja a rácsvonalakat. Mivel a `FrameStyle` semmi grafikon specifikus dolgot nem csinál, a későbbiekben bármely más grafikonnal használható, ami drasztikusan leegyszerűsíti az interfész használatának elsajátítását, továbbá a kód duplikálását is kiiktatja, mivel a funkcionalitást nem szükséges minden grafikon esetén implementálni. Ezzel a megoldással elhelyezhető az ablakba akár több grafikon is, azok beállításai nem ütköznek egymással. Ezek alapján az elvek alapján terveztem meg azt az interfészt, amelynek segítségével egyetlen függvénylánc segítségével tetszőlegesen komplikált grafikon ablak létrehozható.

4. Az alacsonyszintű Java grafikus rendszer implementációja

4.1. Követelmények

A grafikus rendszer megtervezése során a MATLAB konformitás elérése érdekében szükségesnek tartottuk minden paraméterezhető tulajdonság megtartását. Az alábbi ábrán láthatók egy grafikon főbb részei, melyek elhelyezkedését és méretezését automatikus, vagy egyéni beállítások alapján a rendszer kezeli.



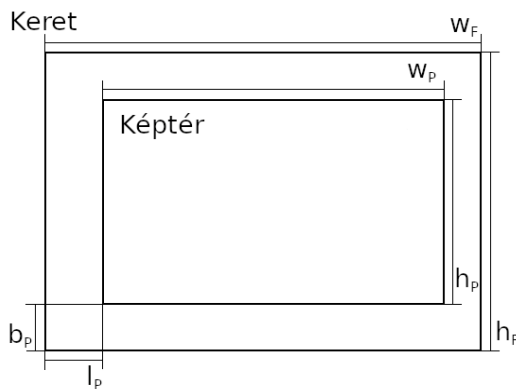
2. ábra Egy MATLAB grafikon fő alkotóelemei

Az általános grafikon fő részei:

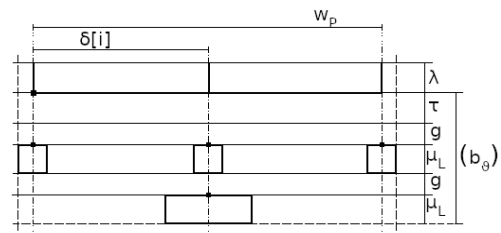
- Keret: Magába foglalja a tengelyt és minden címkéjét.
- Grafikoncím: A grafikon felhasználó által megadható címe.
- Osztás: A tengely alapegyenesére merőleges szakasz.
- Osztáscímke: Az osztás értékét jelölő címke.
- Vízszintes tengelycímke: A vízszintes tengely felirata.
- Függőleges tengelycímke: A függőleges tengely felirata.
- Képtér: A tengely által közrefogott terület, a különféle grafikonok vizuális elemei ebben a térben jelennek meg.

A keret magába foglalja az összes megjelenítendő grafikus elemet és automatikusan kitölti a rendelkezésre álló helyet. A specifikáció során a mértékek $[0,1]$ intervallumon szerepelnek, hogy az ablak átméretezése esetén az arányok nem változnak. Az arányos értékek abszolút, ablakon belüli pozíciója az arány ablakszélességgel, illetve hosszúsággal történő megszorzásával kapható meg.

A keret átméretezésekor a pozíciók normál mennyiséggel történő megadása miatt a távolságarányok mindig megegyeznek. Amennyiben az ablak összenyomásra kerül, a befoglaló téglalap méretéig a grafikus elemek pozíciója arányosan csökken. Amennyiben az ablak mérete ennél is kisebb mértékűre csökken, a képtér összenyomódik. Függőleges irányban a keret aljától mért távolság egyenlő lesz a befoglaló téglalap alsó egységével, a magasság pedig a befoglaló téglalap által engedett legnagyobb értékben maximalizálódik. Vízszintes irányban a keret bal oldalától mért távolság egyenlő lesz a befoglaló téglalap bal oldali egységével, a szélesség pedig a befoglaló téglalap által engedett legnagyobb értékben maximalizálódik.



3. ábra A grafikon elhelyezésének paraméterei



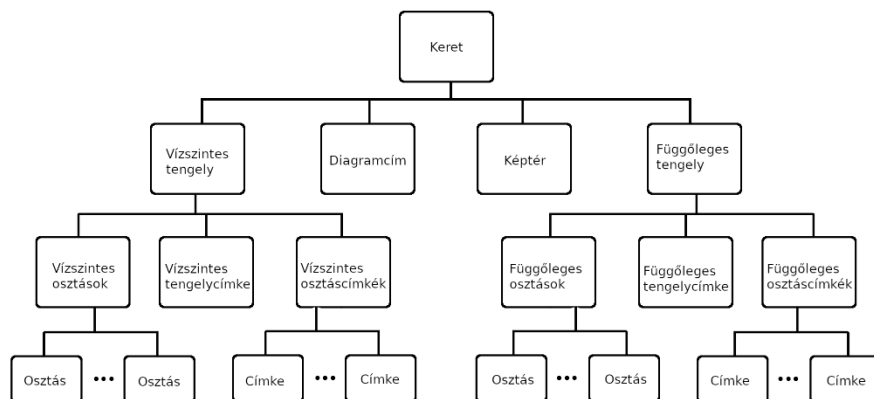
4. ábra Tengelykomponensek elhelyezéseinek paraméterei

4.2. Komponensgráf

Miután a grafikon kinézeti paraméterei módosíthatók, illetve a tartalmazó grafikus ablak átméretezhető, a hagyományos számítógépes grafikai eszközök nem bizonyulnak megfelelőnek a rendszer kifejlesztéséhez. Egyszerű grafikus alkalmazásokhoz elegendő az összes elem grafikus primitívekre bontása, alkalmasint képpontonként generálása, majd az így keletkezett kép megjelenítése. Ebben az esetben azonban folyamatosan frissíteni kell a képet, így lehetséges változtatások esetén a teljes grafikai elem újbóli generálása szükséges. Miután a grafikon jellemzése objektumokkal és azok kapcsolatával könnyen jellemezhető, a részkomponensek közt egy gráf szerű struktúra állítható fel. A gráfban reprezentált tartalmazási viszonyok megkönnyítik a grafika generálását amikor a grafikon teljes struktúrája megváltozik.

A JavaFX a grafikus objektumait a megjelenítés, pozicionálás és átméretezés végett egy komponensgráfba rendezi. Ez a komponensgráf egy olyan fa, amelyben a szülő-gyermek viszony azt adja meg, hogy egy adott objektum pozicionálását melyik másik objektum valósítja meg. Ezek alapján a gyökérelme a minden egyéb elemet tartalmazó ablak, amit az operációs rendszer megjelenít, egy levélelem olyan objektum, ami nem tartalmaz semmilyen másik objektumot, egy köztes elem pedig egy objektumcsoport, amely más objektumokat, objektumcsoportokat tartalmazhat. Ez alapján a grafikonon valóban megjelenő elemek lesznek ennek a gráfnak a levélelemei, a logikai csoportok pedig a szülőelemek. Ez a struktúra lehetővé teszi, hogy a megjelenített elemek közül egy művelettel egyszerre több elemet is eltávolítsanak, vagy beillesszenek, mivel a szülő eltávolításakor annak gyerekei is kikerülnek a JavaFX beépített komponensgrádjából.

A grafikon komponensgrádjának gyökérelme a keret (Frame osztály). Ez a specifikációnak megfelelően tartalmazza az összes többi elemcsoportot. Ennek köszönhetően a grafikon maga is komponenssé válhat és később bármilyen JavaFX csoportba vagy panelbe egy utasítással beilleszthető. A keret egyetlen levéleleme a diagramcímét megvalósító objektum. A keret tartalmaz továbbá tárolóelemeket is, ezek a függőleges tengely, a vízszintes tengely és a képtér. Mindkét tengely osztály tartalmaz egy címkét, vonal elemeket és további tárolókat: osztásokat és osztáscímkéket. Az osztások és osztáscímkék már csak levélelemeket tartalmaznak, amik a vonalak, illetve címkék az egyes tengely alapesetekhez. A függőleges és vízszintes tengelyek nem csak egy-egy tárolót és vonalat tartalmaznak. A diagram generálásakor az összes lehetséges beállítási alapeset elkészítésre kerül, ezeket a futás során a vezérlés ki, illetve be tudja kapcsolni. A képtér egy általános tároló, így bármilyen grafikus elem hozzáadható, ami lehetővé teszi különféle grafikontípusok vegyítését speciális vizualizációs feladatok esetén.



5. ábra A grafikus rendszer komponensgráfa

Az implementáció során a pozíciókat befolyásoló változók (specifikációs ábrákon kis latin, illetve görög betűkkel jelölve) mind egy-egy tulajdonságként kerültek tárolásra. A tulajdonság (property) a JavaFX egy beépített funkcionalitása, Egy tulajdonság értéke beállítható, lekérdezhető,

illetve összeköthető egy másik tulajdonsággal. Ebben az esetben az alaptulajdonság megváltozása esetén minden rákötött tulajdonság objektum értéke automatikusan megváltozik.

A pozíciók kiszámításához szükséges változóknak elérhetőnek kell lenniük a komponensgráf minden szintjén, azonban az objektumcsoportok elkülönítése érdekében célszerűnek találtam, hogy a csoportok mind rendelkezzenek ezekkel a változókkal, így lehetővé téve az önálló komponensként való működést. Ennek megvalósítása érdekében egy absztrakt osztály került létrehozásra, amely a JavaFX beépített Group - csoport osztályából származik, tartalmazza az összes változót reprezentáló tulajdonságot és képes önmagát egy másik ilyen implementációval összekötni. A tulajdonságok mindig a szülő csoport értékeihez igazodnak és automatikusan követik a változásokat. Az osztály neve Container. A szerkezet ilyen módon rekurzív adatszerkezetté vált, a tulajdonságok pedig mindig a gyökérelemig terjeszthetők vissza.

Az implementációban a gyökérelemet a Frame osztály képezi, ez a Container absztrakt osztály implementációja. A Frame tárolja a diagramcímet és a két tengelyt: HorizontalAxis és VerticalAxis néven. A specifikáció a könnyebb megértés érdekében magyarul készült el, a megvalósítás során azonban minden elnevezés angol nyelvű megfelelőjét használom.

4.3. Keret és tengelyek

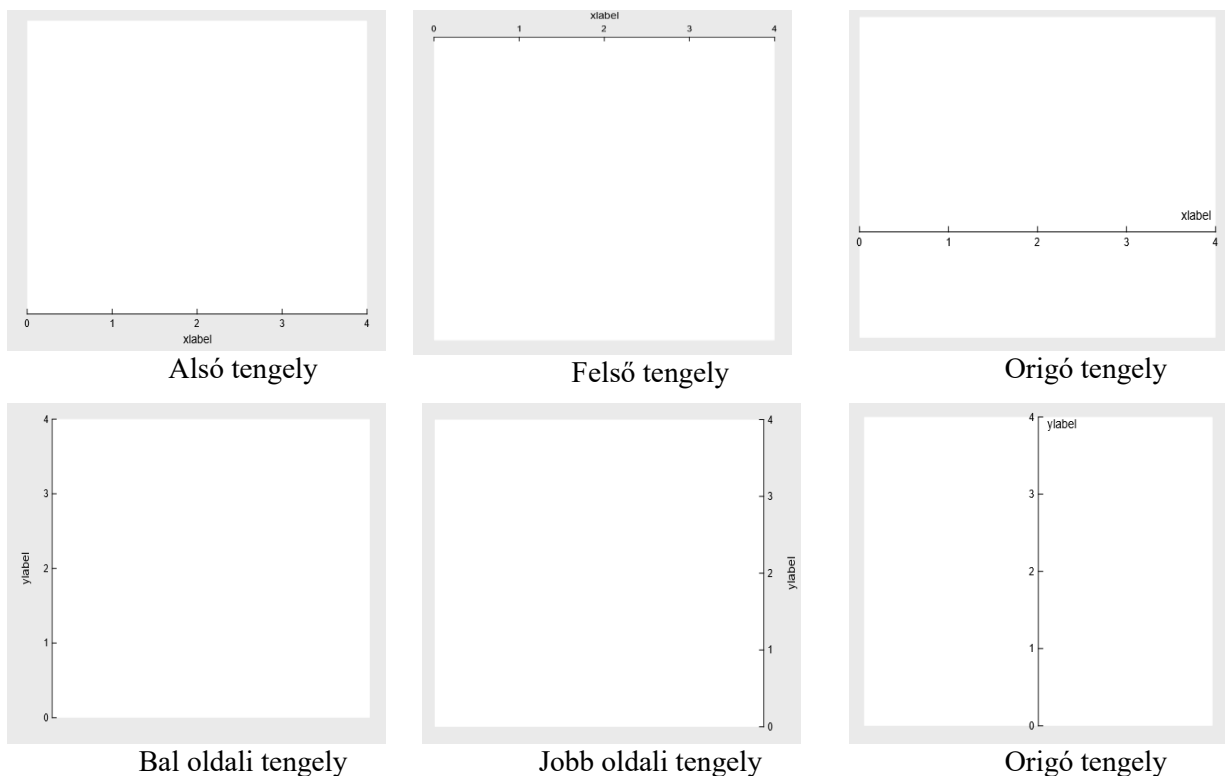
A vízszintes komponens alapértelmezett esetben a képtér alatt helyezkedik el, alatta megjelennek az osztáscímkék és a tengelycímke. A megvalósítás során létrehozásra került a HorizontalAxis, a HTickMarks és a HTickLabels osztályok. Mindhárom osztály a Container absztrakt osztály implementációja. a HorizontalAxis tárolja a tengely fővonalát, és egy-egy példányt a HTickMarks és HTickLabels osztályokból, valamint a tengelycímket Label példányként. Az osztáscímkék és osztások generálásakor csak az előre meghatározott helyre tudja az elemeket elhelyezni, és a csoport osztályokon keresztül kerülnek be a megjelenített objektumok közé. A vízszintes komponensnek a specifikáció szerint képesnek kell megjelennie a Képtér alján, tetején, illetve a képtér egy bizonyos pontján is az origó beállítás miatt. A megvalósításban ez a HorizontalAxis osztály kibővítésével jött létre, a különböző esetekben megjelenő extra elemeket egy-egy új változó bevezetésével kezeljük. A grafikon létrehozásakor ennek megfelelően kiszámításra kerül az összes lehetséges elrendezés, és a komponensek és csoportok egyesével ki-, illetve bekapcsolhatók. A tulajdonságok összekötése és a pozíciók dinamikus számítása miatt az éppen láthatatlan komponensek is minden méretváltozás esetén igazodnak az új állapotokhoz, illetve a ki és bekapcsolás a program futása közben is végrehajtható.

A függőleges komponens alapértelmezett esetben a képtér bal oldalán helyezkedik el, tőle balra megjelennek az osztáscímkék és a tengelycímke. A program struktúrájában a VerticalAxis, VTickLabels és VTickMarks osztályok reprezentálják ezt a működést. Az osztályok a Container absztrakt osztály implementációi, és a vízszintes tengelyhez hasonlóan a VerticalAxis tárolja a

vonalat, a `VTickLabels` és a `VTickMarks` egy-egy példányát és a tengelycímekét. A `VTickLabels` és `VTickMarks` osztályok csak az előre megadott pozíciókra tudják beállítani a megjelenített objektumokat. A függőleges komponens a vízszinteshez hasonlóan lett kialakítva, az éppen elrejtett elemek is folyamatosan követik az átméretezéseket. A specifikáció szerint a függőleges tengelynek képesnek kell megjelennie a képtér bal oldalán, jobb oldalán és a képtérben belül ofszetelve az origó beállítás alapján. A vízszintes tengelyhez hasonló módon.

Miután a tengely minden komponensét futás kezdetén generálja, a kliens kód számára lehetővé kell tenni az elemek megjelenítését és elrejtését. Ezzel a módszerrel elválasztható egymástól a megjelenítő kód és az azt irányító vezérlő programkód. A komponensek mind egy-egy változóként jelennek meg az objektumstruktúrában, így azok metódusokkal kontrollálhatóvá válnak. A gráf különféle szintjein a `Container` implementációktól lekérdezhetők a komponenseik, így azok vezérlése is elérhető.

A függvények párosával kerültek kialakításra, egy megjeleníti és egy elrejteti az adott komponenst. Megjelenítéskor nincs alapértelmezés, ha a vezérlő kód nem adja ki a megjelenítő függvényeket, akkor semmi sem jelenik meg az ablakban. A modularitás következménye továbbá az is, hogy ezt minden szinten végre kell hajtani. Ennek megfelelően nem elég bekapcsolni a vízszintes tengelyt, annak a megjeleníteni kívánt elemeit egyenként láthatóvá kell tenni. A különféle komponens beállítások megfelelő alkalmazásával elérhető bármilyen MATLAB grafikonból vett tengelyrendszer megjelenítése.



6. ábra A tengelyek lehetséges pozíciói

A tengely, címkék és cím külön vezérlés alá került, így ebben a kétszintű rendszerben a különféle diagramtípusok csak a diagram elemeinek megjelenítését kezelik, míg a közös és minden esetben egyező elemek vezérlését egy erre a célra létrehozott osztály kezeli, így elkerülhető a kód duplikálása és az abból fakadó hibák.

4.4. A különböző diagramok jellemző grafikus elemei

A vonaldiagram a MATLAB egy alapvető diagramtípusa, szinte minden vizualizációs feladat során használatos. A fejlesztés során egy olyan vonaldiagram létrehozása volt a cél, amely a JavaFX beépített átméretező rendszerét használja, így az alacsonyabb szintű integrált grafikus rendszer a többi alapértelmezett JavaFX komponenshez hasonlóan natív teljesítménnyel mozgatja és méretezi. Emellett cél, hogy a MATLAB formázási opciói alapján legyen lehetőség vonal vastagságát képpont mértékegységben megadni; a vonal stílusát folyamatos, szaggatott, pontvonal és pont-szaggatott stílusra állítani; a vonal színét megadni RGBA színkódolással, valamint a vonal színét és vastagságát követő markerek beszúrása az adatpontokhoz. A megvalósítás a JavaFX Path osztályát használja az adatpontok által kijelölt görbe megjelenítésére. Ez az osztály vonalszeletekkel működik, ezeknek a vonalszeleteknek a függőleges és vízszintes koordinátái normált rendszerben hozzáadhatók a képter méretéhez, ezzel megvalósítva az automatikus átméretezést. A markerek követik a hozzájuk tartozó vonal stílusát, a görbe komponenseihez hasonló módon pozícionálódnak az adatpontokhoz. A következő markerek kerültek megvalósításra: csillag, kör, pont, kereszt és pluszjel. Az eddig bemutatott vonalrendszer csak a grafikus műveleteket látja el, az adatpont értékeket $[0;1]$ intervallumra normalizálva várja el, így több lehetséges kontroller is használhatja különféle vonal típusú grafikonok megjelenítéséhez. Megjelenítésére egy metódus áll rendelkezésre a `PlotSpace` osztályban, ami a képteret valósítja meg. A metódusnak a vonal minden paraméterét meg kell adni, egy vonalat ad hozzá a képhez, azonban többszöri meghívásával tetszőleges számú vonal hozzáadható.

A fejlesztési folyamat során alkalmazásra került a Model-View-Controller (Modell-Nézet-Vezérlő) tervezési minta, a korábban bemutatott grafikus rendszer ebben az esetben a nézet réteget képezi. A modell és kontroller rétegekben valósulnak meg a különféle grafikontípusokhoz tartozó adatszerkezetek és a képteret vezérlő logika, így a végső felhasználói interakciót megvalósító EDSL és az alacsonyabb szintű grafika közt kialakult egy átmeneti réteg, amely jelentősen leegyszerűsíti az EDSL fejlesztését a későbbiekben. Vonaldiagram esetén a modell meglehetősen egyszerű, az adatpontok valódi értékét tárolja, azok minimum és maximum értékeivel. Ezekből az értékekből számítja a kontroller az összes adatsor végpontjait a modellek normalizálásához. A vezérlés több vonalat tartalmaz, feladata azok kezelése, valamint a hozzájuk tartozó jelmagyarázat adatok generálása. Miután a grafikonok nem csak vonaldiagramok lehetnek, a grafikus rendszer irányába egy `PlotSpaceController` nevű interfész osztály bevezetésével került biztosításra a módosíthatóság és a későbbi diagramtípusoknak megfelelő kontrollerek beépítése. Bármely osztály, ami ezt az interfészt implementálja, alkalmazható kontrollerként a grafikus rendszeren belül.

Az oszlopdiaagram szintén MATLAB alapgrafikon, a megvalósítás célja a multifunkcionalitás és az optimalizáció mellett a MATLAB funkcionalitás minél pontosabb megvalósítása. Ez a több adatsort tartalmazó diagramok esetén kifejezetten fontos, mivel azokat oszlopcsoportokkal vagy egymásra helyezett kisebb oszlopokkal valósítják meg. Ezentúl megjelent a vízszintes oszlopok és a negatív értéket reprezentáló oszlopok megjelenítése. Egy komplex oszlopdiaagramot a `PlotSpace` osztály `addBar()`, illetve `addHBar()` metódusaival lehet a képhez hozzáadni. A vonaldiagramtól eltérően több adatsor is megadható, az adatszolgáltató `im` nevű paraméterben, ami sorfolytonos mátrixként funkcionál, melyben az oszlopok az egymáshoz tartozó adatcsoportok. Ebben az esetben a mátrix sorai eltérő színekben jelennek meg, amiket a `color` paraméter ad meg, ennek feltétele, hogy a `color` tömb hosszúsága legyen egyenlő a mátrix sorainak számával. A `width` paraméter az egy oszlop számára rendelkezésre álló helyhez képest relatív értékben adja meg az oszlopok szélességét. Minden paraméter normál értékben értelmezendő a $[0;1]$ intervallumról, amennyiben az `im` paraméterben negatív érték szerepel, azt az alapvonali eltolás alatt fogja a rendszer megjeleníteni, de a hossza ennek az értéknek is a képtér magasságához képest relatív lesz meghatározva. Vízszintes oszlopdiaagramhoz az `addHBar()` metódus használható, paraméterezése megegyezik az `addBar()` metódusával. Minden beállítás hasonlóképpen működik, azonban az értelmezési tartományt a függőleges tengelyre, az adatsorokat pedig a vízszintes tengelyre vetíti.

A Scatter diagram egyedülálló adatpontok grafikus ábrázolására, és azok közti összefüggések vizualizációjára szolgál. A MATLAB programnyelven elérhető scatter diagram esetén beállítható az adatpontokat jelölő szimbólumok típusa, mérete és színe olyan módon, hogy a szín és méret adatpontonként megadható, így a pont helyén túl még egy tulajdonság ábrázolható. A megvalósítás a projektben már létező vonal markereket használja, ennek eléréséhez azonban szükségessé vált a jelölők módosítása olyan módon, hogy a típusuk mellett a méretük és színük is megadható legyen. A megvalósítás hasonló a vonaldiagramok marker rendszeréhez, az adatpontokat összekötő vonalak nélkül. Az adatpont értékeket az ábrázoló függvény $[0;1]$ intervallumra normalizálva várja. A méret és szín paramétereket is tömb formájában kell megadni, amennyiben a méret, vagy szín értékek tömbje egy elemet tartalmaz, azt alapértelmezetten az összes adatpontra érvényesíteni fogja a grafikus rendszer, így elkerülhető a fölösleges memória használat. A program emellett ellenőrzi, hogy ellenkező esetben a szín és méret tömbök megegyeznek az adatsorok hosszával, ezzel előre kiiktathatók a tömbcímezésre visszavezethető futási hibák.

5. A beágyazott nyelvet megvalósító interfész

Az interfész kialakítása során három hierarchikus szint lett kialakítva, melyek a teljes ablakra vonatkozó formázástól az egyes grafikonokban alkalmazott adatsorok személyre szabásáig kezelik az előbbieken vázolt grafikus rendszert. A továbbfejlesztés egyszerűsítése végett a Fluent Interfész nem végez közvetlenül semmiféle műveletet a grafikus alaprendszerrel.

Az ábrát a `Figure` osztály generálja, ennek a függvényeivel hozhatók létre a különböző grafikontípusok, melyeket azok specifikus stílus és adat objektumaikkal állíthatók be. Ezzel a módszerrel a Java nyelv típusrendszere biztosítja, hogy mindig megfelelő adatok és formázás kerül egy grafikonnak megadásra. Amennyiben a végfelhasználó példának okáért egy vonaldiagram adatait szeretné egy oszlopdiagramba megadni, a fordító azt azonnal jelzi és nem engedi lefutni.

A `Figure` a felsőbb absztrakciós szint, segédfüggvényeket biztosít önálló ablakban történő megjelenítésre. Ezt támogatva rendelkezik az ablakhoz köthető beállításokkal: cím és pozíció. A mintaként szolgáló MATLAB rendszerben azonos nevű eljárással indítható az egy ablakba elhelyezendő grafikonok definiálása. A `Figure` osztály továbbá rendelkezik egymás után láncolható függvényekkel, melyek az egyes grafikontípusokat létrehozzák azok saját függvényeivel.

Az ablakon belüli második szint a grafikontípusok esetén mindig megegyező tengelybeállítások szintje. Ebben a rétegben kerültek implementálásra a tengelyeket, címkéket és a jelmagyarázatot pozicionáló és méretező kód. Ezt a működést a `FrameStyle` osztály objektumai biztosítják, így az interfész minden grafikontípus esetén ugyanazt a paramétertípust használhatja a grafikonok létrehozó függvényeiben. Egy ábra több grafikont is tartalmazhat, azok tengelybeállításai eltérhetnek, ezért szükséges a két beállításcsoportot külön szintbe sorolni.

A harmadik, legbelsőbb szint a grafikonok adatsorainak jellemző beállításai. Ezt a szintet szükséges elkülöníteni a grafikonoktól, miután egy grafikon több adatsort is ábrázolhat. Ezen a szinten már minden grafikontípus adatsoraira más beállítások értelmezhetők, azonban bizonyos esetekben típusok felhasználhatják más típusok formázó- és adatszerkezeteit, például a logaritmikus diagram pontosan ugyanazt az adatszerkezetet használja, mint a vonaldiagram.

5.1. Több diagram egy ábrában

A MATLAB környezetben a `subplot()` függvény meghívásával tetszőleges mennyiségű és elhelyezkedésű grafikon beilleszthető egy grafikus ablakba. Ez úgy oldható meg, hogy a függvénybe minden grafikon esetében meg kell adni egy rács méretét, majd azoknak a celláknak a sorszámát, amelyeket a grafikon elfoglal. Ezek a sorszámok a bal felső cellával kezdődnek egyes sorszámmal és jobbra, majd lefelé haladva növekszenek. Az interfészbe beépítéskor ezt a kényelmetlenséget kiküszöbölve a függvényfüzér egy külön eleme adja meg egyszer a rácsméretet, az aldiagramok pedig már csak a saját pozícióikat. Ezzel a megoldással könnyebben módosíthatók az elrendezések és a

felesleges adattárolás, paraméterek is mellőzhetők. Ezt leszámítva a rács és a cellaszámok teljes mértékben igazodnak a MATLAB módszeréhez.

A funkcionalitás a `Figure` osztályba került beépítésre. Alapja a függvénylánc elején meghívott `tiles()` metódus, melybe megadható a sorok és oszlopok száma. Beépített átmeneti tároló osztályok segítségével sikerült elérni, hogy a függvény paraméterezése egyértelmű legyen, kiváló példája egyszerű attribútumok fluent interfésszel való megadására. Az átmeneti objektum tartalmazza a sorok és oszlopok számát, egy statikus indítófüggvénye van a `rows()`, amit egy módosított önmagát visszatérítő `cols()` metódus követhet. Ezzel a `tiles(1,2)` helyett a sokkal könnyebben olvasható és megjegyezhető `tiles(rows(1).cols(2))` függvény szerepel a DSL-ben. Ezután a `subplot()` metódussal adhatók meg a lehetséges cellák közül azok, amelyeket az algrafikon elfoglaljon. A különböző grafikonok megadása a saját osztályaikhoz tartozó indítófüggvénnyel történik.

5.2. Általános formázás

Bármely grafikontípus közös pontja a tartalomjegyzék és a tengelyek, így azok beállítását végző DSL komponens egy külön osztályban került megvalósításra, így támogatva az egész rendszer modularitását, mivel a mögöttes grafikus rendszer implementációja lehetővé teszi, hogy a `FrameStyle` osztály objektuma bármely grafikontípus vezérlőjével kompatibilis legyen. A `FrameStyle` osztály rendelkezik a szükséges metódusokkal a tengelyek pozicionálására és skálázására, valamint a tartalomjegyzék elhelyezésére és formázására.

5.3. Vonaldiagram

A korábbi példákban látható módon MATLAB nyelven a `plot()` függvény biztosít lehetőséget vonaldiagramok megjelenítésére. Miután általánosan használt grafikontípus, a fejlesztés során ezt használtuk a prototípusokhoz és egyéb példákhoz, továbbá a vonaldiagram adja a mintát a további implementált típusokhoz. A DSL szempontjából egy elemi komponens, a `Plot` osztály. Ez az osztály rendelkezik a hasonló nevű `plot()` függvény egyes változataival, melyek a különböző létrehozási módokat valósítják meg: az egyik esetben megadhatók a tengelyek beállításai, a másik esetben az alapértelmezések szerint működik. Miután az egyes adatsorok stílusát külön a külön rétegbe szervezett `Line` és `LineStyle` osztályok menedzselik, a `Plot` felelőssége azok tárolása, majd a biztosított, illetve alapértelmezett `FrameStyle` osztállyal történő beágyazása a `Figure` egy példányába.

A `Line` és `LineStyle` kialakítására példák az interfész megtervezésénél láthatók, a DSL kialakítása érdekében szükséges a két osztály megléte, mivel a `LineStyle` egy saját fluent interfésszel rendelkező átmeneti osztály, célja a konfigurálási adatok tárolása, amiket a `Line` osztály olvas ki a megjelenítéskor. A `Line` osztály rendelkezik egy metódussal, amely a `LineStyle` DSL nélkül, a paraméterek Stringként történő megadását is támogatja. Ez szintén a MATLAB nyelvből átemelt elem, azonban csak az alsó szinteken adatok megadására alkalmazható. Ennek oka a Java nyelv által biztosított támogatás változó számú függvényparaméterekre, azokat csak a függvény sorrendben utolsó

paramétereire lehet alkalmazni, melyeket egy tömb formájában érünk el a függvényből. Ennek komoly hátránya, hogy csak egyfajta típusú paraméterlistára alkalmazható, illetve a típus eltüntetésével `Object` alapobjektumokra, amely esetén futási időben lenne szükséges interpretálni típusok alapján a paramétereket, ahogyan azt a dinamikusan típusos MATLAB is végzi.

Az interfész többszintű modularitásának következtében a különféle logaritmikusan skálázott vonaldiagramok teljes mértékben kompatibilisek a korábban ismertetett vonal és annak formázó rendszerével (`Line` és `LineStyle` osztályok). A mintául szolgáló MATLAB programnyelv három változatot biztosít logaritmusosan skálázott grafikonokhoz: `loglog` (mindkét tengely), `semilogx` (logaritmusos abszcissa, lineáris ordináta) és `semilogy` (lineáris abszcissa, logaritmusos ordináta). A DSL megvalósításában a `Log` osztály azonos nevű függvényei implementálják a működést, melyek paritási okokból paraméterezés szerint mind a `Plot` osztály `plot()` függvényével megegyező változatokkal és paraméterezéssel rendelkeznek. A logaritmus alapértelmezett esetben tízes alapú, azonban ennek a beállításnak a DSL-be ágyazása a `FrameStyle` osztályba került, miután a tengelyekhez köthető beállítás és a vonalstílusba való beépítése redundanciát okozott volna. MATLAB programnyelven a logaritmus alapja a grafikon készítő API segítségével nem végrehajtható, kizárólag a megjelenítés után a mögöttes adatszerkezet módosításával, ami semmilyen módon nem illeszthető be a fluent interfészbe.

5.4. Scatter diagram

A scatter diagramot a MATLAB a `scatter()` függvénnyel támogatja, ezzel adhatók meg az adatok és a formázás. A scatter vezérléséhez egy a vonaldiagram alapján kialakított Fluent Intefésze készült el. Fő komponensei a `Scatter`, `ScatterStyle` és `Series` osztályok, melyek a `FrameStyle` osztállyal kiegészülve a `Figure` osztályba ágyazódnak be. A `Series` osztály az értelmezési tartomány, értékkészlet, méretek, színek tömbök mellett tárolja a grafikonban használt marker típusát. A `ScatterStyle` osztály a Fluent interfészbe kapcsolódik be, indítófüggvénye a `marker()` és a szín, méretek valamint marker típus megadásához használható. A `Scatter` osztály tartalmazza a létrehozó függvényeket, valamint beágyazható a `Figure` osztályba.

5.5. Oszlopdiagram

Az oszlopdiagramot a MATLAB programnyelv a `bar()` és `hbar()` függvényekkel támogatja. Az adatok értelmezési tartomány nélkül, illetve mátrixként is megadhatók, azonban a Fluent interfészhez igazodás végett az adatsorok csak külön adhatók meg, és azok azonos indexen szereplő elemei alkotnak egy adatcsoportot. Ezt a működést a `Category` és `Bar` osztályok valósítják meg. A `Category` opcionálisan tartalmazhat értelmezési tartományt, valamint a teljes oszlopdiagramra vonatkozó formázást. Emellett ezzel az osztállyal adhatók hozzá újabb adatsorok, MATLAB-tól eltérő módon egyenként adhatók meg azok színei. A `Category` osztály beágyazott osztálya a `CategoryStyle` tárolja a diagram stílusát (`group,stack`), valamint az oszlopok relatív szélességét.

6. Eredmények

6.1. Az elkészült diagramok példákkal

A legegyszerűbb és legáltalánosabban használt diagram a vonaldiagram. Amennyiben semmilyen formázást nem biztosít a felhasználó, az alapértelmezett palettáról vett színsorral színezt, 1 pont vastag összefüggő vonal jelenik meg MATLAB, valamint az annak mintájára készített Java rendszerben. Az interfész ugyanazokat a függvényneveket használja, azonban a Java nyelvi korlátjai miatt abban az esetben a `line()` függvényt is meg kell hívni az adatmegadáshoz.

A vonaldiagram számos formázási lehetőséget biztosít, ezek használata esetén szembetűnő a különbség a két interfész között, azonban a függvénynevek konzisztensen ugyanazok. Míg a MATLAB esetén a sorrend teljesen kötetlen, a Java DSL ezt csak azonos logikai blokkba tartozó parancsok esetén teszi lehetővé, mivel azok egy beágyazott függvényláncba tartoznak. A használat megkönnyítése érdekében továbbá a vonalak formázása is egy füzérbe lett szervezve. Ennek célja, hogy a MATLAB módszerét feljavítsuk olyan módon, hogy az első ránézésre egyértelműen jelölje, hogy pontosan milyen paramétert is állít be a fejlesztő.

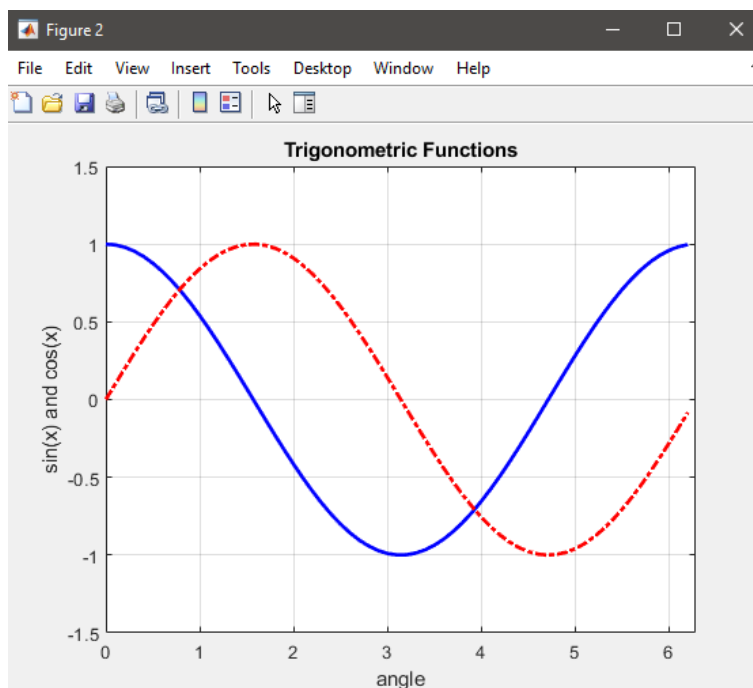
Az alábbi példán egy vonaldiagram MATLAB formázó kódja látható. A `figure` függvény előtt a szükséges adatpontok előállítás, utána a formázókód szerepel.

```
x = [0 : pi/100 : 2*pi];
y1 = cos(x);
y2 = sin(x);
figure
plot(x, y1, 'b', x, y2, 'r-.', 'LineWidth', 2)
grid on
xlim([0, 2*pi])
ylim([-1.5, 1.5])
title('Trigonometric Functions')
xlabel('angle')
ylabel('sin(x) and cos(x)')
```

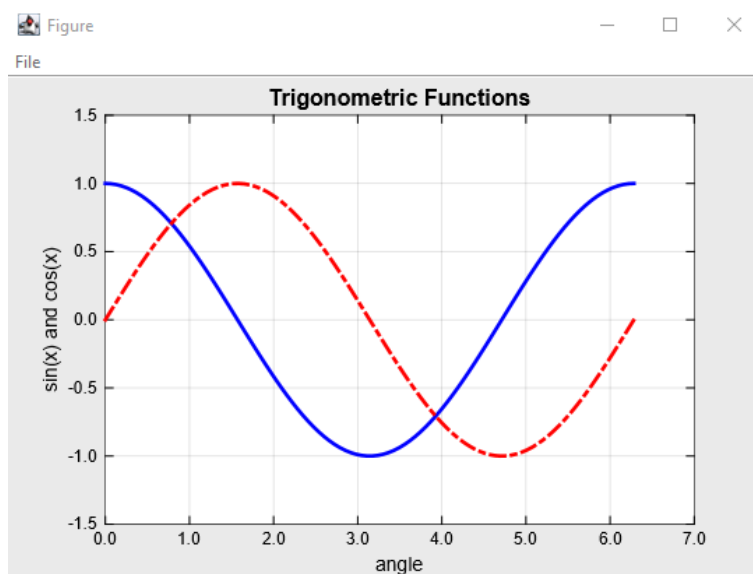
Az előző MATLAB példa az elkészült Java EDSL-re átültetett változata látható. Az előbbi kódtól eltérő módon a függvények meghívásának sorrendje valamelyest kötött, mivel a logikailag összetartozó beállítások egy függvényfüzérben szerepelnek. Azonban ez lehetővé teszi, hogy a metódusok paramétereinek sorrendje tetszőlegesen változtatható.

```
Figure fig = figure()
    .plot(title("Trigonometric Functions")
        .grid("on")
        .xlim( 0, 2*Math.PI)
        .ylim(-1.5, 1.5)
        .xlabel("angle")
        .ylabel("sin(x) and cos(x)")
        , line(x,y1,style("-").color("b").width(2))
        , line(x,y2,style("-.").width(2).color("r")));
fig.show();
```

A két rendszer által létrehozott grafikon ablakok a teljes mértékben személyre szabható grafikus rendszer következtében nagyban hasonlítanak egymásra.



8. ábra A MATLAB API által vormázott vonaldiagram



7. ábra A Java API által formázott vonaldiagram

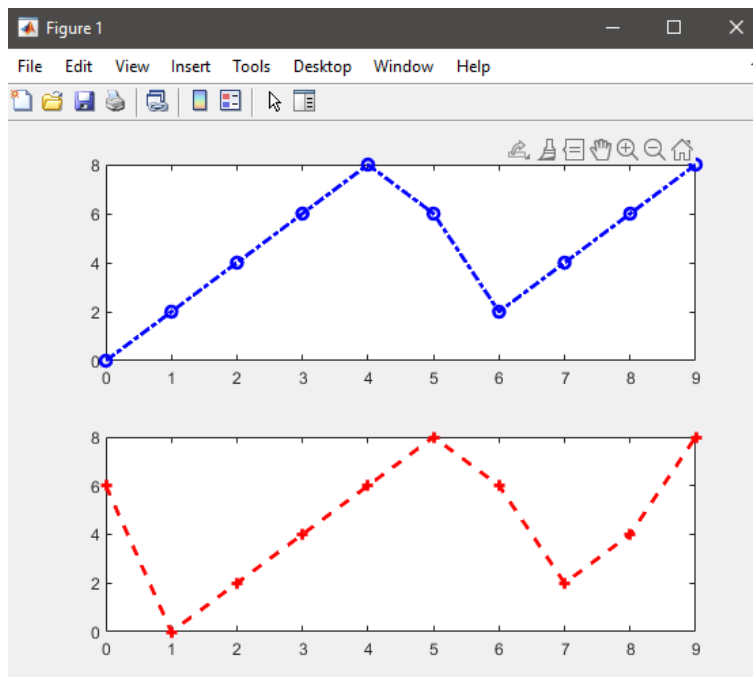
Több grafikon egy ablakban történő elhelyezését a MATLAB a `subplot` parancs segítségével támogatja. Miután a MATLAB egy script környezet, a kiadott parancsok egymásutánisága befolyásolja a működést. A Java programnyelven felépített DSL esetében ez nem lehetséges, így az interfész némileg eltér a mintától.

```
figure
subplot(2,1,1)
plot(x,y1, 'LineStyle','-.', 'Color', 'blue', 'LineWidth', 2, 'Marker', 'o')

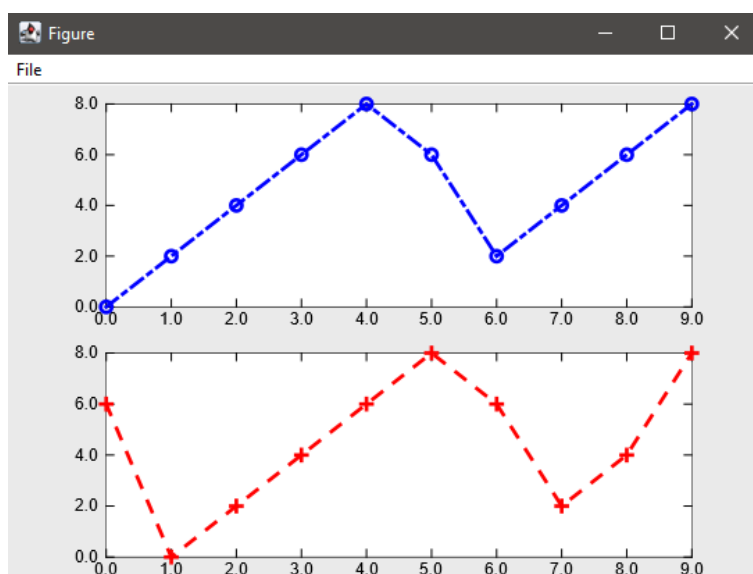
subplot(2,1,2)
plot(x,y2, 'LineStyle','--', 'Color', 'red', 'LineWidth', 2, 'Marker', '+')
```

A sorok és oszlopok számának folyamatos kiírásának elkerülése érdekében került beépítve a `tiles()` metódus, amelyben egyszer kell csak megadni a két paramétert, utána a `subplot()` függvény már csak a pozíciókat kéri az alábbi módon:

```
Figure fig = figure()
    .tiles(rows(2).cols(1))
    .subplot(pos(1),
        plot(line(x,y1,style("-.").color("blue").width(2).marker("o"))))
    .subplot(pos(2),
        plot(line(x,y2,style("--").color("red").width(2).marker("+"))));
fig.show();
```

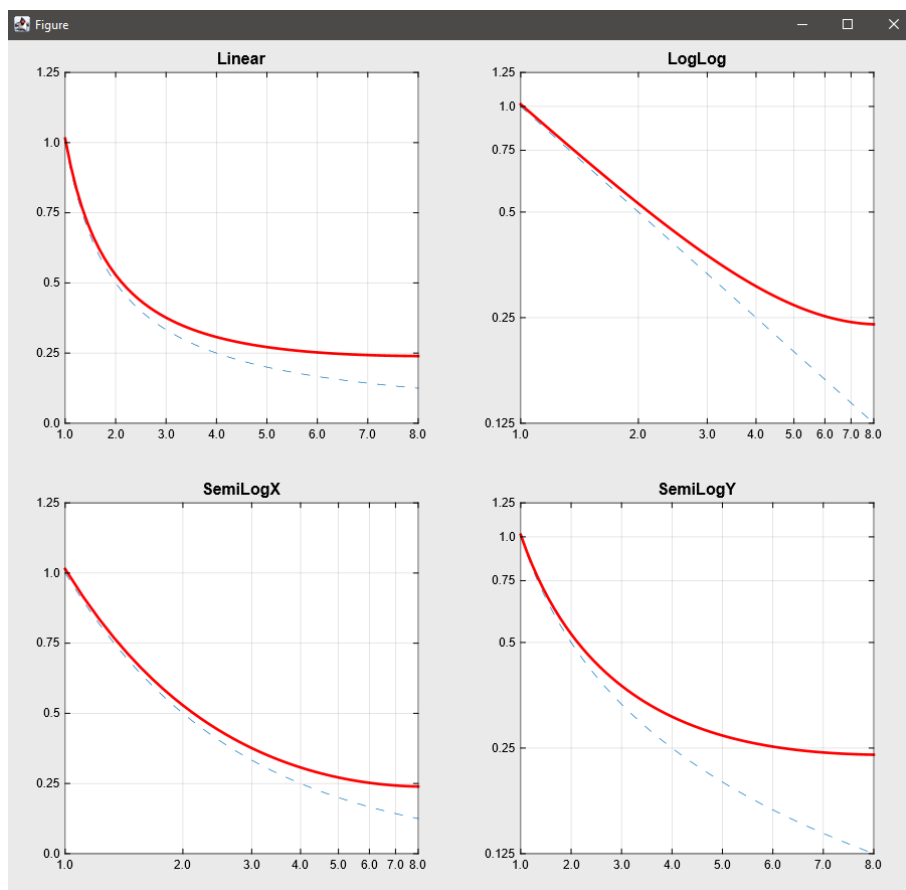


9. ábra A multiplot példa MATLAB változata



10. ábra A multiplot példa Java változata

MATLAB környezetben a logaritmikus formázású diagramokat az interfész sajátosságai miatt kezelhetjük különböző, önálló típusként, azonban mivel formázásuk megegyezik, a kifejlesztett interfészben jelentős mértékben nincsenek szétválasztva, a létrehozó függvény nevétől eltekintve pontosan úgy működnek, mint a fent bemutatott vonaldiagram.

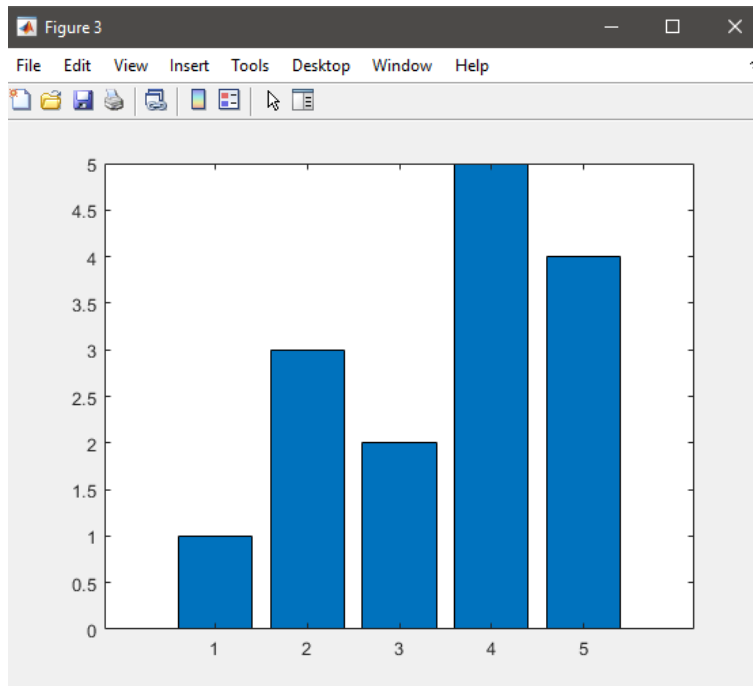


11. ábra Logaritmikus vonaldiagram változatai

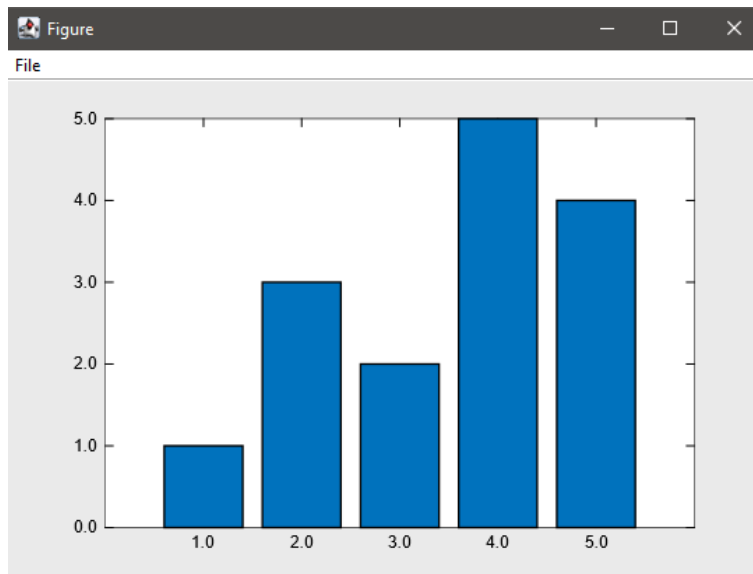
A fejlesztés elsődleges célja a vonaldiagram minél részletesebb és pontosabb támogatása volt, a korábban említett grafikontípusok is maradéktalanul elkészültek. Az alábbi példán egy egyszerű oszlopdiagram létrehozó kódjainak összehasonlítása szerepel, bal oldalon a MATLAB, jobb oldalon a Java kóddal. Az oszlopdiagram tartalmazhat több adatsort, azonban az értelmezési tartomány megadása gyakran nem szükséges. A Java rendszer a `series()` függvény segítségével teszi lehetővé egy adatsor megadását, melyeket aztán automatikusan mátrix szerű adatszerkezetté alakít, mivel a Java programnyelv nem rendelkezik nyelvi szintű mátrix támogatással.

```
figure
bar(x)
```

```
Figure fig = figure()
    .bar(category().series(x));
fig.show();
```

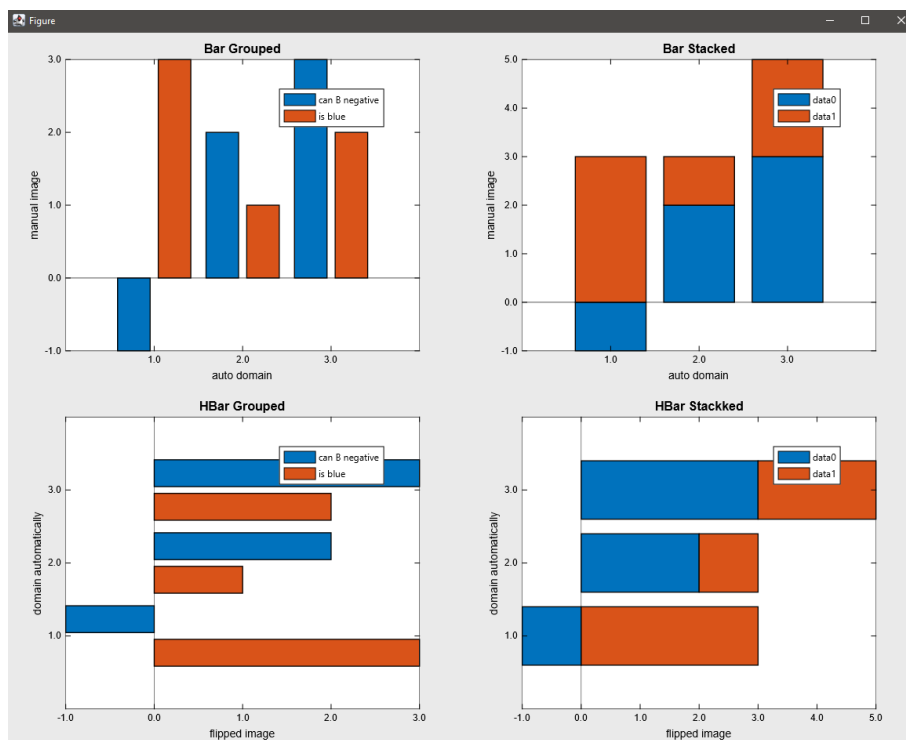


13. ábra Az oszlopdiagram példa MATLAB változata



12. ábra Az oszlopdiagram példa Java változata

Az oszlopdiagram elérhető vízszintes formában, több adatsor ábrázolása pedig egymás mellett csoportban elhelyezett oszlopokkal, valamint egymásra helyezett oszlopokkal lehetséges. A negatív tartományt a rendszer automatikusan kezeli.

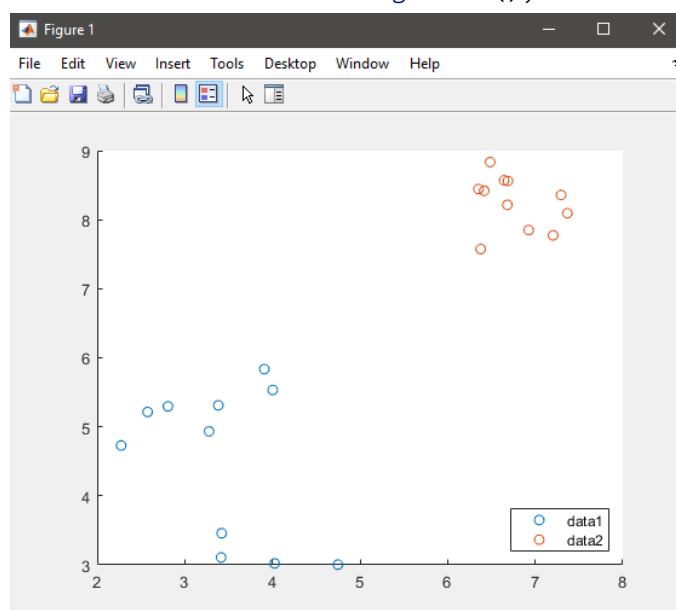


14. ábra Az oszlopdiagram támogatott típusai

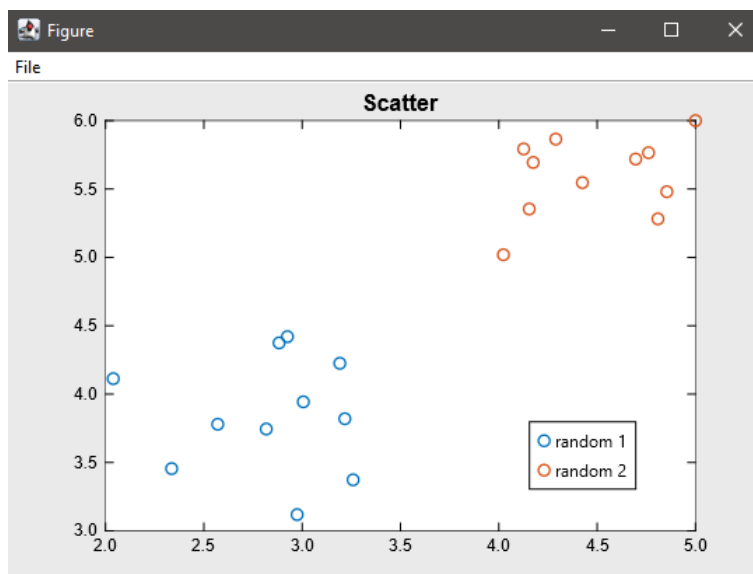
Az oszlopdiagram mellett megvalósításra került a statisztikából ismert scatter típusú diagram. Az alábbi példakódban a bal oldali módon definiálható MATLAB rendszerben a scatter diagram, amely több adatsor megjelenítéséhez mátrix formájában várja az adatokat. Ennek megkönnyítése érdekében a jobb oldali Java példában a `series()` függvény segítségével adható meg egy független adatsor, a formázás is ebben a függvényben valósítható meg.

```
figure
scatter([xs,xs2],[ys,ys2], "o")
legend(["random1","random2"],
       "Location","southeast")
```

```
Figure fig4 = figure()
    .scatter(title("Scatter")
    .legend(location("southeast"),
    "random 1", "random 2"),
    series(xs,ys,marker("o")),
    series(xs2,ys2,marker("o")));
fig4.show();
```



15. ábra A scatter példa MATLAB változata

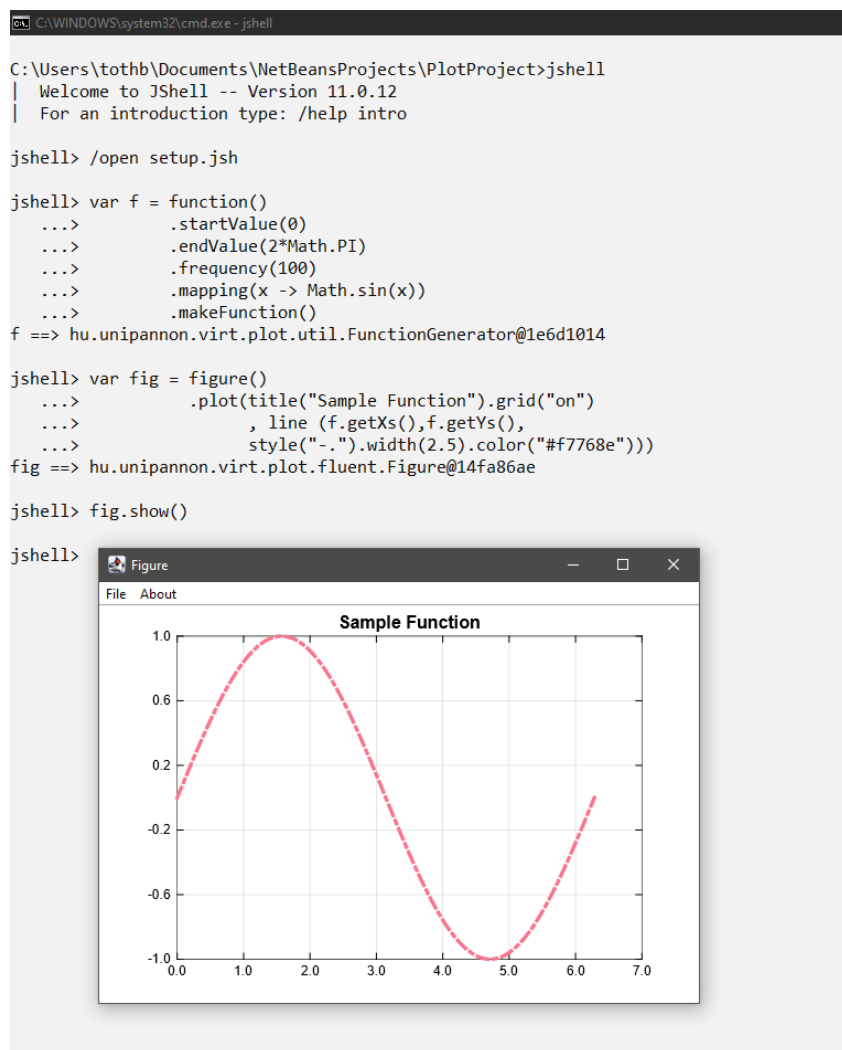


16. ábra A scatter példa Java változata

6.2. Shell támogatás

A Java programnyelv 9-es verziójától kezdve biztosít egy parancssori eszközt REPL (Read Evaluate Print Loop) alapú prototípusos fejlesztéshez, ez a JShell. A JShell programban kifejezésenként adhatók meg programsorok, amelyeket a rendszer Java bájtkóddá fordít és azonnal használhatóvá tesz. Miután a grafikonfejlesztő interfész egy függvényláncra épült fel, minimális beállítások megtétele után natív módon használható a JShell programban. Ezzel a módszerrel a munkafolyam szinte megegyezik a MATLAB parancssori folyamatával, leszámítva természetesen a szintaxisbéli eltéréseket. Miután a JShell lehetőséget biztosít könyvtárak, modulok betöltésére, valamint korábban megírt, fájlokban elmentett scriptek lefuttatására. Így minden feltétel adott, hogy a parancssori környezetben is kényelmesen használható legyen a modul, miután az azt tartalmazó archív állomány betöltése és a statikus indítófüggvények importálása is egy script segítségével előre megírt módon elemi műveletként elvégezhetőek.

A fejlesztés alatt elkészült teljes programcsomag egy Java 9 modul formájában öltött alakot, így a nyelv moduláris rendszerével könnyen hozzáadható bármely más alkalmazáshoz. A JShell használat érdekében készült egy inicializáló script, ami importálja az összes indítófüggvényt és hozzáadja a projekt archívumát a shell folyamat moduljaihoz.



17. ábra JShell környezet működés közben

7. Összegzés

A fejlesztési folyamat során külön lépésként került kivitelezésre egy multifunkcionális grafikus rendszer, a hozzá tartozó hagyományos objektum orientált programozói interfész, valamint a megszerzett ismeretek alapján egy deklaratív beágyazott doménspecifikus programnyelv a MATLAB nyelv mintájára. A Fluent Interfész használata opcionális, illetve lehetséges a két rendszer együttes használata is. Ezzel biztosításra került a minél szélesebb körben való felhasználhatóság, a modul jelenlegi állapotában bármely JavaFX alkalmazásban használható.

Sikeresen megvalósult a rendszer önálló programként történő üzemeltetése a JShell környezet segítségével, emellett biztosított a komponensként való beágyazhatóság is. Az interfész kialakítása segíti a későbbi felhasználókat és fejlesztőket MATLAB programból átvett példakódok egyszerűbb átültetésére. Az interfész bármilyen domén grafikonjainak megvalósítására alkalmas, emellett a meglevő rendszer struktúrájának szem előtt tartásával tetszőleges módon bővíthető.

Az elkészült modul alkalmas alapvető adatvizualizációs feladatok ellátására mellett további fejlesztési lehetőségeket is biztosít, az EEG elemzési folyamata során használt komplex, egyedi grafikonrendszerek kialakításának megvalósítása formájában. Ebben az esetben a DSL kiegészítése is indokolt az agykutatás és jelfeldolgozás doménjeinek jellemző szakkifejezéseivel és a nyelv formalizálásával. Az így kialakítandó újabb változat továbbra is kompatibilis maradna a jelenlegi verzióval. A későbbi fejlesztések előre gördítése céljából a projekt nyílt forráskódúvá tételének folyamata a jelen dolgozat megírásakor tart.

8. Irodalomjegyzék

- [1] M. Fowler, „DSL Guide,” [Online]. Available: <https://martinfowler.com/dsl.html>. [Hozzáférés dátuma: november 2022].
- [2] S. Klabnik , C. Nichols, The Rust Programming Language (Covers Rust 2018), No Starch Press, 2019.
- [3] „The Apache Groovy Programming Language - Domain Specific Languages,” [Online]. Available: <https://groovy-lang.org/dsls.html>. [Hozzáférés dátuma: november 2022].
- [4] P. Seibel, Practical Common Lisp, Apress, 2006, p. 277.
- [5] Z. Porkoláb, „Functional programming with C++ template metaprograms,” in *Central European Functional Programming School*, Springer Berlin Heidelberg, 2010, pp. 306-353.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Elements of Reusable Object-Oriented Software, Gamma, 1995.
- [7] Alex Ruiz, Jeff Bay, „An Approach to Internal Domain-Specific Languages in Java,” 2008. [Online]. Available: <https://www.infoq.com/articles/internal-dsls-java/>. [Hozzáférés dátuma: november 2022].
- [8] Steve Freeman, Nat Pryce, „Evolving an embedded domain-specific language in Java,” *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 855-865., 2006.
- [9] M. Fowler, Domain-specific languages, Pearson Education, 2010.