

# DIPLOMADOLGOZAT

Kedves Loránd

2016



Pannon Egyetem

Műszaki Informatikai Kar

Villamosmérnöki és Információs Rendszerek Tanszék

Mérnökinformatikus MSc

## DIPLOMADOLGOZAT

Dinamikus tudásreprezentáció

Kedves Loránd

Témavezető: Dr. Czúni László

2016

## Nyilatkozat

Alulírott Kedves Loránd hallgató, kijelentem, hogy a diplomadolgozatot a Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén készítettem Mérnökinformatikus MSc szak (MSc in Computer Science Engineering) megszerzése érdekében.

Kijelentem, hogy a diplomadolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy a diplomadolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja. A dolgozatban bemutatott DustCompact platform szellemi jogainak tulajdonosa az általam 2006-ban létrehozott Hajnalvilág Alapítvány, amelynek alapító okirata az oktatási és nonprofit felhasználást korlátozás nélkül lehetővé teszi.

Veszprém, 2016. november 30.

.....

Aláírás

Alulírott Dr. Czúni László témavezető kijelentem, hogy a diplomadolgozatot Kedves Loránd a Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén készített Mérnökinformatikus MSc szak (MSc in Computer Science Engineering) megszerzése érdekében.

Kijelentem, hogy a diplomadolgozat védeésre bocsátását engedélyezem.

Veszprém, 2016. november 30.

.....

Aláírás



# PANNON EGYETEM

## MŰSZAKI INFORMATIKAI KAR

### Mérnökinformatikus MSc szak

Veszprém, 2016. október 12.

#### DIPLOMATÉMA-KIÍRÁS

Kedves Loránd

Mérnökinformatikus MSc szakos hallgató részére

#### Dinamikus tudásreprezentáció

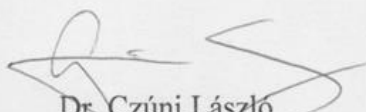
Témavezető: Dr. Czúni László

#### A feladat leírása:

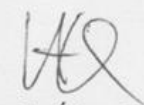
Az informatikai fejlesztések a futtató platformok, eszköztárak, tervező rendszerek, módszertanok gyorsan növekvő számának ellenére, vagy éppen emiatt folyamatosan idő és erőforrás hiánnyal, minőségi és fenntarthatósági problémákkal küzdenek. Segítséget jelenthet, ha a rendszerek elemzése, felépítése és fenntartása során keletkező, folyamatosan változó tudás nem kötődik a konkrét implementációhoz, önállóan tárolható, kezelhető. A hallgató feladata a problémakör elemzése, a megoldás szerkezetének és implementációjának ismertetése.

#### Feladatkiírás:

- Az informatikai szoftverfejlesztések hátterének rövid összefoglalása.
- A tudásreprezentációs módszerek irodalmi áttekintése.
- Az „informatikai rendszer, mint adatszerkezet” koncepció alapkomponeenseinek bemutatása példákon keresztül.
- A javasolt megoldás alapelveinek ismertetése.
- A jelenlegi és javasolt módszertan összehasonlítása általános feladatok elvi megoldása során.
- Konkrét komponens fejlesztési folyamat és példa alkalmazások bemutatása.
- A rendszert Java nyelven készítse el, mutassa be több platformon (parancssori alkalmazás illetve szerver oldali rendszer webes felhasználói felülettel).



Dr. Czúni László  
egyetemi docens  
témavezető



Dr. Bertók Ákos Botond  
egyetemi docens  
szakvezető



## Köszönetnyilvánítás

Köszönöm feleségemnek az állandó támogatást, amellyel egyetemi tanulmányaimat kísérte, fiaimnak, hogy a rájuk fordítható idő csökkenését elfogadták.

Köszönöm munkahelyeimnek az itt felhasznált sok értékes tapasztalatot, kollégáimnak és barátaimnak a kitartást, bizalmat és hitet, amely nélkül a Dust Platform nem jöhetett volna létre. Köszönöm mestereimnek, akikkel vagy személyesen, vagy műveik révén találkozhattam, akik példájából bátorságot meríthettem lehetetlen célok kitűzésére, és kitartást ahhoz, hogy a jelen állapotig elérjek.

Köszönöm a Pannon Egyetem tanárainak a magas színvonalú oktatást, amely megvilágította addig magányosnak érzett kutatásom tudományos hátterét, és különösen témavezetőmnek, Dr. Czúni Lászlónak azt, hogy segített felkutatni a terület irodalmát és kapcsolódó elemzéseit, állhatatos módon igazította „ipari” hozzáállásomat az akadémiai környezetnek is megfelelő formába.

---

*... lassan és remegve játszani kezdtem a melódiát,  
amit régen, régen, régen hallottam egyszer zengeni és  
zokogni a szívemben.*

*Karinthy Frigyes: Cirkusz*

---

## TARTALMI ÖSSZEFOGLALÓ

Informatikai rendszereket azért hozunk létre, hogy az infrastruktúra segítségével aktuális ismereteinket tároljuk, feldolgozzuk, közzé tegyük. Valójában a reprezentált tudás nem csak a kezelt adat, hanem annak szerkezete, a rajta végzendő művelet sor, sőt ezek terve, vagy a keletkező használati naplók is. A mai rendszereink alkalmasak az adatok változásának követésére, viszont a műveletek, struktúrák, célok biztonságos módosítása igen bonyolult feladat.

Ezzel analóg problémára adott megoldást Neumann János 1945-ben: definiált egy „tetszőleges algoritmus” gépi végrehajtására alkalmas komponens készletet, illetve megépítette a szolgáltatásokat végrehajtani képes fizikai eszközöket. Ennek révén a fizikai, mérnöki tevékenység elválhatott az algoritmus megalkotásától, a két szegmens gyors, önálló fejlődésnek indulhatott.

Ma a relék és huzalok helyét platformok, programozási nyelvek, eszköztárak vették át, de a rendszer szerkezetében reprezentált tudás ugyanúgy zárt és statikus. A megoldás egy homogén fogalomrendszer kialakítása az adatrepresentáció, rendszerépítés és futtatás alapvető komponenseire és szolgáltatásaira, amely ezt a magasabb szintű tudást képes ugyanúgy kezelni, átláthatóvá és dinamikussá tenni, mint az adatokat.

A dolgozatban a probléma kifejtése után elemzem az objektum reprezentáció szintjeit, egy jobb módszerrel szembeni alapvető elvárásokat. Bemutatom az általam létrehozott Dust Platform fogalmait, ismertetem a használatukat néhány fontos, de kellemetlen feladat megoldása során.

A platform elveire felépítettem a DustCompact Java fejlesztő és futtató környezetet. Ennek használatát a szokott „Hello, world” alkalmazás változatain, egy integrátor modul, illetve egy valós piaci alkalmazás példáján mutatom be. Végül értékelem a DustCompact-ot, mint fejlesztői keretrendszert, képet adok jelen állapotáról és a fejlesztési tervekről.

**Kulcsszavak:** tudásrepresentáció, szoftver architektúra, meta nyelv, modularitás, platformfüggetlenség



## ABSTRACT

We create information systems to store, process and publish our current knowledge using the infrastructure. In fact, the represented knowledge is not only the data we manage, but also its structure, the process algorithms, even their plans and the operation logs as well. Our current systems are optimized for dealing with the changes in the managed data, but changing the operations, structures or aims in a reliable manner is a very complex task.

John von Neumann solved a very similar problem in 1945. He defined a component set able to describe “any algorithm” for automatic execution, and built the physical machines that could do the required services. In this way, the engineering activity could separate from building the algorithms, the independent segments could start a quick development.

Instead of relays and wires, IT systems today are made of platforms, programming languages and tools, but the knowledge embedded in their structures are closed and static. The solution is a homogeneous terminology for data representation, system building and execution, which can handle this meta-knowledge the same, transparent and dynamic way as the managed data.

After elaborating the problem, I analyze the layers of object representation, and the fundamental requirements against a better methodology. I introduce the terms of the recommended solution, Dust Platform; demonstrate using them in some important, but unpleasant tasks.

I have also built DustCompact, a Java development and execution framework on this terminology. I demonstrate it by the common “Hello, World!” variants, an integrated module and a real-life application. Finally, I evaluate DustCompact as a development framework, detail the present state and future improvement plans.

**Keywords:** knowledge representation, software architecture, meta language, modularity, platform independence

## Tartalomjegyzék

1	A probléma feltárása .....	12
1.1	Alapfogalmak .....	12
1.1.1	A tudás fogalma és használata .....	12
1.1.2	A tudás szerkezete .....	13
1.1.3	A reprezentáció jelentősége és módszerei .....	14
1.2	Dinamikus tudásreprezentáció .....	15
1.3	Informatikai rendszer, mint tudásreprezentáció .....	17
1.4	Aktuális helyzetkép .....	19
2	Feladat – Az informatikai tudás feltárása .....	23
2.1	Az informatikai rendszerek rejtett szerkezete .....	23
2.1.1	Adatmodell.....	23
2.1.2	Alacsony szintű adatkezelés és műveletek .....	24
2.1.3	Üzleti szintű munkafolyamatok .....	26
2.1.4	Prezentáció, kommunikáció, integráció .....	27
2.2	Elvárások egy jobb megoldással szemben .....	28
2.2.1	Szoftver és adatszerkezet kapcsolata .....	29
2.2.2	Minimális kernel .....	29
2.2.3	Egyetlen adatelérő és kommunikációs csatorna .....	30
2.2.4	Egyszerű, hálózatos, dinamikus definíciók.....	31
2.2.5	Nincs új objektum.....	31
2.2.6	Nincsenek közvetlen gyűjtemény műveletek .....	32
2.2.7	Late binding .....	33
2.2.8	Transzparens algoritmusok .....	33
3	Módszer – Önhordó, minimális platform .....	34
3.1	Az „objektum” fogalom összetevői.....	34
3.1.1	Modell .....	34
3.1.2	Entitás .....	35
3.1.3	Szolgáltatás .....	36
3.1.4	Az új megközelítés előnyei.....	38
3.2	Meta réteg.....	38
3.3	Alkalmazás építés.....	41
3.3.1	Unit .....	41
3.3.2	Modul.....	42
3.3.3	Application.....	43
3.4	Futó rendszer .....	44
3.4.1	Események, üzenetek .....	44
3.4.2	Algoritmusok és környezetük .....	45
3.4.3	Kernel interfész .....	46
4	Működés – A módszer értékelése .....	48
4.1	Általános, gyakori problémák megoldása .....	48
4.1.1	Adatkezelés .....	48
4.1.2	Adattárolás és továbbítás .....	49
4.1.3	Dátumok, események, történet.....	50
4.1.4	Szövegek kezelése (nyelv, formázás, megjelenítés) .....	51
4.1.5	Adatmegjelenítés (felhasználói felület) .....	52

4.1.6	Biztonság – adat és szolgáltatás védelem .....	54
4.2	Fejlesztés Dust Platform környezetben .....	56
4.2.1	Feladat: adattárolás adatbázisban .....	56
4.2.2	Környezet és projekt konfiguráció .....	57
4.2.3	Adatbázis konnektor komponens .....	58
4.2.4	A választott adatbázis szerkezet ismertetése .....	59
4.2.5	EAV tároló komponens .....	60
4.2.6	Mellékszál: bináris adattároló .....	61
5	Összegzés .....	62
5.1	Szubjektív áttekintés .....	62
5.2	Jellemzés objektív szempontok szerint .....	63
5.3	Jelen állapot .....	65
5.3.1	A dolgozat háttérét adó munka .....	65
5.3.2	A DustCompact implementáció jellemzői .....	67
5.3.3	Minta alkalmazás: Hello world! .....	68
5.3.4	Éles használat: ERPort, egy webes ETL rendszer .....	69
5.4	Fejlesztési tervek .....	70
5.4.1	Dust Platform .....	70
5.4.2	Párhuzamos végrehajtás kezelés .....	71
5.4.3	Konfiguratív kifejezések, algoritmusok .....	71
5.4.4	Felhasználói felület .....	72
5.4.5	A Java környezet elhagyása .....	73
6	Mellékletek .....	74
6.1	DustCompact projekt szerkezet .....	74
6.2	Hello, World! .....	79
6.2.1	Kizárólag konfiguráció .....	79
6.2.2	Saját forráskód .....	80
6.2.3	Integráció .....	81
6.2.4	Mozgatás web szolgáltatás alá .....	82
6.3	ERPort .....	84
6.3.1	Projekt .....	84
6.3.2	Képernyőképek .....	85
6.4	Adatbázis kezelés .....	88
6.4.1	EAV táblák .....	88
6.4.2	Unit konfiguráció .....	89
6.4.3	Modul konfiguráció .....	92
6.4.4	Teszt alkalmazás konfiguráció .....	93
6.4.5	Szerver konnektor forrás .....	96
6.4.6	EAVStore forrás .....	98
6.5	Ábrajegyzék .....	103
6.6	CD melléklet tartalma .....	103
6.7	Irodalomjegyzék .....	104

# 1 A PROBLÉMA FELTÁRÁSA

## 1.1 ALAPFOGALMAK

A tudásreprezentáció és az esetleg ehhez köthető dinamizmus izgalmas, sokféle módon értelmezhető téma, ahogy az a kapcsolódó termékekből és szabványokból, illetve a terület sokszínű kutatásából könnyen kideríthető. Az első feladat tehát annak tisztázása, hogy a dolgozat milyen jelentést kapcsol ezekhez a fogalmakhoz, honnan vezeti le a megoldandó feladatot.

### 1.1.1 A TUDÁS FOGALMA ÉS HASZNÁLATA

A tudás fogalma már a köznapi értelmezésben sem egyértelmű, értünk alatta:

1. „okosságot”, tehát egyfajta absztrakt, koherens, megfelelően alátámasztott és ellenőrzött *ismerethalmazt*;
2. „ügyességet”, amely akár fizikai gyakorlatból, akár az „okosság” *eredményes alkalmazásából* származik; illetve
3. „tapasztalatot”, amely arra utal, hogy akár az ismerethalmaz, akár az alkalmazás módja *fejleszthető* a korábbi tevékenységek eredményének *elemzése, értékelése révén*.

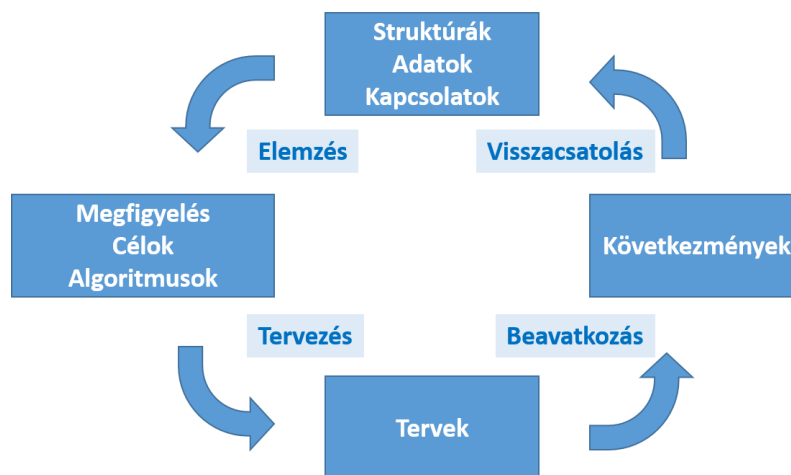
Ha a tudást „fekete doboz modellként”, egy nagyobb rendszerbe integrálva próbáljuk definiálni, kézenfekvő az összehasonlítás a reflexes, tudás használata nélküli tevékenységgel, amely a szükséges és rendelkezésre álló erőforrások felhasználásával mindig azonos eredményre vezet. Ha ehhez „tudást adunk” (például korábbi tevékenységet tapasztalatait, vagy paramétereket, amelyek alapján több módszer közül választhatunk), akkor ezek révén „hatékonyabb” tevékenység és „jobb” eredmény érhető el.

Ennek látványos bizonyítéka az evolúció: az adaptív reakciót lehetővé tevő idegrendszer fenntartása élettani szempontból igen „drága”, sok energiát követel – mégis, az egyre fejlettebb fajok egyre bonyolultabb idegrendszerrel rendelkeznek. Az emberi faj sem fizikai képességeinek köszönheti mai kiemelkedő szerepét, hanem különleges és rendkívül összetett idegrendszerének.

## A PROBLÉMA FELTÁRÁSA

### 1.1.2 A TUDÁS SZERKEZETE

A tudás később ismertetésre kerülő informatikai reprezentációjából nyertem vissza az alábbi hasznos, és fontos következtetésekhez vezető, szerkezetre összpontosító „fehér doboz modellt”. A folyamatok tekintetében csak a legegyszerűbb „generatív ciklust” ábrázoltam.



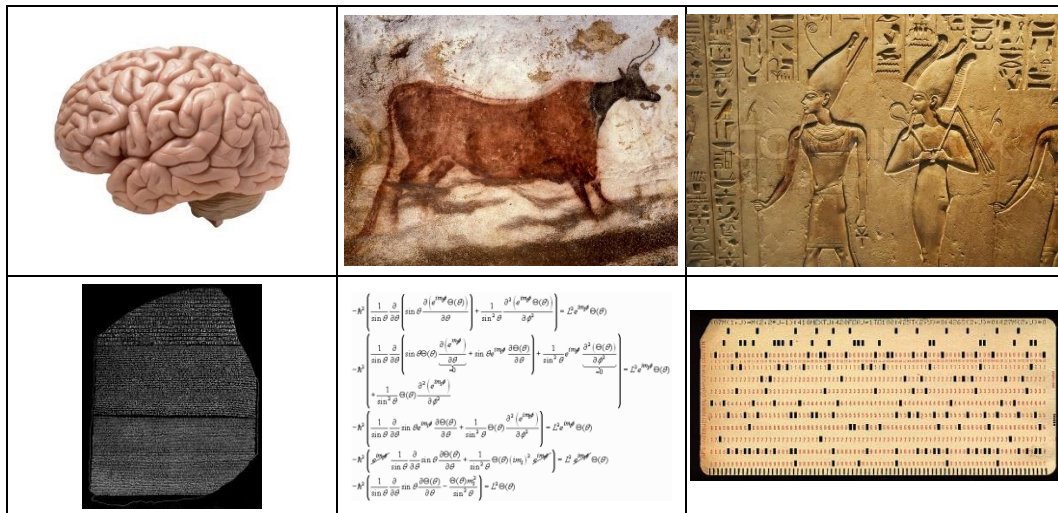
Ábra 1 A tudás praktikus, „fehér doboz” modellje

A felső téglalap a „helyesnek tekintett” **kiinduló ismereteket** tartalmazza, amelyeket megbízható, független forrásból szereztünk; ezek az alapfogalmak, azok szerkezete, kapcsolata, lehetséges műveletek. Ilyennek tekinthető például egy absztrakt algebra generáló elemeinek halmaza és műveleteinek definíciói. Ebből **elemzés**, következtetések révén egy **teljes rendszert** hozunk létre, az alap definíciók, mennyiségek és összefüggések segítségével képleteket alkotunk. Ennek megbízhatósága formális eszközökkel ellenőrizhető, becslhető. Köznapi értelemben itt keletkezik az ismeretből tudás, képesek leszünk megfigyelések végrehajtására, célok kijelölésére, majd egy konkrét cél megvalósításához vezető **terv elkészítésére**. Ennek birtokában interakcióba lépünk a tudásunkkal lefedett rendszerrel, **beavatkozunk** annak állapotába, működésébe. A keletkezett állapot felmérése a fogalmi rendszerünk által kezelhető újabb adathalmazt eredményez, a **következményeket** össze tudjuk hasonlítani a céljainkkal. Eddig csak az adatok változtak szabadon, kezelésük módszertana viszont csak tőlünk függött – azonban megtehetjük, hogy a következmények egyes elemeit **visszacsatolás** révén az ismereteink halmazába emeljük, így a tudásunk minden komponense dinamikus, adaptív rendszerré válik.

## 1.1.3 A REPREZENTÁCIÓ JELENTŐSÉGE ÉS MÓDSZEREI

A visszacsatolás történhet *szelekció* révén (az evolúció folyamatában így „tanulja meg” számtalan generáción keresztül egy faj az aktuális környezethez leginkább megfelelő tulajdonságokat), vagy *adaptáció* által, amikor a visszacsatolásból származó információt a konkrét egyed vagy csoport közvetlenül integrálja, és a saját versenyképességét ezzel növeli.

Az emberi tudás elsődleges reprezentációja az emberi agy neuronhálójának kapcsolatrendszere, amely természetesen csak birtokosa számára elérhető. A közösség kommunikációja révén azonban az egyed képes mások, máskor szerzett tapasztalatait is integrálni, ehhez valamilyen reprezentációs módszerre van szükség, amint az néhány kiragadott példán is látható.



Ábra 2 A tudásreprezentáció fejlődése

A fejlődés egyértelmű: a fogalmak közvetlen képi reprezentációja után keletkezett egyszerűsített jelrendszerek már komplex történetek, tevékenységek leírására is használhatók; a jelrendszer absztrakciója miatt megjelenik a fordítás fogalma; a matematikai jelrendszerek már közvetlenül nem megtapasztalható fogalmakat jelölnek; és ma már vannak nem emberi feldolgozásra szánt jelrendszereink is.

Megállapítható, hogy a reprezentációs módszertanok egyre *hordozhatóbbá* és *időtől függetlenebbé* tették a tudást; viszont egy konkrét jelsorozat *dekódolása maga is tudást igényel*, a fogalmak definícióinak és a jelek használati szabályainak ismeretét.

### 1.2 DINAMIKUS TUDÁSREPREZENTÁCIÓ

Az eddigieket összefoglalva elmondhatjuk, hogy:

1. Egy tetszőleges rendszerrel való interakcióhoz *aktuális* képet kell alkotnunk annak *szerkezetéről*: fogalmak, kapcsolatok, műveletek, majd ezek használatával az *állapotáról* is. Csak így lehetünk képesek *reális célokat megfogalmazni* és elérésük érdekében *hatékonyan beavatkozni* a rendszer működésébe.
2. Tudásunkat valamilyen *reprezentációs módszertan* segítségével tudjuk tárolni, átadni; a mai emberi tudásnak csak elenyésző része közvetlen tapasztalat, a legnagyobb rész ilyen úton jut el hozzánk.
3. A rendszerek szerkezetére és állapotára vonatkozó *tudásunk állandóan változik*, visszacsatolás révén minden szegmense frissítendő. Mivel az elavult ismeretek csökkentik a tevékenység hatékonyságát, *az új ismeretek integrációjának sebessége kulcsfontosságú*.
4. Gyakori, hogy tudásunkat több jelrendszer segítségével is tárolhatjuk, viszont ezek nincsenek teljes fedésben. A jelrendszerek közötti *fordítás időigényes, torzíthat, véletlenszerű hibák forrása lehet*.

Az emberi nyelv a tudás minden rétegét azonos nyelvi eszközkészlettel írja le, az ismeretkörökhöz szavakat alkotunk, ezekből mondatokat fogalmazunk, tudásunkat állítások, kérdések formájában rögzítjük, adjuk közre.

Ugyanilyen megszokott, de már nem annyira egyértelmű, hogy a különféle tudományágak mindegyike algebrai jelölésrendszert használ a típusok és tulajdonságaik, műveletek, folyamatok (szerkezet), illetve mennyiségek és viszonyok (állapot) leírására. A homogén reprezentáció révén nincs szükség „fordításra”: két kémiai anyag összekeverése, reakciója ugyanazon a „nyelven” értelmezhető, mint két test ütközése; a kísérlet eredményének reprezentációja, az ismeretek bővítése ugyanezzel a jelrendszerrel történik.

Az ember már nagyon régóta készít fizikai eszközöket különféle rendszerek vezérlésére, illetve információ továbbítására, amelyekben a „jel” szerepét egyre inkább valamilyen elektromos jellemző vagy annak változása tölti be, mert ennek előállítása és továbbítása általában igen egyszerű, könnyen tervezhető.

## A PROBLÉMA FELTÁRÁSA

Az irányítani kívánt rendszerek azonban egyre bonyolultabbak lettek, például egy telefonközpont kapcsolási rendszerének megtervezése, a terv megvalósítása, illetve hibák keresése már nem volt tisztán villamos mérnöki feladat. Ezt helyezte új megvilágításba Claude Shannon tanulmánya [14], amelyben a kapcsoló áramkörök elemzésére a matematikai logika nyelvét használta. Ezzel egyidőben a matematika (főleg a titkosítással kapcsolatos igények miatt) az algoritmusok, illetve az őket leíró nyelvtanok kutatásában tett hatalmas lépéseket.

Fontos tudatosítani tehát, hogy az 1940-es évekre

1. Egyre összetettebb elektronikus rendszerek valósítottak meg egy-egy irányítási algoritmust tekercsek, relék és huzalok stb. formájában.
2. Egy „fordítási módszertan” átjárást teremtett a kapcsolási rajzok és a matematikai jelrendszerei, elemzési módszerei között.
3. Az algebra eddig csak az elemek és műveletek definíciójáig terjedt, a számítások végrehajtását az emberre bízta. A Turing gép és a hozzá hasonló elvi konstrukciók viszont a végrehajtás folyamatát is matematikailag leírhatóvá, elemezhetővé tették.

A korszakalkotó áttörést a három kutatási irányt összegző Neumann architektúra [18] hozta el, amely egyrészt tartalmaz egy tetszőleges (természetesen matematikai pontossággal definiált) algoritmus végrehajtására alkalmas komponens készletet és műveleteiket (be-kimenet, memória, aritmetikai és logikai egység, végrehajtás vezérlő), illetve a definiált absztrakt komponensek fizikai megvalósítását.

Ezzel megszűnt az algoritmus leírása, és a végrehajtására képes fizikai eszköz megépítése közötti lassú, megbízhatatlan „fordítás”: a Neumann architektúra által biztosított szolgáltatások segítségével leírt algoritmust az általános végrehajtó eszköz azonnal képes volt futtatni. Az eszköz számára ugyanaz a „nyelv”: bitek sorozata volt az adat és a végrehajtandó utasítássor – csak az utóbbi dekódolása fizikailag megvalósult benne. A szoftver mérnök számára ugyanez a felület homogén tudásreprezentációt tett lehetővé: egyetlen jelrendszerben tudta rögzíteni a megvalósítani kívánt rendszer szerkezetével, műveleteivel és állapotával kapcsolatos ismereteit.



### 1.3 INFORMATIKAI RENDSZER, MINT TUDÁSREPREZENTÁCIÓ

A Neumann architektúra megjelenését az tette szükségserűvé, hogy az irányítani kívánt rendszerek túl nagyra nőttek, túl gyorsan változtak; a fejlesztés és adaptáció költsége meredeken nőtt, a minőségbiztosítás nehézkessé vált. A homogén, egyértelmű, a megvalósítás kötöttségeitől elszakadó algoritmus leírás előnyei jócskán felülmúlták az architektúra technológiai szempontból jóval összetettebb és lassabb komponensei által okozott hátrányt.

A független fejlesztés lehetősége a hardver előtt is megnyitotta az utat, hiszen innentől lényegében ugyanazt a pontosan definiált szolgáltatáskészletet lehetett újra, egyre magasabb színvonalon megvalósítani – *nem kellett minden vezérlési feladathoz új berendezést építeni*. Például az Apollo Guidance Computer (AGC) vezérlő programja a Neumann architektúra nélkül, tisztán hardver elemekből felépítve elképzelhetetlen lett volna – nem is első sorban a méret, sokkal inkább a komplexitás, javítások, módosítások követése, tesztelés nehézségei miatt.



Ábra 3 A csoportvezető Margaret Hamilton és az AGC forráskódja

## A PROBLÉMA FELTÁRÁSA

A modern informatikát egy hasonló ellentmondás feszíti. Ez a világ már nem közvetlenül fizikai eszközök vezérlését végzi, *elsődleges feladata a „nyers információ” tárolása, feldolgozása, továbbítása*. Az informatikai eszközök mérete és feldolgozó képessége felfoghatatlan fejlődésen ment keresztül (a köznap példája szerint ma egy átlagos személygépkocsi vezérlését nagyobb számítástechnikai kapacitás látja el, mint amekkorával a NASA rendelkezett az Apollo program idején). A globális informatikai környezet (a világ szerencsésebb felén) bárki számára elérhető; nem csak felhasználásra, hanem rendszerek építésére, programozásra. Számítalan játék bizonyítja, hogy az eszközeink ma már tényleg alkalmasak virtuális világok teremtésére: élethű térbeli, fizikai modellek között akár több tízezer is tevékenykedhetnek, léphetnek kapcsolatba egymással.



Ábra 4 Az eszközök és a virtuális világ fejlődése

Amikor viszont valós feladatokat szeretnénk megoldani, mintha láthatatlan korlátokba ütköznénk. A valós fejlesztések költségei és időigénye gyakran messze túllépi az eredeti terveket, rendszereink komplexitása a számtalan rendelkezésre álló programozási nyelv, eszköztár, platform, tervező eszköz, fejlesztési metodika ellenére drasztikusan nő. Rendkívül bonyolult, és folyamatosan változó, nehezen összehangolható komponens halmazból építünk még bonyolultabb rendszereket, láncuk csak annyira erős, mint leggyengébb szeme. A csúcstechnológiát képviselő Tesla gépkocsi felett az Android operációs rendszer sebezhetőségét kihasználva átvehető az irányítás; fertőzött wifi-képes eszközök ezreinek segítségével sikeres DDoS támadás indítható az internet működtetéséért felelős szerverek ellen.

### 1.4 AKTUÁLIS HELYZETKÉP

A probléma forrása az, hogy *merőben új célt próbálunk elérni egy korábbi feladat megoldására kialakított eszköztárral*. A Neumann architektúra a fizikai megvalósítást választotta el az algoritmus leírásának módszertanától, homogén tudásreprezentáció lehetőségét, ezáltal lehetővé tette a gyors tudás fejlesztést és adaptációt. A mai informatika célja viszont közvetlenül az, hogy a tudásreprezentáció eszközévé váljon. Ilyen szempontból vizsgálva *a programok létrehozására használt eszközök nem alkotnak homogén rendszert!* Az igényeket gyakran szöveges vagy rajzos formában kezeljük, azokból ismét hasonló terveket, majd forráskódokat, konfigurációkat készítünk; a teszt eredményeket hibakezelő rendszerben rögzítjük, amelyből megint dokumentációk keletkeznek. Az információ továbbítása a helyi szokásoknak megfelelően változatos kommunikációs csatornákon, rendezetlen formában zajlik, a jelrendszerek közötti fordítás természetesen itt is időigényes, nem egyértelmű, rendszer szintű (például a költségek tendenciózus alábecslése a piaci verseny miatt) és véletlenszerű hibáktól (prioritások, fogalmak félreértésétől a forráskódig) egyaránt terhelt.

Nem csoda, hogy amennyiben a teljes informatikai piacot szemléljük, egyáltalán nem a kooperatív, adaptív fejlődés, hanem a biológiai evolúció folyamatos szelekciós harcát láthatjuk. Azonos feladatokon cégek százai, programozók tízezrei dolgoznak, lényegében analóg, de egymással nem kompatibilis kódok és rendszerek tömegét előállítva. Ezek közül a könnyen befolyásolható vásárlói igények, a tömeghatás, a „cargo cult” jelensége stb. útján kerülnek ki a győztesek, akik felvásárlás, illetve programozók átcsábítása révén szívják fel a gyengébbeknél keletkezett tudást. Amíg pedig a szabványok, szabadalmak birtoklása révén szerezhető piaci előny határozza meg a nagy költségvetésű ipari kutatás irányát, egy valóban globális, szabadon hozzáférhető, szolgáltató-semleges, homogén eszköztár kialakítása rendkívül nehéz.

„Nagy elődök” közül elsőként a Charles Simonyi nevéhez fűződő Intentional Programming (IP) kutatást említeném, amelyet a Microsoft Research keretében kezdett az 1990-es években [15]. Az IP szemlélete szerint az algoritmus maga is adatszerkezet, amely más adatokra (változók, függvények) hivatkozik.

## A PROBLÉMA FELTÁRÁSA

Ezzel a megközelítéssel 2000 előtt tett lehetővé számos, ma megszokott szolgáltatást (kód bejárás a hivatkozásokról definíciókra ugrással, refactor, átjárás különféle formázási konvenciók, sőt, az algoritmus szöveges és grafikus reprezentációja között). A kutatás ma is tart, ígéretes célokkal [1], de a mai napig nem publikus eredményekkel. A megközelítés hibáját abban látom, hogy középpontjában a művelet („intention”) áll, amely a tudás egy szegmense csupán.

Tudást reprezentál minden informatikai rendszer. A szövegszerkesztők például karaktersorozatok formázási és szerkezeti beállításait, illetve elrendezését tárolja, ám minden szövegszerkesztő máshogyan. Amennyiben lehetőség nyílna a leírási módszertan szabványosítására, ez átvihetővé tenné a rögzített tudást (szöveget) bármilyen szerkesztő vagy megjelenítő platform között. Egy ilyen módszertan kialakításában nem feltétlenül érdekelt egy olyan cég, amely szövegszerkesztőt gyárt – számára csak az fontos, hogy más formátumban létrehozott szövegeket importálni legyen képes. Az eredmény az, hogy a digitálisan tárolt adatok az őket kezelő zárt rendszerekkel együtt gyorsan elavulnak, és míg egy kőtábla vagy könyv akár ezer év után is használható marad, egy húszéves digitális állomány tartalma már nem biztos, hogy könnyen elérhető.

A 2000-es évek táján az adatbázisok homogenizálása érdekében komoly erőfeszítéseket tettek egy általános leíró meta-nyelv létrehozására. Az egyik ilyen csoport a Metadata Coalition volt – ma már a domain nevük is szabad. Aktuális, szerényebb kísérlet az XBRL, üzleti jelentések adatainak általános, szabványos kezelésére, Open Information Model néven, de már az 1.0 változat is küzd a nem egyértelmű definíciókkal, és az eltérő reprezentációs formátumokkal (XML-re épül, de az adatok természetesen JSON, CSV stb. formátumban ugyanúgy tárolhatók lennének). Erős piaci csoportot egyesít az 1992-ben alapított Distributed Management Task Force (DMTF), és az általuk kezelt Common Information Model (CIM). Ennek aktuális, 2.43-as változata egy 40MB-os XML formájában tartalmazza számtalan közösen használt objektum leírását, részletes UML diagramok mutatják be a használatukat stb. Sajnos a piaci viszonyok itt elsődlegesek, konfliktus esetén gyorsan jelenik meg egy konkurens csoport azonos szabványa; egy ilyen méretű „mindent tartalmazó” XML haszna egy szokványos feladat megoldásában kérdéses.

## A PROBLÉMA FELTÁRÁSA

A szükséges támogató eszköztár kifejlesztése számos programozási nyelven óriási feladat lenne, ezek a csoportok inkább ajánlásokat adnak. Ennek hiánya viszont súlyos következménnyel jár azokra nézve, akik például a CIM 2.37-es változatára építették az infrastruktúrájukat...

Az informatikai tudásrepresentáció központi területe a rendszer tervezés, illetve a tervekől forráskódok adatbázis adminisztrációs parancsok stb. létrehozása, ennek zászlóshajója az 1997 óta az Object Management Group által fejlesztett Unified Modeling Language (UML) [16]. Ez alapvetően vizuális eszköztár, pontosan definiált diagramok segítségével lehet leírni a rendszerek szerkezetét és működését, így segíti az elvárások formalizálását, fejlesztési tervekké alakítását és bizonyos szintű ellenőrzését. Valós fejlesztési körülmények között a módszernek nem csak előnyei vannak [13]: a rendszer tervéből generálható forráskód váz még nem a teljes rendszer. A tervek és a valóság az utólagos módosítási kérések és a fejlesztés során jelentkező problémák helyi megoldásai miatt egyre nő. A probléma, hogy a tervet és a futó rendszert más jelrendszerben kódoljuk javarészt manuális fordítással, nem megkerülhető.

Az ipari gyakorlat a problémákra módszertanokkal igyekszik reflektálni: tervezési minták definíciója és ismertetése [3], az infrastruktúra komponensek atomizálása (például Microservices architektúra [2]) révén, a rendszeralkotási folyamat modellezésével (a közismert Agile, vagy az ígéretes Adaptable Design [8]), ellenőrzésének szabványosításával: CMMI/SPICE [19]. Az új módszertanok igyekeznek ugyan követni a tudásrepresentáció dinamizmusát, de ez megnehezíti a minőségbiztosítás és ellenőrzött fejlesztés feltételeinek teljesítését – a reprezentációs módszertanok közötti manuális „fordítás” pedig változatlan.

Az elmúlt években nagyszabású fejlesztés indult a Big Data, Artificial Intelligence, Internet of Things területek igényeinek hatására, számos vezető informatikai cég Platform as a Service (PaaS) megoldásokon dolgozik. Ezekre viszont a fejlesztők elvárásainak [9] kiszolgálása miatt egyértelműen „integráló” törekvés: elvi kérdések megoldása helyett a meglévő szolgáltatások és nyelvek lefedése, beolvasztása jellemző [5, 7, 10]. A homogén fogalomtár nem oldja fel az eltérő nyelvek miatti redundanciát, és függő helyzetbe hozza a fejlesztőt.

## A PROBLÉMA FELTÁRÁSA

A tudományos kutatás számára régóta egyértelmű, hogy a szoftverfejlesztés kulcskérdése a transzparens és követhető tudásreprezentáció, a „Knowledge Based Engineering” – azonban az elemzések általában csupán ennek problémáira tudnak rávilágítani, és további vizsgálatok fontosságát hangsúlyozzák [17], elemzésük szerint az „ipar oldaláról” egyelőre nem látható az okokra fókuszáló megoldás.

A tisztán tudományos kutatásban két alapvető irány érzékelhető. Az egyik tárgya az, hogyan feleltethetünk meg informatikai fogalmakat a természetes gondolkodási, nyelvi konstrukcióknak, és ebből próbálunk valamilyen automatizmust alkotni ezek formalizálására. Ezen az úton hamar beleütközünk az emberi gondolkodás nem egyértelmű természetébe, így több, lényegében ekvivalens nevezéktant alkothatunk, ugyanakkor az eszköz használata már betanulást igényel, így éppen a lényegét veszíti el [12].

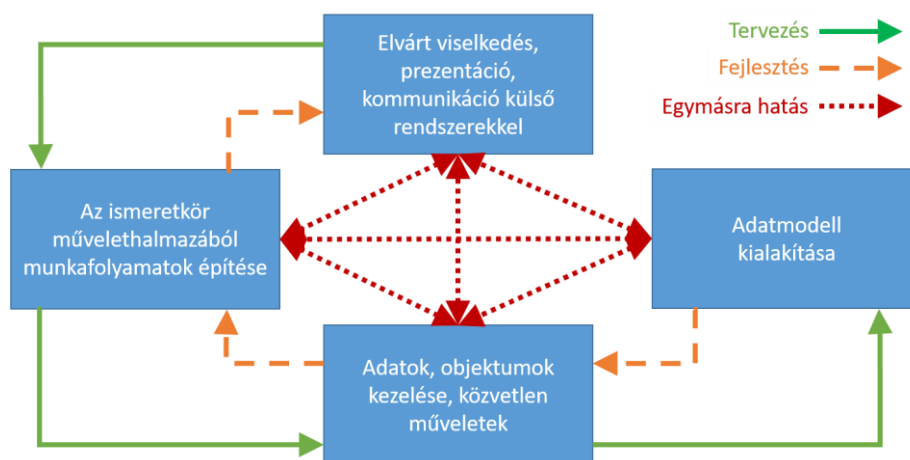
Technológia vonalon haladva kutathatjuk a folyamatok, algoritmusok megvalósítási környezettől, programozási nyelvtől független leírási módszertanát. Ennek a megközelítésnek szintén igen széles irodalma van, amelyek átfogó elemzése inkább további feladatokat jelöl ki, semmint praktikus konszenzusra utalna [4]. Bizonyos tekintetben a Neumann elvet követve, elszakadhatunk a konkrét környezettől, és vizsgálhatjuk az a rendszert, amelynek feladata a tudásunk rögzítése. Ez egyfajta absztrakt rendszer, a „rendszerek rendszerének” építése [11], vagy a másik irányból közelítve „meta-meta modellezés”: a modell építés során használt fogalmak modellezése [6]. Sajnos ezek a meta-rendszerek pontosan ugyanolyanok, mint bármely konkrét rendszer: az elemzés során ugyanazok a nehézségek jelentkeznek, ráadásul a célterület fogalmait még nehezebb definiálni.

A dolgozatban ismertetett Dust Platform elvi kérdések elemzése helyett azt a minimális fogalomtárat keresi, amely lehetővé teszi 1: bármely „dolog” informatikai leképezését (beleértve leképezéshez használt fogalmakat is), ezekkel 2: tetszőleges feladat leírását, és 3: egy cél platformon a végrehajtását. A létező megoldásokat nem a „hordozó” forrásnyelvek támogatásával integrálja, hanem absztrakt konnektor modulok létrehozását, megosztását és használatát teszi lehetővé. A platform használatát egy működő Java implementáción mutatom be.

## 2 FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

### 2.1 AZ INFORMATIKAI RENDSZEREK REJTETT SZERKEZETE

Ha félreteszünk minden megoldással kapcsolatos specifikus ismeretet, és kizárólag azokra a belső mintákra koncentrálunk, amelyek mérettől és környezettől függetlenül minden működő informatikai rendszernek alkotóelemei, akkor az alábbi ábrát kapjuk. Ennek komponenseit fogjuk vizsgálni elvi feladat, és a gyakorlati életben megjelenő formái alapján (önkényesen a „Fejlesztés” irányából sorba véve ezeket).



Ábra 5 A szoftver alapelemei, egymásra hatásuk a rendszer élete során

#### 2.1.1 ADATMODELL

Informatikai rendszert mindig valamilyen programozási eszköztár segítségével hozunk létre (programozási nyelv, adattároló rendszer, cloud, ...). Ennek „be kell mutatni” az adatszerkezetet valamilyen nyelvi szerkezettel: C struktúrák, Java osztályok, adatbázis táblák, LDAP osztály definíciók stb. Enélkül nem tudunk hozzájuk férni, viszont ezzel az adatszerkezettel kapcsolatos absztrakt, környezetfüggetlen tudásunkat „lefordítottuk” az adott környezetre. Egy iteratív fejlesztési folyamatban ez egyre bonyolultabbá válik, a fejlesztők egyre nehezebben tudják visszafejteni a struktúrák kapcsolatait és azok jelentőségét. Ráadásul a fejlesztés közbeni változások már nem mindig vihetők végig az elkészült részek függései miatt, nincs idő refaktorra, *a kód már nem képes reprezentálni a tudásunkat*, csak az állítható (jó esetben), hogy a tesztesetekben rögzített elvárásoknak megfelelően viselkedik.

## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

A terv->futó rendszer transzformáció elsődleges eszköze szövegállományok létrehozása és karbantartása, annak minden esetlegességével és hibaforrásával, ezért nagy rendszerekben céleszközöket használunk (UML szerkesztő, DSL eszközök), forráskódok generálására. A fejlesztés iteratív időszakában azonban ezek háttérbe szorulnak, és gyakran szerkezeti változtatások is „örökéletű gyorsjavítás” formájában, közvetlenül forráskódban keletkeznek.

Az eredmény mindkét megközelítés hibáit örökli: *van ugyan egy tervünk, amely a magasabb szervezeti szinteken a kommunikáció alapja, de sosem lehet tudni, pontosan mennyire tér el a rendszer valós állapotától.*

### 2.1.2 ALACSONY SZINTŰ ADATKEZELÉS ÉS MŰVELETEK

A rendszer által a saját nyelvén kezelhetővé tett adatokon ezután műveleteket kell végezni. Itt felmerül ugyanaz a probléma: a feladat konkrét gépi utasításokra fordítása esetleges. Tény, hogy a tervek (elvileg), illetve az elkészült kód gyakorlatilag is „fehér doboz modell”, hiszen egyébként a fordító nem lenne képes belőle futtatható kódot generálni. A kettő közötti transzformációt azonban emberek végzik, egyénileg értelmezve az átalakítás módját, és természetesen hibákat elkövetve közben. Ezek egy része szintaktikai és forráskód szintű ellenőrzéssel megtalálható – a veszélyesebb része viszont rejtett maradhat: bizonyos konvenciók megsértése, félreértések, kommunikációs problémák. Ennek eredménye lehet például beágyazott rendszerek esetén, nagyon speciális feltételek együttállásakor bekövetkező lassú feldolgozás, leállás, tervektől eltérő működés.

Az ilyen rejtett hibákat automatikus ellenőrző programok, tesztek halmaza igyekszik megtalálni a futó rendszerben (például autóipari környezetben tesztpályán rögzített eseménysorok visszajátszásával), azonban az ilyen tesztek számára a forráskód „fekete doboz”: a sikeres lefutás nem igazolja az elvárások és a forráskód közötti transzformáció helyességét.

A másik megközelítés a rendszer modulokra bontása, illetve létező modulok felhasználása, hogy a lokális feladat megoldásához szükséges transzformáció mennyisége csökkenjen. Ez külső (más cég által fejlesztett) modulok használata esetén annak feltételezése, hogy ami nekem házon belül nem sikerült, azt valaki más hibátlanul elvégezte.



## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

Tény, hogy így megoszlik a terhelés, viszont ez továbbra sem garancia a hibátlanságra, külső függést jelent: esetleges hibák lokális javítása nehézkes háttérismeretek nélkül. A nem megfelelő működés helyi megkerülése szintén bonyolult, későbbi javítócsomagok, új verziók átvétele lassú, hiszen a saját rendszerben ezek nem tervezett mellékhatásokkal járhatnak. Ráadásul az általános eszköztár jellemzően sokkal szélesebb, mint a valóban igényelt szolgáltatás, és gyakran további függéseket is behoz.

A belső modulfejlesztés elkerüli a fenti problémákat, azonban mivel csak „háttérmunka”, szűk erőforrásokkal dolgozik, kevés idő jut a tervezésre és tesztelésre, a későbbi módosítások általában elvezetnek a „miért nem használtunk külső modult?” kérdéshez.

Az eredmény fejlesztői oldalon általában hibrid: külső modulok és saját komponensek cégre jellemző arányú keveréke. Újakkal (most felfutó rendszerek és szolgáltatások) több a külső eszköz, régebbi vagy minőségi szempontból kritikus rendszereknél pedig jellemző a „már tíz éve működő” kódbázis konfiguráción keresztül való igazgatása (operációs rendszerek, beágyazott alkalmazások – home automation, kontrollerek stb.)

A nyilvánvaló piaci igény miatt számos informatikai cég specializálódik modulok, eszköztárak, környezetek létrehozására, más fejlesztők igényeinek kiszolgálására. Az ilyen szolgáltatók jól elkülöníthető rétegei:

1. C jellegű nyelvek - amelyek kizárólag az adatmodell és az algoritmusok leírására szolgálnak;
2. a nyelv részeként kezelt eszköztárak (C STL);
3. a nyelv kiterjesztése egy futtató platform lefedésére (Windows API);
4. a programozás általános eszközeinek nyelvbe, környezetbe integrálása (Java, .Net, J2EE – gyűjtemények, matematikai szolgáltatások, ...);
5. az általános eszközök nyelv feletti megjelenése absztrakt formában: design patterns (GoF, microservices, stb.), konkrét szolgáltatásként: „aaS” környezetek, Cloud szolgáltatók, ennek csúcsa a PaaS (Platform as a Service).

### 2.1.3 ÜZLETI SZINTŰ MUNKAFOLYAMATOK

Látható, hogy a fejlesztés alacsony (géphez, platformhoz kötődő) szintje sok problémát okoz, a feladat megismerése, elemzése során keletkezett absztrakt tudás „elsüllyed” benne. Ugyanakkor természetesen egyértelmű, hogy a matematikai algoritmusok, üzleti adatkezelés, munkafolyamat vezérlés, ... fogalmai és szolgáltatásai az aktuális futtató környezettől függetlenül, egyformán kell működjenek. Ennek megfelelően megjelennek azok az eszközök, amelyek egy célterület, „business domain” fogalmait definiálják, és az alacsony szintű adatkezelés feladatát a háttérben valósítják meg egy (nagyobb rendszerek esetén pedig számos) ismert platformon. A konkrét feladat megoldására pedig kényelmes kezelőfelületet biztosítanak azok számára, akik a kínált fogalmakból és szolgáltatásokból rendszert szeretnének építeni.

Itt a hangsúly tolódik el: *a létrehozandó rendszer már nem „használja” a külső komponenseket*, és épít belőlük a saját nyelvi környezetében koherens szolgáltatást (forráskódok, vagy script halmaz formájában), *hanem beleépül*: a fejlesztő az eszköz által adott fogalmakat használva, az eszköz felületén keresztül tölti bele saját „tudását”. Ezzel minimálisra csökkenti az alacsony szintű validáció és verifikáció igényét, a felhasznált eszközt hibamentesnek tekinti, és egy jól definiált, kipróbált fogalomrendszerben, transzparens módon rögzíti a feladat elemzésekor feltárt tudást.

Ennek ára a végletes függés: így már terv szinten is a célrendszer fogalmait kell használni; ami ott nincs, azt nagyon nehézkes integrálni, a rendszer pedig csak azon a platformon lesz működőképes, amelyen az eszköz futni tud. Nem egyszerű például egy vállalatirányítási rendszer kiterjesztése mobil funkciókkal; ez vagy nagyon „vékony”, vagy duplikálja a távoli rendszer tudásának egy részét, ami továbbfejlesztés, verzióváltás esetén komoly gondokat okozhat.

Például egy formázott szöveg létrehozásához is szükség van alapfogalmakra, és egy kezelőfelületre, amely a fogalmak alkalmazásával a karaktersorozatból egy nyomdakész dokumentum előállítását lehetővé teszi. Bár a fogalmak általánosak, az adott környezetben létrehozott dokumentum csak ott használható, nincs szabad, oda-vissza átjárás egy HTML lap, egy doc/ odt file, TeX projekt között.

## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

Gyorsan változó eszköz esetén (például Word, főleg, ha nem csak a nagyon általános eszközökkel élünk) még egy verzióváltás is okozhat nehézségeket, nem beszélve a megszokott kezelőfelület megváltozásáról. Ebbe a körbe sorolható a rendszerépítés látványosabb eszközei közé tartozó MatLab, LabView; munkafolyamat-vezérlő rendszerek, mint a Salesforce, ServiceNow; vagy például jelentéstervező, felhasználói felület építő eszközök. A skála másik vége az SAP, amely „mindent tud” a vállalatirányítás lehetséges kérdéseiről, eszköztárát azonban minden bevezetésnél testre kell szabni egy komoly erőforrásigényű projekt keretében.

### 2.1.4 PREZENTÁCIÓ, KOMMUNIKÁCIÓ, INTEGRÁCIÓ

Az informatikai rendszer egy másik rendszerrel kapcsolatos tudásunkat (szerkezet, pillanatnyi állapot, irányítás és tervezés szempontjából) megbízhatóan tárolja, kezeli. Elsődleges célja azonban az, hogy ezt számunkra elérhetővé tegye, ehhez kényelmes felhasználói felületet biztosítson, amikor és ahol szükségünk van rá. Ez egy zárt rendszer tekintetében általában sikerül is – azonban a felhasználói tevékenységekhez, döntéshozatalhoz szükséges adatok nem egy rendszerben érhetők el. Akár egy vállalat informatikai infrastruktúráját, akár a saját számítógépünket, mobil eszközeinket tekintjük, informatikai rendszerek halmazát látjuk. Zárt platformok esetén (mobil, különösen az Apple ökoszisztémában) a monopol helyzetben lévő birtokos eszközöket adhat, bizonyos mértékben kényszerítheti is az alkalmazásfejlesztőket az együttműködésre. Egy szervezet környezetében azonban ilyen nincs, a helyi informatikus csoport válogatja és állítja össze az infrastruktúrát, ahol jellemzően csak az operációs rendszer, böngészők, Java verzió, központi alkalmazáserver stb. szabványos. A telepített vagy fejlesztett informatikai rendszerek egymással nem kommunikáló egységek, adatintegrációjukról helyi fejlesztésű megoldások, például „köztük” szinkronizációs scriptek, „felettük” közös monitoring, reporting rendszerek kerülnek kialakításra. Így jelenik meg az informatikai fából vaskarika, a Big Data terület központi fogalma: a „strukturálatlan adat”.

## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

Ezek az adatok informatikai rendszerekből származnak, számítógépeken futó, vagyis azon a szinten „fehér doboz” szintű modellek kezelik őket – csak a felhasználás helyén és időpontjában „ismeretlen” a szerkezetük. Ennek két jellemző forrása van. Céges környezetben sok (akár több száz) független informatikai rendszer kezel (sajnos részben átfedő) adatokat, gyárt logokat, eseményeket, jelentéseket. Adatreprezentációjuk zárt, a keletkező adat jellemzően bináris vagy részben formázott szöveges állomány tömeg, amelyet azonban tárolni, elemezni kell minőségbiztosítási, felelősségi okokból, vagy a tevékenység optimalizálása érdekében. A másik forrás az, amikor egy kritikus adattartalom elsődleges tárolója egy ügyintézők által kézzel karbantartott adatállomány: a céghierarchiát, munkaidőnaplót, terveket, jelentéseket tartalmazó Excel állomány. Az Excel természetesen „fehér doboz” modellként kezeli az adattáblát, a rajta lévő stílusokat, kifejezéseket, makrókat – de ez a modell nem tartalmazza az adatok mögöttes jelentését, azt szükség esetén ezzel a tudással szinkronban lévő makrók, scriptek, programok tudják csak ismét hozzárendelni.

Következésképpen *hiába tökéletesen strukturált minden adatunk az őket kezelő informatikai rendszerhalmazban, a bennük rejlő átfogó tudás hozzáférhetetlen.* Az utólagos „kibányászt” pedig az teszi lehetetlenné, hogy az erre a célra létrehozott újabb informatikai rendszer egy adott pillanatban fedje le a mögötte álló rendszerhalmazt; annak folyamatos változását követni (különös tekintettel az Excel táblákra) képtelen. Amennyiben pedig létrehozunk egy ilyen rendszert, az a mögötte álló rendszereket fékezi abban, hogy az általuk lefedett külső rendszerek (gyártási folyamatok, törvények, ...) változásait kövessék.

### 2.2 ELVÁRÁSOK EGY JOBB MEGOLDÁSSAL SZEMBEN

A Neumann architektúra tanulságai alkalmazhatók a feltárt általános informatikai rendszer komponensekre, ennek eredménye a fenti négy alapkomponenst tartalmazó megoldással szemben támasztható absztrakt követelménylista. Természetesen az itt felsorolásra kerülő szempontok is közismertek: teljesülésük mértéke határozza meg egy rendszer hosszú távú minőségét, a törvényszerűen érkező változási kérésekhez való alkalmazkodás képességét.

### 2.2.1 SZOFTVER ÉS ADATSZERKEZET KAPCSOLATA

Neumann előtt is voltak programozható számítógépek. Ezekben rendelkezésre állt számtalan aritmetikai, logikai stb. modul, amelyeket egy kapcsolótáblán, kábelek segítségével lehetett egymáshoz kötni.

Egy informatikai rendszer esetében a kapcsolótábla a futtató környezet által kezelhetővé tett memória, a kábelek pedig a tetszőleges programnyelven megfogalmazott utasítások és elem kapcsolatok. Ennek adatszerkezetté alakítása során az történik, hogy a komponensek, azok egymással való kapcsolata és kommunikációja ugyanazzal az eszköztárral adminisztrálható, amellyel magukat az adatokat kezeli a rendszer. Ahhoz, hogy erre képes legyen, a komponensek szerkezetét is ugyanazzal a módszerrel kell „bemutatni” a futtató rendszernek, mint amellyel az adatokat.

Ez nem újdonság, egy objektum orientált programozási nyelven sem különbözik a rendszer által kezelt adatot leíró osztály, a rendszer komponenseinek osztályaitól; sőt a Javához hasonló nyelvek esetében az osztályleírásokat tároló Class is ugyanilyen osztály. Jelen esetben viszont *az az elvárás, hogy a leírás platformtól független, bármilyen környezetben értelmezhető legyen.*

### 2.2.2 MINIMÁLIS KERNEL

A rendszerek stabilabbá tételének általánosan elterjedt iránya az, hogy a környezet gyártója igyekszik a lehető legszélesebb eszköztárat biztosítani a fejlesztő számára. Így az aktuális feladat megoldása kevesebb helyi fejlesztéssel, hibalehetőséggel jár, a kisebb kódméret növeli a transzparenciát. Mellékhatásként azonban „megfertőzi” a megoldást az aktuális eszköztár struktúrájával, totális implementációs függést épít ki vele szemben. A Java nyelv elég régi ennek bemutatására, egy Java program lehet a futtató platformtól független, de a nyelv verziójától nem: például a korábbi (például 1.3-as) collection osztályok teljesen más módon működnek, mint a jelenlegiek. Érdekes tanulság a java.io, java.nio package, az utóbbi újabb és a stream kezelés valódi természetét (a szerző véleménye szerint) sokkal pontosabban reprezentáló, de „túl későn érkezett” eszköztár; illetve a Java chip (Java nyelvű alkalmazás közvetlen futtatására alkalmas hardver eszköz) terve és sikertelensége is.

## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

Minél szélesebb szolgáltatáskészletet nyújt a kernel, annál erősebben kötődik a létrehozásának idején rendelkezésre álló ismeretekhez, önmagán keresztül ehhez az állapothoz köti a rá épülő rendszereket is, amelyek vele együtt avulnak el, és nem képesek követni a kernel esetleges újításait.

A Neumann architektúra erejét, időtállóságát az biztosítja, hogy megtalálta az abszolút minimumot. Például definiálta az Arithmetic Logic Unit (ALU) modult, és azokat a műveleteket, amelyeket tőle várunk. Nem beszélt komplex matematikai szolgáltatáskészletről, nem adott kifinomult eszközöket, csak a legegyszerűbb utasításokat. Az eszköztárak már erre épültek, az alkalmazások az eszköztárakra – az ALU konkrét megvalósítását folyamatosan lehetett fejleszteni, és sehol nem szólt bele a rá épülő rendszerek egymással való kommunikációjába.

Következésképpen *a rendszer kernelnek minimális felületet kell biztosítani arra, hogy a rá épülő komponensekből kommunikálni képes hálózat legyen létrehozható*. Természetesen a kernel igen komplex algoritmus, működéséhez szüksége van egyéb komponensek meglétére.

### 2.2.3 EGYETLEN ADATELÉRŐ ÉS KOMMUNIKÁCIÓS CSATORNA

Az informatikai rendszerek transzparenciájának egyik legsúlyosabb akadálya az, hogy a modulok közötti kommunikáció forráskód szinten zajlik. Lehetnek a tervek bármilyen szépek, lehet az implementáció szigorúan interfész alapú; ha egy hiba kijavításához meg kell hívni egy függvényt, hozzá kell férni egy adathoz, akkor az meg is fog történni. A platform csak akkor lesz ebből a szempontból megbízható, ha ezt képes lehetlenné tenni, ugyanakkor elég használható maradni ahhoz, hogy a programozók ne érezzék készletet a határok áthágására.

Bár ez elég szigorúnak tűnik, a Java nyelv megjelenésekor, C/C++ programozóként ugyanilyen idegenkedéssel fogadtuk a tényt, hogy ebben a környezetben nem férünk hozzá a memóriához, csak a JRE által lefoglalt és felszabadított objektumaink lesznek. Dacára annak, hogy a korábbi programjaink legtöbb, egyúttal legnehezebben megtalálható problémáját éppen a memóriakezelés során elkövetett hibák okozták, amelyeknek egyébként semmi közük nem volt a megoldandó feladathoz.

### 2.2.4 EGYSZERŰ, HÁLÓZATOS, DINAMIKUS DEFINÍCIÓK

Egy objektum definíciója egy szigorúan típusos programozási nyelv esetében forráskódban rögzíti annak szerkezetét. Ez biztonságos megoldás, hiszen fordítási hibát eredményez a téves hivatkozás, az ára viszont az, hogy a szerkezet változtatásához minden olyan rendszert újra kell fordítani, amely erre az elemre hivatkozott. Ez okozza a gyengén típusos nyelvek népszerűségét: igaz, hogy könnyebb bennük hibázni, de sokkal gyorsabb a változások követése.

Ugyanígy problémát jelent, hogy egy adott objektum egyszerre több minőségben is jelen lehet a rendszerben (például egy szerver számítógép, mint hardver eszköz, mint a rajta futó kiszolgáló programok gyűjteménye, mint egy térképen megjelenítendő hely, mint egy 3D-ben lerajzolható objektum stb.) Ezt többszörös örökléssel, aspektusokkal stb. szokás megközelíteni, de a megoldások általában elég nehézkesek (öröklési konfliktusok, bytecode módosítás stb.), és tovább rontják a rendszer átláthatóságát.

A megoldás a „sokarcú” természet elfogadása és közvetlen reprezentációja, ahogy az „oszthatatlan atom” modellt felváltotta a nukleonok felismerése. Bár az „objektum” egy rendszer szempontjából általában rögzített szerepeket tölt be, az őt reprezentáló adatszerkezet mégsem végleges, létezése során komponensek változhatnak, cserélődhetnek. Ezt a rendszer alapszolgáltatásainak támogatni, kezelni kell.

### 2.2.5 NINCS ÚJ OBJEKTUM

A platform szigorúan korlátozza az új objektumok létrehozását. Ez jelenleg is jellemző a nagyobb rendszerekben használt inversion of control container vagy cloud megoldásokra: az objektum példányok a környezettől „kérhetők el”, ez pedig gondoskodik arról, hogy amennyiben a megjelölt objektum még nem létezne, jöjjön létre a megfelelő típussal, és töltődjenek be az adatai a kapcsolódó háttértárból. A megközelítésnek több előnye is van.

- Nem szükséges a hívó kód oldalán ellenőrizni az objektum létezését, illetve ismerni a létrehozás műveletsorát.

## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

- Utóbbi egy konfiguratív eszközre bízva (Java Spring vagy valamilyen cloud szolgáltatás) a keresett objektum konkrét implementációja teljesen elválasztható a rendszertől, így utólag kizárólag a konfigurációt változtatva a szolgáltató modul lecserélhető, feltéve, hogy az objektumok ugyanazt az interfészt implementálják. Ehhez természetesen gondos tervezés, hálón belüli fejlesztés vagy a felhasznált külső modulok tökéletes elrejtése (wrapper interfészek használata) szükséges.
- A művelet szinkronizálható: adott azonosítóval rendelkező felhasználót reprezentáló objektum létrehozása és adatainak betöltése akkor is egyetlen objektum példányt eredményez, ha egy időben érkezett több kérés.

### 2.2.6 NINCSENEK KÖZVETLEN GYŰJTEMÉNY MŰVELETEK

Gyakori, hogy egy műveletet (keresés, gyűjtés, összesítés, csoportos módosítás) nem egy, hanem számos objektumon egyszerre kell végrehajtani. Ehhez minden környezet számos gyűjtemény osztályt (Array, Set, Map, Stack, Pool, ...) kínál, amelyek az adott környezetben közvetlenül használhatók, bejárhatók, összesíthetők, módosíthatók. Ez a módszer azonban több kellemetlen mellékhatással jár.

- A feltételeknek megfelelő objektumok száma előre nem ismert, ami memória problémát okozhat; különös tekintettel arra az esetre, amikor a halmazból végül csak néhány elemet szeretnénk valóban felhasználni.
- Az összegyűjtés időbe telik, az iteráció azután kezdődhet el, hogy minden objektum felkerült a listára – ez lassú betöltés esetén komoly késlekedést jelenthet.
- Amennyiben a felsorolás egy másik objektum része, a helyzet még rosszabb: a listán való iteráció alatt annak tartalma változhat, ami szinkronizációs problémát jelent. Ha viszont az iteráció idejére az objektum zárolható, a rendszer válaszsideje, akár működőképessége kerül veszélybe egy rosszul megírt művelet miatt.

A megoldás a felsorolás típusú adatmezők esetén a tároló objektum elrejtése, a bejáráshoz kizárólag „visitor” interfészt szabad kezelni, amely a platform felé induló egyetlen hívás hatására minden megtalált objektumra visszahívást kap.



## FELADAT – AZ INFORMATIKAI TUDÁS FELTÁRÁSA

Ezzel a kernel oldalán marad a bejárás vezérlése, a művelet kizár minden párhuzamos módosító hozzáférést, ugyanakkor ellenőrizheti a futás idejét, hiba esetén képes gondoskodni a biztonságos kilépésről. Ráadásul, amennyiben a futtató környezet lehetővé teszi, mód van a párhuzamos végrehajtásra.

### 2.2.7 LATE BINDING

Természetesen az alkalmazást felépítő komponensekhez tartozik forrásnyelvi implementáció, ez azonban nem az objektum része. Egy vektorgrafikus ábra bármely platformon ugyanazokkal az absztrakt fogalmakkal dolgozik (sokszög, ellipszis, ...), amelyeket ugyanolyan paraméterek írnak le, belőlük ugyanúgy kell rajzot készíteni. Az ábra elemeihez csak a legutolsó pillanatban kell az aktuális futtató környezetben (például Java vagy iOS Objective C nyelven megírt, lefordított) implementációt csatolni. Az is elképzelhető, hogy a futtató környezet csak az első megjelenítés pillanatában tölti le a kirajzolásért felelős könyvtárat egy megbízható forrásból, integrálja a futtató környezetbe (könyvtár inicializálás), hozzákapcsolja az ábrához, majd átadja neki a vezérlést. Így az ábra meg tud jelenni minden 2D rajzolásra alkalmas környezetben.

### 2.2.8 TRANSPARENS ALGORITMUSOK

Az előző példát folytatva alkalmazás rétegek jelennek meg. Például az iménti 2D rajzoló könyvtár alkalmas egy általános formázott szöveg – így pedig tetszőleges dokumentum megjelenítésére. Ha ezt kiegészítjük egy alacsony szintű eseménykezeléssel (egérmozgás, kattintás, karakter), egy interaktív grafikus felhasználói felület kezelő komponensre kapunk. A komponens feladata csupán egyszerű eseményekre reagálva megfelelően paraméterezett, előre tudható parancsok kiadása a rajzoló komponens „nyelvén”, ezt már ma is adatszerkezet formájában tároljuk számtalan eszközben (munkafolyamat vezérlők, script engine-ek stb.). Amennyiben rendelkezésünkre áll egy adott platformon a kifejezések és algoritmusok kezelésére alkalmas modul, és a 2D rajzolás eszköztára, a közös fogalomkészlettel létrehozott tetszőleges felhasználói felület minden további beavatkozás nélkül futtatható lesz. A példáért nem kell messzire menni: kívülről nézve ez a JavaScript motorral egybeépített böngésző – csak ott a tudást elrejt a böngészőtől, HTML és JavaScript verziótól függő forráskód.

### 3 MÓDSZER – ÖNHORDÓ, MINIMÁLIS PLATFORM

A korábbi fejezetek elemzései alapján elkészíthető az elvárt transzparens, platformfüggetlen rendszerépítést lehetővé tevő alapvető komponensek terve. Ezek a komponensek szükségképpen kívül esnek az eddig tárgyalt szempontok körén, átértelmezik a programozási nyelv, futtató környezet, és a rajtuk megvalósítható programok szerepét és alapvető definícióit. Pontosan emiatt bármilyen környezetben azonos szolgáltatáskészlettel megvalósíthatók (számos részleges Java, C#, Objective C és JavaScript implementáció készült az elmúlt évek során). A megoldásnak eredetileg Dust Framework nevet adtam, azonban célszerűbb Dust Platform néven hivatkoznom rá, mert a programozási környezet megteremtéséről a hangsúly átkerült az általánosabb tudásreprezentációra.

#### 3.1 AZ „OBJEKTUM” FOGALOM ÖSSZETEVŐI

*Az objektum orientált megközelítés legsúlyosabb hibája maga az objektum fogalom, amely valójában három elkülöníthető, csak adott pillanatban összetartozó elem kényszerű egysége.* Ezek független kezelése alapjaiban teszi rugalmasabbá a rá épülő rendszereket.

##### 3.1.1 MODELL

A **Modell** az objektum technikailag a benne egyedi mezőnévvel azonosítható adatok logikai egysége. Szigorúan típusos nyelvek esetén a szerkezet fordítás időben rögzített, a forráskódban a mezőhivatkozás név és adattípus szerint ellenőrzött (Java, C++, ...). Scriptnyelvek esetén, ahol interpreter dolgozza fel a forráskódot (vagy futásidőben történik a fordítás), az objektum szerkezete változhat, a mezőkre nem csak fordítási azonosítóval, hanem szöveges formában a nevével is lehet hivatkozni. A kettő között átmenetet jelent az Objective C, amely szigorúan típusos, de a szöveges hivatkozás is része a nyelvnek; ehhez hasonló műveletek a Java reflection csomagjával végezhetők.

Egy „dolog” egy rendszerben több „arccal”, tulajdonságcsomaggal is rendelkezhet. Ezt az osztályok leszármaztatásával igyekszünk közelíteni, azonban ez csak valódi fa struktúrák esetén kényelmes, egyébként a többszörös öröklés problémáiba ütközünk (az ősoosztályok mezőnevei lehetnek azonosak).

A valóság reprezentációja ennél is bonyolultabb lehet: az objektum „arcai” létezése során változhatnak. Egy személy élete során típusokat „szerez” és „veszít”, például orvos lesz, és meg kell találnunk őt amikor az ügyeletes orvost keressük. Ez is kezelhető bizonyos szintig az aspektus-orientált programozás fogalmaival, az Objective C nyelvi eszközökkel is rendelkezik ennek kezelésére.

Az „objektum, mint egy külső egyed informatikai képe”, és az „adatok logikai egysége” nem egyértelmű és végleges kapcsolat. *Az egyedet különböző modellek halmaza alkotja, ennek ellenére ez nem „has a” hanem „is a” kapcsolat, a Személy egyidejűleg Orvos, vagy akár Ügyeletes is (ez másként fogalmazva, a „composition over inheritance” tervezési minta, csak megkerülhetetlen formában).*

Ez a struktúra teljesen megszünteti a többszörös öröklés problémáját: a típus megkövetelheti egy másik típus jelenlétét, például Orvos csak Személy, Ügyeletes pedig csak Orvos és Személy lehet. A konfliktus nem jelentkezik: mindegy, hogy az Orvos vagy az Ügyeletes típuson „keresztül” érjük el a Személy modellt, csak az biztos, hogy az őket tartalmazó modell halmazban jelen van. Az azonos mezőnevek sem jelentenek problémát, mert adat hivatkozásnál mindkét szintet, a Típus és a Mező azonosítóját is meg kell adni.

### 3.1.2 ENTITÁS

Az **Entitás** felel meg a „külső egyed informatikai képe” fogalomnak, vagyis egy konkrét modell halmaznak. Önállóan nem rendelkezik semmilyen adattal, ugyanakkor a rendszer szempontjából „atomi egység”: ha egy egyed képét keresem valamely modell adatai, vagy rá mutató hivatkozás alapján, mindig az entitást kapom vissza az aktuálisan elérhető modell halmazával. Természetesen nincs akadálya, hogy bizonyos környezetekben egy entitás kiegészítő modellekkel rendelkezzen. Például a Személy entitás egy globális rendszerben egyedi; egy kórház viszont ehhez az entitáshoz lokálisan tárolhat privát modelleket. Az entitás betöltésekor a globális adatok az elsődleges külső forrásból érkeznek, de kiegészülnek a helyi adatokkal, és a belső rendszer mindezt homogén adatszerkezetként látja, és képes hozzáférni bármelyik modelljének adataihoz.

*Az entitás élete során a hozzá kapcsolódó modellkészlet folyamatosan változhat, természetesen a függési szabályok ellenőrzése mellett – egy modell kivételével, amely az entitás létrejöttkor elsődleges típusaként lett megjelölve. A Személy entitás ezt a tulajdonságát soha nem veszítheti el (technikai szempontból ez jelöli ki az elsődleges perzisztens tárolót, amely az entitás „gyökér” adatainak, szinkronizációs és zárolási állapotának tárolásáért felel). Ezt azért fontos figyelembe venni, mert az entitások aktuális informatikai rendszerek felett álló egyedek: a Dust Platform keretein belül minden egyes Személy elsődleges adatait pontosan egy tároló (ebben az esetben természetesen egy megfelelő elérhetőséggel és védelemmel rendelkező központi adatbázis) őrzi. Minden Dust Platformra épülő rendszer ehhez a tárolóhoz fordul, amikor a Személyre vonatkozó adatokat szeretne elérni.*

### 3.1.3 SZOLGÁLTATÁS

Az objektum fogalom harmadik oldala a viselkedése. Ez szokványos programozás során az elvárt szolgáltatást az adott nyelvi környezetben megvalósító forráskód megírását jelenti. A rendszer kialakításához azonban ez nem szükséges, hiszen kívülről nézve érdektelen, hogy egy adott komponens *hogyan* fogja végrehajtani a műveletet – csupán az az érdekes, hogy milyen konfigurációval dolgozik, és *milyen szolgáltatásokat kínál* a környezetnek. Fekete doboz modellre van szükségünk, ennek felel meg az „interfész” fogalma – amely azonban a Dust Platform számára nem megfelelő, forrásnyelvi konstrukció.

A *Szolgáltatás* ugyanúgy logikai szempontból összetartozó **Parancsok csoportja**, ahogy a modell adatoké. A szolgáltatás tartalmazhat egyetlen parancsot (Processor – Process), vagy többet (Stateful – Init, Release). Vannak parancsok, amelyek esetében az információ kizárólag az esemény bekövetkezése, ilyen például az Init / Release parancs, ahol a művelet feladata az entitás beállításoknak megfelelő erőforrás managementje (például egy adatbázis konnektor inicializálja/felszabadítja a connection pool-t, socket listener, job manager, stb.). Más esetben a művelet egyes paramétereit a hívó küldi (adatbázis lekérdezés, egér pozíció stb.), a feldolgozó kód ezek meglétére számít. A deklaratív megoldás egyszerű: a parancs leírása tartalmazza az elvárt típusok listáját.

## MÓDSZER – ÖNHORDÓ, MINIMÁLIS PLATFORM

Az üzenet maga szintén egy kernel által létrehozott Üzenet típusú entitás, amely az elvárt típusoknak megfelelő modelleket biztosan (egyéb modelleket opcionálisan) tartalmaz. Az Üzenet típus a platform által kezelt adatokat (például a parancsot, futási állapotot, hívási módot stb.) tartalmazza.

A megoldás rugalmassága szembeűnő.

- Nem jelent problémát több visszatérési értéket kezelni.
- Lehetséges a közvetlenül hívott komponens által nem kezelt kiegészítő adatokat küldeni (például: egy szöveg formázó rutin ugyanúgy építi a képet egy hierarchia bejárása révén, viszont az erőforrás szolgáltatónak tudnia kell, milyen nyelven kell válaszolnia).
- A kommunikáció teljesen transzparens (proxy, cache, validation kernel szinten).
- Esetleges struktúra változás hatása kizárólag a végpontokon (üzenet előállítás és feldolgozás) jelentkezik, a továbbító infrastruktúra számára indifferens.

A szolgáltatások között ugyanolyan hierarchia látható, mint a típusok között, például egy karakter eseménysorozat fogadására képes egyszerű komponens és egy sor orientált stream kezelő összekapcsolható ugyanabban az entításban, utóbbi igényli az előbbi meglétét. A sor fogadó rutin a kapott szöveget karakterekre bontva továbbítja a karakter kezelőnek, amely lehet file író, vagy parser. Itt is érvényesül a „composition over inheritance” elv, a megvalósítás egyszerűsége erős vonzást jelent a valódi atomi szolgáltatások kialakítása irányában. Szemben a megszokott, idővel „elbonyolódó” implementációval, itt a beérkező új szolgáltatások és kérések inkább kirajzolják a határvonalakat a szolgáltatás összetevői között, és mivel a refactor igen egyszerű, a szétválasztás költsége a rendszer „életkorától” függetlenül alacsony marad.

Ugyanígy létezik függőségi viszony szolgáltatások és típusok között. Természetes, hogy egy-egy szolgáltatás nem csak a kapott paramétereket, hanem a hordozó entitás adatait is használja: ilyen például egy adatbázis konnektor működési paraméterei (szerver URL, ...). Ilyenkor a szolgáltatás kötelezőnek jelöli az igényelt adatokat hordozó típust.

### 3.1.4 AZ ÚJ MEGKÖZELÍTÉS ELŐNYEI

A fentebb ismertetett konstrukcióval első pillantásra a megszokott eszköztárhoz képest sokkal bonyolultabb módon ugyanazt az eredményt kaptuk: a rendszer adattartalommal és szolgáltatásokkal rendelkező komponensek hálózatával írható le. Ez a környezet azonban:

- Szerkezetében nem kötődik semmilyen programozási nyelvhez, környezethez, eszköztárhoz – éppen ellenkezőleg: minden függést azonnal, az első komponens szintjén elrejt, kívülről kizárólag egy absztrakt, homogén adat és képességthalmaz látszik.
- Teljes mértékben, a legalacsonyabb szinten támogatja a leképezett „dolgoz”, illetve a leképezés folyamatának alapvetően dinamikus természetét: nem csak a modellkészlet ellenőrzött, időnkénti módosulását, hanem a modellek szerkezetének változását is követni tudja – akár futásidőben, memóriában tárolt entitások esetén is.
- A rendszer teljes hálózata, beállításai, működése leírható az alkotóelemeit reprezentáló entitások segítségével, a rendszer „dokumentációja”, „terve” valójában egy olyan adatszerkezet, amely tetszőleges platformon, ahol a felhasznált szolgáltatások implementációi elérhető, közvetlenül (pontosabban: kizárólag ilyen módon) futtatható.
- Az entitások kezelése egy aktuális rendszer felett álló globális környezetet teremt, hasonló az internet címzési rendszeréhez. A Dust Platformra épülő rendszerek integrációja nem kérdés: minden adatot, szolgáltatást, implementációt az itt megadott közös forrásokból, ellenőrzött módon, közösen érnek el – de van mód privát definíciók és adatok kezelésére is.

### 3.2 META RÉTEG

Ahhoz, hogy a fenti entitás kezelő környezetet biztosítani tudjuk, természetesen szükség van egy olyan „nyelvre”, amelynek segítségével a típusok, szolgáltatások, kapcsolataik stb. leírhatók. Ez hagyományosan a rendszermodellező eszközök területe. Léteznek szoftver-specifikus UML tervezők, mint a Rational Rose, Enterprise Architect, vagy az ArgoUML; illetve általános Domain Specific Language (DSL) szerkesztők, mint a JetBrains MPS.

## MÓDSZER – ÖNHORDÓ, MINIMÁLIS PLATFORM

Ezek eszközöket adnak objektum osztályok, kapcsolataik, mezőik, stb. absztrakt (de csak általuk értelmezhető) leírására, illetve különféle környezetekbe való transzformációjára. Forráskódot generálnak számos célnyelvre; felhasználói felület kialakítását támogatják, adatbázis Data Definition Language (DDL) parancsokat hoznak létre stb.

A Dust Platform számára az utóbb felsoroltak kiegészítő szolgáltatások: „valamilyen komponens” majd gondoskodik a forráskódokról, adattárolásról, továbbításról; marad az elsődleges feladat: a rendszer metaszerkezetének leírása. Viszont ez a leírás ugyanolyan eszközökkel (**Típus**, **Mező**, **Szolgáltatás** stb. elsődleges típusú entitásokkal) elkészíthető, mint bármi más eleme a rendszernek! Következésképpen a Meta réteg teljesen vékony, lényegében a Meta leíráshoz szükséges típusokat, konstansokat tartalmazó entitás halmazt jelenti.

Természetesen Meta mező és típusnevek, a megoldás szerkezete felhasználásra kerül a kernelben, szerkezete implementáció szinten rögzített; amennyiben ez változik, minden rendszer típusleírásait tartalmazó entitások szerkezete is módosul. Ezt verziókezeléssel, automatikus migráció támogatással lehet kezelni; a felhasználói szintű programokig ennek a rétegnek a változása nem jut el. Ez a megközelítés lehetővé teszi tehát azt, hogy egy Dust Platform alkalmazás típuskészletét futásidőben változtassuk.

- Amennyiben olyan entitást próbálunk betölteni, amelyben ismeretlen modellek is találhatóak, a típus név alapján ez szintén betölthető, majd az eredeti betöltő rutin folytathatja a működését, hiszen innentől a kernel már ismeri az új szerkezetet. A művelet természetesen transzparens, a kiváltó kód nem is értesül a folyamatról.
- Lehetséges futásidőben az „alaptípusokkal” teljesen analóg módon kezelt helyi típusok létrehozására. Tipikus példa: tetszőleges Excel táblázat oszlopaiból beolvasás közben típus entitást hozunk létre, majd ilyen típusú entitásokként olvassuk be a tábla tartalmát. Ezek az entitások ugyanúgy kezelhetők, szerkeszthetők és menthetők; konfiguratív eszközökkel a betöltés után azonnal őket kezelő algoritmusokat, felhasználói felületeket stb. hozhatunk létre forráskód írás, fordítás, telepítés, ... nélkül.

## MÓDSZER – ÖNHORDÓ, MINIMÁLIS PLATFORM

- A típus entitások szerkesztése révén át tudjuk alakítani a kérdéses adatstruktúrát is, futásidőben módosítva a már meglévő, ilyen modellt tartalmazó entitásokat. Ez természetesen veszélyes művelet, de kialakítható a megfelelő hozzáférésvédelem, illetve ellenőrző, validáló szolgáltatáskészlet, amely az adminisztrátort figyelmezteti a transzparencia miatt pontosan azonosítható veszélyekre. Az egyetlen kivétel az, hogy a forráskódba „nem látunk bele”, tehát minden olyan kód, amely a módosítás alatt álló típusra, szolgáltatásra hivatkozik, „veszélyeztetett”.

Összességében a Meta réteg lehetőséget ad a rendszerünk adattípusainak, szolgáltatásainak megtervezésére: homogén módszerrel írunk le minden típust és szolgáltatást, beleértve a kernel vagy a Meta réteg elemeit; a Meta entitások közös tárolása révén globális adat és szolgáltatástár elemeiből építkezhetünk, saját komponenseinket ezen a felületen megoszthatjuk.

Ma is biztosak lehetünk abban, hogy rendkívül összetett, egymástól és platformoktól függő eszköztárakban minden általunk igényelt adat és szolgáltatás már számtalan módon megtervezésre és implementálásra került, csak éppen nem használható vagy integrálható a rendszerünkbe. A Dust Platform Meta rétege egy igen gyorsan bővíthető, teljes mértékben integrálható atomi elemekből álló, globális eszköztár kialakítását teszi lehetővé. Az egyetlen „függése” az entitás-modell-szolgáltatás fogalmak szétoztása, illetve a Meta entitások névkonvenciója. Ez igény esetén adapterekkel vagy saját kernel implementációval testre szabható, kiterjeszthető (például speciális adattípusok támogatására), nincs akadálya a standard (például UML) fogalmak Dust Platformba történő bevonásának. A *Meta komponens* viszont nem kiterjeszthető, ennek *feladata az objektumleíráshoz szükséges abszolút minimális fogalomkészlet rögzítése*, minden további elem szükségtelen függést hozna a rendszerbe.



### 3.3 ALKALMAZÁS ÉPÍTÉS

Az iménti eszközök és fogalmak lehetőséget adnak egy tetszőleges rendszer elemzésére, a komponensek adattartalmának és szolgáltatáskészletének leírására. A fogalmak globális tárolóban elhelyezhetők, mások által létrehozott komponensek újrahasználhatók, beleértve a kernel leírásához és működéséhez szükséges elemeket is. Az aktuális tudás teljes mértékben adatszerkezet szinten maradt, így mód van a komponensek ellenőrzött továbbfejlesztésére, átalakítására, alternatív párhuzamos megoldások létrehozására, cseréjére.

A cél azonban egy futó informatikai rendszer felépítése. A Meta réteghez hasonlóan az alkalmazás fogalmai, ezek szerkezete és kapcsolata is általánosítható, leírható. A futtató környezet természetesen ezeket is használja, tehát a következő típusok a kernel implementációjához kötöttek, illetve bár számos kiegészítő szolgáltatás is elképzelhető lenne, ez a komponens csak az alkalmazás építéshez szükséges minimális fogalmakat tartalmazza.

#### 3.3.1 UNIT

*Egy Meta réteghez tartozó Típus, Szolgáltatás jellemzően nem önállóan, hanem egy szorosan összetartozó csoport részeként létezik, amelyek egy részrendszer leírásához együtt szükségesek, ez a csoport a **Unit**. Ilyen például a grafikus primitívek csoportja, a Meta komponens tagjai stb. Másik csoportképző lehet, amikor egy külső eszköztár konfigurálható objektumait szeretnénk a Dust Platform környezetébe integrálni – ilyen lehet az UML fogalomkészlete, az email vagy egy adatbázis konnektor integrációja.*

Természetesen a unitok is épülhetnek egymásra. Nyilvánvaló, hogy a „felhasználó”, vagy „hozzáférési jog” fogalmai általánosak és közösen használhatók az email és az adatbázis unit által, ehhez a Meta elemeihez hasonló függés reláció alakítható ki. Természetes, hogy ahogy az entitás esetén a benne szereplő modellek sem hierarchikusan, hanem egymás mellé rendelt viszonyban vannak, a unitok halmazát a hivatkozott unitokkal rekurzívan bővítve olyan halmazt kapunk, amely minden szükséges fogalmat és szolgáltatást tartalmaz.

### 3.3.2 MODUL

A *Modul* a *unit implementáció szintű párja*. Tartalma a unitban definiált szolgáltatásokhoz tartozó futtatható objektumokból alkotott könyvtár (például Java jar file, dll, so, stb.); illetve megfelelő típusú entitások hálózata formájában az az ismeret, amelynek segítségével a modul betöltő a szolgáltatás egy példányát az őt hordozó entitáshoz képes kapcsolni. Amennyiben ez a modul egy külső szolgáltatást integrál (adatbázis konnektor, külső eszköz meghajtó, formátum kezelő csomag stb.), annak futtatható formája is része a modulnak, akár binárisan, akár egyértelműen azonosítható elérési útként.

A kapcsolat leírásának egy része általános a bináris objektum implementációs szintű tulajdonságai vagy a vele kapcsolatos elvárások miatt. Például

- *Stateless* objektum esetében ugyanaz a példány kapcsolódik minden előforduláshoz, mert nincsenek egyedi adatai, vagy forráskód szintű globális szinkronizáció szükséges a műveletek között.
- *AutoInit* objektumnál nem az első hívás, hanem az entitás és a szolgáltatás kapcsolatának létrehozása pillanatában (ez jellemzően a betöltés) példányosítani, csatolni és inicializálni kell az objektumot. Tipikus példa a „listener” server objektumok (mint egy Jetty alkalmazáserver): amennyiben ilyet tartalmaz a rendszer, elvárjuk, hogy az indulás pillanatában működjön, és fogadja a beérkező kéréseket.

Ezen kívül természetesen az adott cél környezet modul betöltő rendszere rendelkezhet egyedi paraméterekkel, amelyek nem a közös modul komponensben, hanem az erre a célra létrehozott speciális unitban helyezhetők el.

A modul jellemzően egy szokványos könyvtár projekt forráskódjainak lefordított halmaza, önálló futtatáshoz szükséges elemeket, pl „main” nem kell tartalmazniuk, a fordítás során a megjelölt környezetben betölthetőnek kell lenniük – JRE verzió, 32/64 bites architektúra, operációs rendszer típus és verzió. A modulon belüli forráskódokra egyéb megkötés nincs, a programozó szabadon használja a fejlesztőeszköz szolgáltatásait – viszont más modulok felé nincs átjárás.

Bizonyos esetekben, erőforrás vagy sebességkritikus környezetben szükség lehet a unitok által meghúzott ideális határokon túl szorosabb, forráskód szintű együttműködésre. Ilyenkor a modul több unitot is implementálhat, de különösen figyelni kell a modul lista összeállításánál: egy unit implementációját egy alkalmazásban csak egy modul tartalmazhatja.

Összességében: *a modul teremt kapcsolatot a unit és a cél platform, esetleg további platformfüggő tartalom között.* A Dust Platform célkörnyezeti implementációja tartalmaz egy modul kezelő komponenst, amely a modult, mint atomi egységet be tudja tölteni, az pedig minden további egyedi beavatkozás nélkül teljesen működőképes.

### 3.3.3 APPLICATION

*A fenti fogalmakat használó **Application** képes egy tetszőleges alkalmazás definiálására, amely alábbi elemeket tartalmazza:*

- Unitok felsorolása. Itt elvárható, hogy minden hivatkozott unit szerepeljen a listában, hiszen ennek ellenőrzése nagyon egyszerű. Idővel a unitok több verziója is elérhető lehet a Dust Platformon, az alkalmazás építése során a szükséges halmaz függéseinek verzió szinten is egyeztetni kell.
- Futtató platform megadása. Ezt a futtató környezet ellenőrzi az alkalmazás betöltése során, így biztosítható, hogy az adott gépen ez az alkalmazás valóban képes lesz futni – amennyiben nem, megfelelő figyelmeztetés adható. Természetesen az alkalmazás definíciója tartalmazhat több platformra érvényes beállításokat, ilyenkor a lista minden elemét ellenőrzi, és a legjobb egyezést választja.
- Modulok listája: a platform „alatt” helyezkedik el, egyértelműen azonosítja a unit halmaz minden elemének implementációját tartalmazó modulkészletet. A modulok saját konfigurációjában adott futtató platformmal kompatibilis kell legyen az itteni tároló, ez a konfiguráció összeállítása, illetve a rendszer indítása során ellenőrizhető.
- A rendszert alkotó entitások hálózata: a unitok által megadott típusú modellekből és szolgáltatásokból összeállított, felkonfigurált rendszer.

- Indító parancsok listája: rendszer komponenseinek megfelelő sorrendben történő elindításáról gondoskodik. Egyszerű konzolos alkalmazás esetén itt egy megfelelően felkonfigurált paraméter beolvasó és ellenőrző komponens a parancssor paramétereit az entitás hálózatba másolja, majd amennyiben ez nem jelez hibát, a következő parancs(ok) hatására a végrehajtó komponensek elvégzik a dolgukat, és a rendszer leáll. Egy kiszolgáló rendszer esetében itt az infrastruktúra elemeinek adott sorrendű indítása (adatbázis kapcsolat, értesítési és naplózó csatornák inicializálása stb.) után a parancsokat fogadó felület (felhasználói felület, konzol olvasó, alkalmazásszerver stb.) inicializálódik. A rendszer a leállításig működik.

### 3.4 FUTÓ RENDSZER

A korábbiak ismeretében látható, hogy a rendszer működését biztosító forráskódok (például Java, C++, Objective C objektumok) lényegében hasonló szerepűek, és viszonylag egyszerűek. A rendszer szerkezetét ugyanis nem forrásnyelvi konstrukciók, hanem a Meta és Application szintű leírások hordozzák, nincs szükség konfigurációk, paraméterek egyéni beolvasására, nincs számtalan függvény folyamatosan bővülő, változó paraméterkészlettel. Nincs „main” rutin sem, a kód lényegében üzenetek fogadását, ezzel kapcsolatos adatmanipulációt és további üzenetek küldését végzi, valójában inkább script nyelvhez hasonlít, leszámítva a komplex külső rendszereket (például egy GUI keretrendszert, bináris adatformátumot, standard szolgáltatót) elrejtő, szükségszerűen bonyolult modulokat.

#### 3.4.1 ESEMÉNYEK, ÜZENETEK

A keretrendszernek természetesen meg kell találnia és meghívnia a szolgáltatást végző függvényt. Ehhez szükség van egy, az adott nyelvvel kompatibilis szigorú névkonvencióra, amely egyértelművé teszi a nevet osztály szinten, a definícióból forráskód is generálható (Java esetén interfész, benne azonosítók és függvények); a saját kódnak a kapcsolódó szolgáltatáshoz rendelt interfészeket kell implementálnia. Természetesen amennyiben a definíciós szinten változás történik, az a generált forrás változása miatt fordítási hibát eredményez; ez természetes viselkedés, a refactor változásait forráskódban is követni kell.

### 3.4.2 ALGORITMUSOK ÉS KÖRNYEZETÜK

Az objektum orientált nyelvek legfontosabb szolgáltatása, hogy a programozó által megvalósított algoritmust egy adatobjektum kontextusába helyezi, ez alkotja a művelet „adat környezetét”, amelynek objektum típusú memberváltozói keresztül természetesen távolabbra is elérhet (`car.location.latitude`). Mivel a Dust Platform átvette a rendszer struktúra reprezentáció feladatát a programozási nyelvtől, ezt a szolgáltatást biztosítani kell a kód számára. Ez a szabadság lehetőséget ad arra is, hogy ne csak a „saját” környezetet, hanem más, hasonlóan kezelhető belépési pontot adjon. A leginkább szembeűnő példa, hogy a függvények paraméterlistája üres: az üzenet a futó kód kontextusán keresztül érhető el. Így az üzenet paraméterlistájának bővítése nem eredményez refactor igényt; belső függvények hívásakor paraméterek átadásával nem kell foglalkozni, minden kód ugyanúgy látja az aktuális környezetet.

- `This` – az az entitás, amelyhez a most futó kódot tartalmazó objektum példány a megadott szolgáltatás biztosítása céljából kapcsolódott. Így éri el a kód az üzemmód-jellegű beállításokat, például sor orientált szövegkezelőnél a sorvége, szöveg vége karakterek. Ezeket természetesen egy „Stateful” komponens beolvashatja az `Init` parancs hatására lokális változóba, amennyiben az adat végleges besorolást is kapott – egyébként minden használatnál érdemes újraolvasni. Amennyiben az implementáló kód „stateless”, a paramétereket minden esetben kötelezően újra kell olvasnia, illetve nem ajánlott állapotot kezelő memberváltozókat használnia.
- `Param` – a feldolgozandó üzenet paramétereit hordozó entitás. Ennek biztosan része a `Message` modell, amelyben a feldolgozás állapotáról, eredményéről lehet tájékozódni vagy hírt adni a futtató környezetnek. A valós adatkommunikáció (`in`, `out`, `in/out` paraméterek) az üzenetre nézve specifikus modell(ek)ben található.

- `Channel` – a kód általában más komponenseket is felhasznál (meghív) a feldolgozás során. Ezek szerepelhetnek a szolgáltatás definícióban (a „`this`” entitásban ezzel az objektummal párhuzamosan léteznie kell), de lehet adat member a „`this`” vagy „`param`” entitásban is. Mivel a definíció tartalmazza a kapcsolódó szolgáltatásokat, a kódban fel lehet építeni a címzettnek szóló parancs entitás adattartalmát, majd a megfelelő szolgáltatással elküldeni, a választ pedig kiolvasni.
- `Context` – bizonyos esetekben a kommunikáció indirekt, több lépcsőn keresztül zajlik. Ilyen lehet például az adott pont „alatt” használni kívánt nyelv, amely eltérhet a rendszer alapértelmezésétől.

### 3.4.3 KERNEL INTERFÉSZ

Mivel a rendszer struktúrájáról és az aktuális kontextusról a futtató környezet gondoskodik, nincs különösebb szükség a nyelv objektum orientált képességeinek kihasználására, ugyanolyan jól használható egy script nyelv, vagy akár konfiguratív algoritmus kezelő komponens is. Ennek megfelelően nincsenek komplex ösosztyúk, nem javasolt és támogatott a rendszer állapotának forráskód szintű tárolása, nincsenek bonyolult osztályok egyes szolgáltatások használatára. Mindez ugyanis a kernel részeként minden cél platformra lefordítandó, a kernel fejlesztése esetén fordítási problémákat és a refactor, verzióváltás késleltetését jelentené.

A kernel a nyelvi „call stack”-hez hasonló, „send stack”-et kezel, az aktuális thread-hez mindig tudja a hívó kontextusát, statikus függvényein keresztül mindig, bármilyen forráskód szintű hívási lánc végén elérhető minden adat, amely az aktuális kontextusban a kód számára látható. Éppen ezért nem javasolt adatok kiolvasása, letárolása vagy paramétereken keresztüli átadása, mert ez a rendszer fejlődése során folyamatosan egyre bonyolultabbá válik.

A kernel oldaláról nézve a forráskód működése folyamatosan követhető, hiszen minden adathozzáférés és üzenetküldés során vezérlést kap. Így transzparens módon van lehetőség olyan tevékenységekre, mint a jogosultság ellenőrzés, zárolás és a szál felfüggesztése, közvetlen kommunikáció proxy műveletekre cserélése a forráskód tudta, módosítása nélkül.

## MÓDSZER – ÖNHORDÓ, MINIMÁLIS PLATFORM

Az alap kernel interfész három szegmensre osztható.

- Adathozzáférés: adat olvasás, módosítás, visit, az adat azonosítója, művelet, érték és bejáró kód megadásával.
- Üzenetküldés: a csatorna és a parancs megadásával (az adatokat a korábbi műveletekkel lehet „betölteni” a parancs entitásba).
- Azonosító előállítás: jellemzően generált konstansok, de a forráskódban előfordul több lépéses hozzáférési utak használata, itt stringek helyett a belőlük generált azonosító objektumokat használ a kernel (egyediség, optimalizálás miatt).

Erre épül még néhány olyan egyszerű szolgáltatás, amelyek a kernel implementációjának, a Meta (és néhány másik) unit definícióinak ismeretében szokványos műveletek végrehajtását teszi lehetővé (naplózás, üzenet műveletek, általánosan igényelt adatelérés). Minden további szolgáltatás kívül esik a kernel hatáskörén.

### 4 MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A következőkben felsorolásra kerülő szolgáltatások túlnyomó többsége minden informatikai rendszerben jelen van, a „hordozó infrastruktúra” részei, kialakításuk és folyamatos módosításuk a fejlesztés és fenntartás erőforrásigényének számottevő részét alkotja. Igaz, hogy ezek a szolgáltatások nem a platform részei – ugyanakkor a működése rájuk is épül. Például ahhoz, hogy a kernel képes legyen használni az általa kezelt adatok Meta leírását, azokat természetesen azt be kellett olvasnia valahonnan; a fejlesztést támogató forráskód generáláshoz szükség van szövegek kezelésére stb.

Természetesen az itt felsorolt feladatok mindegyikének teljes megoldása messze meghaladja egyetlen ember teljesítőkéességét és szaktudását. Ugyanakkor állítható, hogy a Dust Platform fogalmainak felhasználásával ezek jelentősen könnyebben kezelhetők; a kernel futtatásához szükséges elemek legalább referencia szintű implementációja (bizonyos esetekben több platformon is) elkészült, rájuk működő alkalmazást sikerült építeni.

#### 4.1 ÁLTALÁNOS, GYAKORI PROBLÉMÁK MEGOLDÁSA

##### 4.1.1 ADATKEZELÉS

Hagyományos fejlesztés során a rendszer által kezelt adatokról a cél nyelven (vagy akár több nyelven is) osztály leírást készítünk, mert csak így tudunk rajtuk műveleteket végző forráskódot létrehozni. Ez a kód a létrehozása pillanatában érvényes ismereteinket rögzíti, változtatása nehézkes, fordítani, telepíteni kell. Bár nagyobb rendszerek esetén ezt a kódot tervezőeszköz generálja, akkor is a rendszer számottevő százalékát alkotja, és mivel minden kezelő kód rá épül, módosításai komoly mellékhatásokkal járnak. Ha a kódot generáló eszköz nem képes a rá épülő teljes rendszer minden forráskódjának módosítására a refactor során, akkor nem is érdemes elsődleges módosító eszközként használni.

Külön figyelmet érdemel, hogy amennyiben forráskód „fedi” az adatot, onnantól minden további rájuk épülő szolgáltatás (kifejezések, megjelenítés, jelentések stb.) szintén forráskódban készül, a konfiguratív eszközök (például script, template engine) háttérbe szorulnak.



A Dust Platform esetében nem készül semmilyen forráskód az adatok leírására. A rendszer kiegészítő szolgáltatásai (megjelenítés, validálás, stb.) szintén adat szinten létrehozhatók, ami lehetővé teszi egy kijelölt mezőre, típusra hivatkozó komponensek megkeresését, a változtatások mellékhatásainak elemzését.

Bizonyos alkalmazás típusok, például Extract, Transform, Load (ETL) eszközök esetén kritikus az adattípusok flexibilis kezelése. A Dust Platform használata során semmilyen különbség nincs a kernel, az alkalmazás rögzített, illetve az alkalmazás segítségével futásidőben létrehozott típusok, illetve az ilyen típusú adatobjektumok (közönséges entitások) kezelése között. Következésképpen az előzetesen rögzített ismeretek és megkötések nélküli tudásreprezentáció területén *a Dust Platform alapszolgáltatásai a szokványos fejlesztési módszertan számára megoldhatatlan feladatokat képesek kezelni.*

### 4.1.2 ADATTÁROLÁS ÉS TOVÁBBÍTÁS

Hagyományos fejlesztési módszertan esetén az infrastruktúra központi része a rendszer objektumainak „fizikai kezelése”.

- Perzisztens tárolás: bináris vagy szöveges állományokban, adatbázisban.
- Továbbítás: szerver-kliens alkalmazásnál a kommunikációs formátum kialakítása, szerializáció módszertana különös tekintettel az egymásra hivatkozó objektumokra, körkörös referenciára, eltérő platformok esetén esetleg több nyelven (JavaScript kliens, Java szerver).
- Transzformáció: az adatok elérhetővé tétele külső rendszerek számára, például email olvasás/írás, Excel import/export stb.

Az adatok forrásnyelvi reprezentációja esetén ez jelentős mennyiségű, általában komplex, és adattípusonként gyakran egyedi forráskódot igényel, amelyet az adatréteg definíciójának változásával folyamatosan szinkronban kell tartani. Az itt esetleg jelentkező problémák adatfüggők, csak speciális esetben jelentkeznek, tesztesetekkel nehezen kizárhatók.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A Dust Platform entitás fogalma és az adatszerkezet Meta leírása közös „nyelvet” teremt az adatleírásra, így lehetőséget ad megbízható, általános eszköztár kialakítására. Egyszer megoldandó feladat az entitások biztonságos mentése és beolvasása adott szöveges formátum (például JSON, XML, ...), illetve külső tároló szolgáltatás (például SQL adatbázis) használatával; ugyanez igaz az export/import feladatra (CSV, Excel, ... kezelő komponensek). Ezek felhasználásával az adattárolás és továbbítás semmilyen egyedi kódolást, későbbi karbantartást, szinkronizációt nem igényel. A több futtató node (szerver-kliens vagy p2p alkalmazás architektúra) használatának támogatása, adatok és üzenetek transzparens proxy továbbítása, cache-elése szintén megvalósítható rendszertől függetlenül kialakított (csak a kernel meta és tools komponenseire támaszkodó) komponensek integrációjával, egyedi kódolás nélkül.

### 4.1.3 DÁTUMOK, ESEMÉNYEK, TÖRTÉNET

A dátumok kezelése számos rendszerben okoz nehézségeket. Nemzetközi környezetben sokféle időzónával, eltérő formátummal kell dolgozni. Gyakran feladat a munkanapok kezelése, átfedések, találkozók, határidők egyeztetése, erőforrások foglalása stb. A dátumok részei a kezelt objektumoknak, az ilyen típusú adatok elérése, értelmezése, egységes megjelenítése sok egyedi, adatszerkezettől függő forráskódot igényel.

A Dust Platform a dátumot nem generikus mezőtípusként, hanem az Esemény entitás részeként kezeli, az entitáshoz kapcsolódó dátum adatok pedig nem az adatszerkezet részei, hanem az entitás Történet modelljében felsorolt, adott altípusú Esemény entitások. Ez valójában nem bonyolítás, csupán arról van szó, hogy a jelenleg mezőnévként megadott azonosító esemény altípusként kerül a rendszerbe. A megközelítés számos előnnyel jár.

- Az alapvető életciklus események (keletkezés, módosítás, megszűnés, ...) kernel kiterjesztés szinten, közösen definiálhatók és kezelhetők.
- Az entitások Története más rendszerek által transzparens módon bővíthető, például a „hozzáférés” tényét, időpontját, környezetét, felhasználót rögzítő esemény hozzáadása tetszőleges „érzékenynek” jelölt adat esetén utólag, konfiguráció módosítással elérhető.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

- A transzparens megjelenés általános szolgáltatásokat (naptár, idővonal, összesítések, keresések stb.) tesz lehetővé a kezelt entitások egyéb adattartalmától teljesen függetlenül.
- A rendszer technológiai és adatrétegből származó eseményei (például üzemzavar, biztonsági riasztás stb.) homogén módon jelennek meg, ami például egy műszaki hiba által érintett adatok körét behatárolhatóvá teszi. Biztonsági rendszerek tanuló algoritmusai baseline meghatározásra használhatják a transzparens eseményhalmazt.
- Mód nyílik globális naptárak (például nemzeti ünnepek, céges naptárak stb.) közös, megosztott kezelésére, és a rájuk épülő szolgáltatások (nyitva tartás, foglalások kezelése) egyszeri létrehozására.

Ismét egy olyan terület, ahol nagyon sok egyedi kód születik, amely a rendszer elvi struktúráját, a róla összegyűjtött tudást elrejt, ráadásul könnyű hibázni (tipikus példa: a Java SimpleDateFormat objektuma nem thread safe). A Dust Platform Esemény fogalmának használatával a forráskód mennyisége a töredékére csökken, ugyanakkor magas szintű szolgáltatások válnak elérhetővé.

### 4.1.4 SZÖVEGEK KEZELÉSE (NYELV, FORMÁZÁS, MEGJELENÍTÉS)

Hagyományos fejlesztési módszertan esetén a forráskódok és szövegek viszonyára nincs semmilyen megkötés. Írhatunk szöveget közvetlenül a kódba, jobb esetben használhatjuk az aktuális környezet általános erőforráskezelő szolgáltatását (Java Resources) a kódból történő szöveg összeállítás során. Felhasználói felületek létrehozásakor a megjelenített fejlécek, címkék, gomb szövegek hasonló eszközökre épülnek; webes felületeknél, jelentések készítésénél számos eszköztár létezik az adatok körül megjelenő „szöveges keret” nyelvi változatainak létrehozására. A rendszer ezen felül gyakran tartalmaz értékkészlet típusú adatokat, amelyek közül választani lehet; ezek megjelenítendő nevét ugyanúgy lokalizálni kell, mint a rendszerbe bevitt adatok egy részét (például szerződések, bemutatkozás, részletes szöveges leírás, help, stb. több nyelven). A szöveges tartalom encoding információját számos helyen kell megadni (forráskód, fordítás, a szöveges állományok beolvasása és írása, vagy web szerver esetén a böngésző felé irányított válasz csatorna és tartalom).

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A Dust Platform belső szöveg encoding egységesen UTF-8; a külső csatornák (stream olvasás és írás, adatbáziskezelő stb.) esetén a kódolásért az adott komponens felel, a platformon belülre nem kerülhet idegen módon kódolt adat. A szöveget a dátumhoz hasonlóan nem generikus adatként, hanem entitásként kezeli, amelynek elsődleges tartalma a karaktersorozat, ehhez mindig tartozik „nyelv” információ.

A szöveg ezen felül egy hierarchikus struktúra, amelyben egy node tartalmazhat további szöveg node-ok listáját (folyó szöveg bekezdései), illetve reprezentálhat magasabb logikai szintet (fejezetek). Szöveggé alakítható az entitások bármely adata, ezek felsorolása révén bármely entitás szöveggé formázható, ezek felhasználásával dokumentumok készíthetők.

A szöveg entitás tartalmazhat formázási információt, amelyet a megjelenítő komponens fog majd kezelni. A formázás mindig a teljes tartalomra vonatkozik, amelyet a belső elem helyi beállításai felülírnak. Ismét egy lépéssel magasabb szinten a szöveget elrendezhetjük egy megadott felületen (layout) paramétereizhető algoritmusok segítségével, ezek hierarchiája a dokumentum.

Egy hierarchikus, adatot is tartalmazó szöveg megjelenítése során a rajzoló a hierarchiának és layoutnak megfelelő módon bejárja a szöveg fát, elkéri a karaktersorozatokat, és azokat a beállításainak megfelelően helyezi el a képernyőn, export állományban stb. Az olvasó művelet a konstansok és adatok esetében figyelembe veszi a kiválasztott nyelvet, és az ennek megfelelő karaktersorozatot adja vissza.

### **4.1.5 ADATMEGJELENÍTÉS (FELHASZNÁLÓI FELÜLET)**

Amennyiben a szöveg megjelenítést kiegészítjük azzal, hogy az adatmezőket az adott típust szerkeszteni képes kontroll (szövegdoboz, lista, rádiógomb, stb.) objektummal kezeljük, teljes interaktív felhasználói felületet kapunk. Mivel a kernel adatelérése és módosítása homogén, csak az elérési utat igényli, az alapvető interakcióhoz nincs is többre szükség. Speciális szerkesztő felületek bevonására (összetett editor, vagy egy platform egyedi képességei) természetesen mód van az entitásban az ezeket leíró modellek segítségével.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

Egy komoly felhasználói felület azonban nem csak adatok megjelenítésére és szerkesztésére alkalmas, hanem segíti is a felhasználót a helyes kitöltésben: figyelmeztet a hibás adatokra, egyes mezőknél változtatja a kiválasztható értékeket, alapértelmezett és nem szerkeszthető segéd adatokat állít elő. Ez a logika azonban nem a felülethez, hanem az entitáshoz kapcsolódik. Ennek megfelelően az entitás típusaihoz adható meg konfiguratív módon vagy forráskód által a kapcsolódó validáló logika, amelynek válaszai egy Állapot modellben tárolhatók. A felhasználói felület az innen kiolvasható információra reagál: hibalistát jelenít meg, mező háttérét módosít, illetve amennyiben az entitás összesített állapota „hibás”, letiltja a mentés / végrehajtás parancsot.

Így tehát egy Dust Platformra épülő rendszer felhasználói felületei a szöveg formázáshoz használt leírással, entitások hálózata formájában felépíthető, ez minden támogatott platformon azonos módon futni képes. A leírás tartalmazhat tetszőleges platformhoz egyedi kiegészítéseket, amely a többi platformon való megjelenést nem befolyásolja.

Az entitás modell listája, annak típusleírása, a típushoz kapcsolt validáló logika azonban a felhasználói felület számára is elérhető – következésképpen egy alapértelmezett szerkeszthető, validáló logikát is kezelő adatlap automatikusan generálható bármilyen entitáshoz. Továbbá, mivel a platform minden parancsa szintén egy-egy entitás, az összes művelethez ugyanígy, automatikusan generálható a parancsot ellenőrzött módon kiadni képes felület. Ehhez csupán a Meta réteg entitásra vonatkozó beállításaira van szükség – amely amúgy is rendelkezésre áll, hiszen a kernel is ennek segítségével kezeli az adatokat.

Összességében ez azt jelenti, hogy amennyiben a Meta és Application fogalmak használatával sikerült felépíteni az alkalmazást, annak teljes adatkezelése és művelethalmazára elérhető egy automatikusan generált felhasználói felületen. A rendszer testre szabása valójában a felhasználói interakció ergonómiájának javítása, ezt a szöveg és felhasználói felület fogalmait használva egyedi felületek továbbra is konfiguratív építésével lehet megtenni.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A tevékenység során semmilyen egyedi forráskód nem készül, a teljes rendszer különféle platformokra portolása nem azonos feladatot megvalósító forráskódok írását és a későbbi változások folyamatos követését, hanem a közösen használt leírás finomhangolását jelenti a platform jellegzetességeinek megfelelően (méret, orientáció, kontroll készlet), esetleg kiegészítve adatok vagy adat csoportok szerkesztésére alkalmas, csak az adott platformon elérhető komponens hozzáadásával (például a telefon lokális kontakt listájának felhasználása).

### 4.1.6 BIZTONSÁG – ADAT ÉS SZOLGÁLTATÁS VÉDELEM

Egy informatikai rendszer által kezelt adat jogosulatlan megszerzésére, meghamisítására, módosítására, tönkretételére számtalan ok elképzelhető, ezért a tervezés és üzemeltetés során biztonsági kérdések is felmerülnek.

Sajnos valóban biztonságos informatikai környezetet létrehozni rendkívül nehéz. A legalsó szinten álló forráskód adatai nem védettek, azokat nyelvi szinten reprezentált objektumokba töltötték, amelyhez a programkód további ellenőrzés nélkül hozzá tud férni. Amennyiben az adatok valamilyen külső tárolóban, például adatbázisban találhatók, a programkód valahol tárolni fogja az eléréshez szükséges belépési információt – ezt a programozó ismerheti, és a segítségével jogosulatlan adatokhoz is hozzáférhet.

A jogosultsági rendszer felépítése általában igen összetett, valamilyen külső rendszer (például LDAP, Active Directory) tárolja; de a szervezetnél telepített különféle alkalmazások rendelkezhetnek saját jogosultsági rendszerrel. Általános tapasztalat, hogy egy szervezet informatikai szakembereinek komoly feladatot jelent az alkalmazottak hozzáférésvédelmének beosztásukkal való szinkronban tartása. A rendszerek közötti adatszinkronizációért, a teljes rendszer adataira támaszkodó állapotjelentésért felelős segédprogramok pedig gyakran operációs rendszer scriptek formájában készülnek el, ezek forrása tartalmaz adminisztratív hozzáférési adatokat. Ez nem csak azért veszélyes, mert a forráskódból kiolvasható, hanem azért is, mert egy ilyen felhasználónév és jelszó megváltoztatása nagyon körülményes, gyakorlatilag a rendszer szinkronjának széthullásával járhat.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A Dust Platform természetesen nem garantálhatja minden biztonsági probléma megoldását, viszont szigorúan központosított felépítésével eszközt ad a jelenleginél megbízhatóbb rendszerek építésére.

Az entitások adatai nem jelennek meg forrásnyelvi szinten; minden egyes adathozzáférés a kernel függvényein keresztül valósul meg. Ez statikus belépési pont, a platform aktuális kontextusa (belépett felhasználó, hívó entitás, hívási lánc, ...) a kernel birtokában van, a hívó ezt befolyásolni nem tudja. Következésképpen minden ilyen híváskor egyértelműen eldönthető, hogy az adott ponton a kért adat kiadható-e a hívónak. Ez természetesen teljesítmény problémát felvethet ugyan, de az adatvédelem ebben a környezetben elsődleges fontosságú, másrészt az ellenőrzés során hívott műveletek eredménye a kontextuson belül gyorsítótárba tehető, stb. Nyilván az elkért adatot a forráskódban már bármilyen módon felhasználhatja a hívó, de legalább az elérés ténye naplózható.

A jogosulatlan hozzáférés a hívás helye alapján is megakadályozható. Például: egy adatbázis eléréséhez létre kell hozni egy Felhasználó entitást, amely tartalmazza a belépési azonosítót és jelszót. Az entitásra hivatkozik az alkalmazás, az adatbázis konnektor entitást ennek nevében szólítja meg. Azonban a Felhasználó entitás hozzáférésvédelmi beállítása (az entitás egyik modellje) a felhasználónév és jelszó olvasását kizárólag az adatbázis konnektor entitás számára engedélyezi. Hiába éri tehát el az entitást az alkalmazás kódjának írója, a példány belső azonosítóján kívül nem lát belőle semmit. Amikor viszont ezt az entitást paraméterként egy parancshoz köti, azt elküldi az adatbázis kezelő entitásnak, az ki tudja olvasni a szükséges adatokat, és végre tudja hajtani az adatbázis műveletet. A különféle rendszerek fejlesztői tehát használhatják az adatbázis adminisztrátorok által létrehozott admin Felhasználó entitasokat, de azok adattartalmát nem ismerik; hozzáférési beállításaik a rendszer működőképességének veszélyeztetése nélkül megváltoztathatók maradnak.

Ugyanez igaz a szolgáltatásokra is: a rendszer komponenseinek egyetlen módon lehet parancsot adni: a kernel üzenetküldő mechanizmusán keresztül. Itt mód nyílik a jogosultság megkerülhetetlen ellenőrzésére, a hívás naplózására.

## MŰKÖDÉS – A MÓDSZER ÉRTÉKELÉSE

A platform a megszokott módon, specifikus entitás hálózat révén kezeli saját jogosultsági rendszerét. Ezt tárolhatja a saját perzisztens tároló megoldásainak valamelyikében, vagy használhat egy vagy több külső forrást a felhasználó azonosítására, illetve a hozzáférésvédelmi beállítások ellenőrzésére, importálására. Ezek szabványos megoldások (Active Directory, Google, Facebook, OpenID, stb.), a hozzáféréshez szükséges kódok csak a külső szolgáltatóhoz kapcsolódnak; a bennük tárolt adatok átvitele konfiguráción keresztül szabályozható.

Összességében megállapítható, hogy a Dust Platform alapelvei lehetővé teszik egy transzparens, tetszőlegesen szigorú és a használó kódoktól függetlenül változtatható, nagyon nehezen megkerülhető hozzáférésvédelmi és naplózó rendszer kialakítását. A felhasználói és jogosultsági konfiguráció a platform eszközeivel is kezelhető, de mód van az azonosítás és jogosultságellenőrzés tevékenységet szabadon konfigurálható speciális konnektoron keresztül külső szolgáltatóra bízni.

### 4.2 FEJLESZTÉS DUST PLATFORM KÖRNYEZETBEN

Eddig feladatok elméleti megoldásáról esett szó, azonban fontos azt is tisztázni, hogyan zajlik egy fejlesztési feladat megoldása a Dust Platform eszközeivel. A következőkben létrehozunk egy adatbázis konnektort és egy olyan komponenst, amely képes tetszőleges entitás szerkezetet adatbázisban tárolni, majd onnan visszaolvasni. Az itt röviden ismertetett konfigurációk és forráskódok részletei és magyarázatuk dolgozat végén található Mellékletben olvasható.

#### 4.2.1 FELADAT: ADATTÁROLÁS ADATBÁZISBAN

A Dust Platform egy perzisztens tárolótól a következőket várja el.

- Legyen képes egy tetszőleges modell szerkezettel rendelkező entitás minden adatának tárolására.
- Rendeljen kizárólagos azonosítót minden újonnan létrehozott entitáshoz, ennek alapján legyen képes előkeresni, illetve módosított adatokkal felülírni a már meglévő entitást.



- Kezelje a hivatkozásokat, egy entitás betöltésekor legyen képes beolvasni az összes hivatkozott entitást is.
- Legyen képes több entitás egy tranzakcióban történő mentésére.
- Legyen képes megadott feltételekkel rendelkező entitások keresésére. (Jelen esetben: az entitásra vonatkozó logikai kifejezésből generáljon olyan SQL WHERE feltételt, amely a megfelelő entitásokat megtalálja.)
- Legyen képes automatikusan követni a modell típusának változásait.

A megoldásból később általános „Tároló” szolgáltatás definíciót kell kiemelni, hogy a kernel szempontjából homogén felületet alkosson más implementációkkal (például lokális fájlrendszer, weben elérhető központi perzisztens tároló).

### 4.2.2 KÖRNYEZET ÉS PROJEKT KONFIGURÁCIÓ

A számítógépen telepített a MySQL szerver, Java 8 JDK, Eclipse Mars, Eclipse Data Tools Platform. Az aktuális workspace tartalmazza a `DustCompact` projekteket fordítható, tesztelhető állapotban.

- Létrehozunk egy új Eclipse projektet, neve `DustCompDBMySQL`.
- Projekt referenciának beállítjuk a `DustCompact` projektet, ezáltal a kernel szolgáltatások, azonosítók, utility függvények elérhetővé válnak. Kapcsolódunk az `ExtLib` projekthez, ez a külső szolgáltatások tárolására szolgáló, nem forduló projekt.
- Kiválasztjuk a használni kívánt adatbázis konnektor szolgáltatást, jelen esetben ez a MySQL 5.1.38 jdbc konnektor. Letöltés után ezt a jar-t elhelyezzük az `ExtLib` projektben, és külső könyvtárként hozzákapcsoljuk a projekthez.
- Létrehozunk egy MySQL adatbázist, az adminisztrátor felhasználót, az Eclipse Data Source plugin-hez kapcsoljuk, ellenőrizzük az elérhetőségét.
- Egy egyszerű JDBC példaprogram felhasználásával készítünk egy teszt projektet, amely segítségével az adatkapcsolat működőképessége ellenőrizhető.

Ezzel készen állunk az adatbázis konnektor komponens létrehozására.

### 4.2.3 ADATBÁZIS KONNEKTOR KOMPONENS

Az adatbázis konnektort komponenszt „be kell mutatni” a környezetnek, ehhez készül a `Database.unit.json` (közös) és a `Jdbc.MySQL51.module.json` (a konkrét implementációra specifikus) leíró. A json formátum a kernel által használt JSON serializer által kezelhető adattárolási forma (jelenleg az egyetlen aktív perzisztens tároló formátum, amelyet a szerver-kliens kommunikáció is használ).

A unitban a `JdbcServer` típus a konfigurációs adatok tárolására szolgál: az adatbázis elérési útja, alapértelmezett felhasználó stb. Ugyanitt található a `JdbcServer` szolgáltatás leíró, amelynek saját függvénye nincs, viszont implementálja a `Stateful`, `TextHandler` és `Block` szolgáltatásokat. Ugyanis, ha jobban megnézzük, egy adatbázis szerver valójában egy `TextHandler`: szöveges parancsokat kap, és azokra valamilyen műveletet hajt végre. Tranzakciókat kell kezelnie, ez valójában a `Block` szolgáltatás: van kezdete és vége. Továbbá `Stateful`: az inicializálása lassú, és a rendszer leállításakor kötelező gondoskodni a nyitott kapcsolatok lezárásáról (a MySQL szerver csatornái egyébként elfogynak).

Létrehozzuk a `DustCompJdbcServer` Java objektumot, amely implementálja a definícióban szereplő szolgáltatásoknak megfelelő interfészeket: `DustUtilsStateful`, `DustUtilsTextHandler`, `DustUtilsBlock`; a függvények implementációja nem releváns.

A modulban létrehozzunk a `LogicAssignment` entitást, amely összeköti a szolgáltatást a java osztállyal, az `AutoInit` tag jelzi, hogy a konnektor entitás létrejöttkor inicializálást is kér.

Ezután átlépünk a `Test01` projektbe, létrehozzuk az `sqlTest` alkalmazást. Ennek `RootEntities` listájára felteszünk egy `JdbcServer` entitást, a kapcsolathoz használható konfigurációs adatokkal.

Az `InitMessages` listára pedig egy `ProcessLine` parancsot, benne az első SQL select utasítással. Érdeemes megfigyelni, hogy az alkalmazás szinten nincs utalás az aktuális implementációra, bármilyen SQL szerverre ugyanígy kellene konfigurálni.

Az alkalmazás indulásakor létrejön a `RootEntities` listán szereplő konnektor, automatikusan kap egy `Init` parancsot. Az entitáshoz kapcsolt `JdbcServer` szolgáltatás a `LogicAssignment` miatt megtalálja a Java osztályt, létrehoz belőle egy példányt és az entitáshoz köti, majd meghívja a `public void dust_utils_stateful_init() throws Exception { ... }` függvényt. A `ProcessLine` parancs a már létrehozott konnektor osztály megfelelő függvényét hívja, így le is fut a megadott `select` parancs.

Ezzel gyakorlatilag létrehoztunk egy általános SQL konzolt (csak a `Jdbc` `ResultSet` szöveggé alakítása hiányzik); bármikor direkt kapcsolatot szeretnénk egy adatbáziskezelővel, ez tökéletesen meg fog felelni a célnak. Viszont most általános adattárolás a feladat, tehát további komponensre van szükség.

### 4.2.4 A VÁLASZTOTT ADATBÁZIS SZERKEZET ISMERTETÉSE

Olyan adatbázis tárolót szeretnénk, amely tetszőleges struktúrájú adatot képes tárolni. Ez elsőre nem tűnik egyszerűnek, hiszen az SQL szerver rögzített táblaszerkezettel dolgozik. Szerencsére a probléma régen ismert, ilyen esetben sokkal kényelmesebb az Entity – Attribute – Value (EAV) modell alkalmazása. Ez hasonló a Dust Platform alapelveéhez: arra épül, hogy egy rekord összetartozó attribútumok halmaza, tehát minden rekord két táblával lefedhető: a „master” tábla egy rekordja tartalmazza az entitás adatait és elsődleges kulcsát, a „detail” sorok pedig külső kulcson ehhez kapcsolnak „attribútum=érték” párokat.

Jelen esetben az adatszerkezet kicsit bonyolultabb, hiszen az entitás és az érték „között” megjelenik a modell, kezelni kell a különféle adattípusokat és a hivatkozásokat is, de azért nem is túl összetett a feladat referencia implementáció szintű megoldása. A későbbiekben megoldandó feladat a mentett típusok definícióinak automatikus tárolása, mert amennyiben változik a típus, a meglevő érték rekordokat automatikusan frissíteni kell.

Természetesen előfordulhat, hogy bizonyos típusok tárolására egyedi táblák használata lesz célszerű (ha a típus struktúrája „végleges”, jellemző a sok mező alapján történő keresés, nagy elemszám stb., például az Esemény típus), ettől függetlenül egyéb modellek maradhatnak EAV szerkezetben.

### 4.2.5 EAV TÁROLÓ KOMPONENS

Az EAV tároló a konnektorhoz hasonlóan kap saját típust és szolgáltatást a unitban, `LogicAssignment` entitást a modulban, és egy megfelelő Java osztályt.

Egy entitás mentése során annak tartalmát be kell járni: modellek, azon belül adatmezők (felsorolás esetén minden tag), illetve a hivatkozott entitások esetén rekurzívan ismételni, ha még nem jártuk be, illetve hivatkozni rá, ha már igen. Ezt a kernel `Scanner` szolgáltatása végzi, minden lépésnél a feldolgozási lánc következő elemének küld `Explorer` parancsokat (amely a `Block` és a `Processor` szolgáltatásokat implementálja). Beolvasáskor megcserélődnek a szerepek: a beolvasó rutin küld `Explorer` parancsokat a kernel `Builder` entitásának, amely ennek alapján felépít egy entitást. (Klónozáshoz megbízható eszköz egy `Scanner -> Builder` páros).

A `DustCompJdbcEavStore` osztály tehát a `DustUtilsExplorer`, `DustUtilsProcessor` interfészeket implementálja. A Dust Platform atomizáló jellege itt jelenik meg igazán: a szolgáltatásnak nincs is egyedi parancsa.

Mentéshez az `Explorer Block` utasításait egyszerűen továbbítja a konnektornak (ettől kerül egy tranzakcióba a művelet), illetve gondoskodik a szintnek megfelelő Entity és Model rekord létrehozásáról, megjegyezve az entitás egyedi azonosítóját. A `Process` parancsok az aktuális kulcsok alatt Value tábla sorokká alakulnak. Ezek a műveletek valójában SQL parancsok, vagyis egyszerű szöveg sorok, amelyeket `TextHandler` parancsok formájában lehet a következő komponensnek elküldeni. Az első kísérletek során nem is az adatbázis konnektort, hanem egy konzol vagy file stream-et érdemes következő elemként konfigurálni, ezzel lehet tesztelni az SQL parancsok összeállítását. Amikor az alapok rendben lévőnek tűnnek, át lehet kötni a konnektorra, hogy a valós környezetben is működik-e; a végeredmény egy sikeresen mentett egymásra hivatkozó entitás halmaz.

A beolvasás teszteléséhez a `Processor` szolgáltatás is használható, a paraméter az imént sikeresen elmentett entitás egyedi azonosítója. Az entitás betöltése ismét egy megfelelően paraméterezett SQL select, amely most vissza fogja adni a cél entitáshoz kapcsolódó összes adatot (az entitás, modell és adat táblák kulcsaikon joinolva). A választ soronként olvasva valójában „bejáró események” keletkeznek, amelyet megfelelő kontextusba helyezve a feldolgozási lánc következő elemének továbbít a komponens. Mivel a `Builder` elég összetett kód, a teszteléshez először a JSON formattert érdemes rákötni, a feldolgozási hibákat sokkal egyszerűbb a hibás JSON alapján megtalálni. Amikor már annak szerkezete helyes, a láncra beköthető a `Builder`.

Ezek a komponensek természetesen sokkal bonyolultabbak, mint például megfelelő annotation-nel egy Java osztályt Hibernate-tel perzisztenssé tenni. Ugyanakkor ezt csak egyszer kell létrehozni, és onnantól kezdve *minden egyedi beavatkozás és forráskód gépelés nélkül bármilyen adat mentésére, megkeresésére alkalmas komponens kaptunk, amely minden Dust Platformra épülő rendszer alatt változtatás nélkül felhasználható*. Ez már megéri a fáradságot.

Megjegyzés: a tároló implementációja is kicsit bonyolultabb lett végül, a beolvasást megkönnyítő view, illetve az írás gyorsításához prepared statementek használata miatt. Amikor viszont a szöveg kezelő komponensek elkészülnek, az SQL utasítás generálás teljes mértékben átkerül majd oda.

### 4.2.6 MELLÉKSZÁL: BINÁRIS ADATTÁROLÓ

A fejlesztés első fázisában még szerepelt a lehetséges adattípusok között a CLOB, stream tárolás támogatására. Az elemzés során azonban kiderült, hogy az ilyen adatok kezelése teljesen eltér a többitől, így kimaradt az adattábla oszlopai közül. Sokkal valószínűbb, hogy a konnektor fölött, az EAV tárolóval párhuzamosan meg fog jelenni egy Stream store komponens is, amely Stream objektumok adatbázisban történő tárolását teszi majd lehetővé, természetesen itt is szolgáltatás szinten analóg módon a más típusú (helyi fájlrendszer, webes távoli elérés, ftp tároló, stb.) stream szolgáltatókkal. Így a stream store konfiguráció változtatásával, minden egyéb beavatkozás nélkül lehet majd váltani közöttük.

### 5 ÖSSZEGZÉS

#### 5.1 SZUBJEKTÍV ÁTTEKINTÉS

Az informatika rendkívül látványos fejlődése, a mérnöki terület (számítási teljesítmény, tárolókapacitás növekedése, méretek és fogyasztás csökkenése), és a globális infrastruktúra (nagy sáv szélességű hálózat, informatikai eszközök és szolgáltatások elérhetősége) szó szerint egy új világot teremtett, teljesen természetes az állandóan velünk lévő eszközről elérhető kommunikáció, navigáció. Ezzel a változással nem tartott lépést az informatikai rendszerek belső, tartalmi fejlődése, jó példa erre a fertőzött okos eszközökről indított sikeres DDoS támadás a Dyn szolgáltató ellen, a távolról lefékezhető „feltört” autó.

Az 1970-es években lényegében kitalálták a mai informatikai környezet minden elemét (Advanced Research Projects Agency - ARPA, Palo Alto Research Center – PARC, ...) az internettől a tabletekig. A fejlesztésben akár a mai gyakorlatot is megelőző technológiákkal dolgoztak, a mérnöki terület azonban lehetetlenné tette az akkor futurisztikus eszközök fizikai megvalósítását, elterjesztését. Azután viszont fordult a kocka: a számítógép kiköltözött a kutatólaborokból először a gyárakba, majd megjelent a személyi számítógép fogalma, mára eljutottunk az okos otthonokig, „önvezető” autókig.

A mérnöki fejlesztéshez óriási erőforrásokra és ilyen szemléletű kutatókra, tervezőkre, és a „népszerű tudományra” nyitott vevőkre volt szükség, a valós problémák absztrakt hátterét kutató tudomány elvesztette a terepét. A felhasználás a tömegtermelő ipar területe, amely mérnöki szemlélettel keres megoldásokat, és konfliktusban áll az „elvont” kutatással, a kutatók pedig nem tudnak hatékonyan dolgozni a szabadalmakkal, gyártási titkokkal körbezárt ipari környezetben. Bár a szoftver adatszerkezet jellege újra a legnagyobb cégek érdeklődési körébe került (?aaS szolgáltatások), a keresés irányát a mérnöki szemlélet határozza meg, a tudományos eredmények (akár a Palo Alto Research Center - PARC lassan fél évszázados koncepciói és működő megoldásai) nem jelennek meg bennük.

## ÖSSZEGZÉS

A Dust Platform szemléletmódját a két terület határán dolgozó kutató-mérnökök inspirálták, például az ENIAC-ot megvalósító Neumann János, az áramkörök és Boole algebra összekapcsolásától az információelmélet megteremtéséhez jutó Claude Shannon, vagy az Apollo vezérlő számítógépének programozását vezető Margaret Hamilton. Az eredmény nem tekinthető sem tudományos, sem ipari szempontból lezártnak – viszont szigorú elméleti alapokra épül, ipari feladatok megoldására képes, így a tervezés, fejlesztés, fenntartás és együttműködés új, sokkal hatékonyabb eszközévé válhat.

### 5.2 JELLEMZÉS OBJEKTÍV SZEMPONTOK SZERINT

Az alábbi lista a tervezhető, megbízható fejlesztés néhány kiragadott szempontja szerint jellemzi a Dust Platformot.

1. **Hatékonyság** – Azonos, forráskódot nem igénylő eszköztár írja le a rendszer által kezelt adatok szerkezetét és a rendszert felépítő komponenseket. A mai rendszerek létrehozása és karbantartása jelentős részben ezekről a kódokról szól, itt fel sem merül ilyen igény.
2. **Rugalmasság** – A rendszer komponensei kizárólag absztrakt szolgáltatás szinten kapcsolódnak egymáshoz; hasonló szolgáltatást nyújtó másik implementációval bármikor kicserélhetők. Ez a fejlesztés során (teszt környezet előállítása) ugyanakkora jelentőségű, mint amikor egy aktív rendszer futtató környezetét vagy kulcselemét (perzisztencia, hozzáférésvédelem) kell megváltoztatni.
3. **Megosztás és újrahasznosítás** – Az egységes leíró nyelv kötelező használata révén minden komponens létrehozása egy környezettől független, transzparens leírással, és a globális kontextusban való elhelyezéssel kezdődik, nincs különbség az aktuális rendszer és a platform által teremtet globális környezet között. Ugyanígy a kernel felépítéséhez használt eszköztár, illetve bárki által bármikor létrehozott szolgáltatás beépíthető a saját rendszerbe; ennek feltételei (más komponensek, futtató környezet) átláthatóak.

## ÖSSZEGZÉS

4. **Rapid development** – A kommunikációs zavarok elkerülésének leghatékonyabb módja működő szolgáltatás bemutatása, ennek gyenge pontja a prototípus fejlesztési költsége és későbbi felhasználhatósága. A Dust Platform eszközeivel az adat és szolgáltatások tervezése azonnal használható rendszert eredményez: az entitások létrehozhatók vagy importálhatók, tárolhatók és kereshetők, automatikus felhasználói felületen karbantarthatók. A kiinduló adatszerkezet folyamatosan változtatható, a mellékhatások pontosan behatárolhatók. Nincs külön prototípus, az induló terv folyamatosan bővül az átadható rendszer szintjére.
5. **Tervek és valóság** – A Dust Platform számára a terv (vagyis a rendszert homogén eszköztárral leíró adatszerkezet) maga a futtatható alkalmazás, nincs „emberi beavatkozás” a két szint között. A dokumentáció máshol a tervező fejében lévő adatszerkezet „exportja” – itt a rendszer entitásaihoz közvetlenül fűzhető leírás, a futó rendszer aktuális struktúráján keresztül bejárható „hipertext”.
6. **Organikus fejlődés** – A kernel kialakítása során igazolt fejlesztési tapasztalat, ahogy egy teljesen forráskód alapú szolgáltatás szerkezete letisztul, a hasonló részfeladatok bezáródnak, majd önálló „utility” szolgáltatássá alakulnak és kiszakadnak a forráskódból. A szokványos fejlesztési módszertanok merevsége „silók” irányába vonzza a fejlesztést és feszültséget kelt tervező és fejlesztő között – a Dust Platform a granularitás felé törekszik és párhuzamos motivációt eredményez.
7. **Biztonság** – A Dust Platform által kezelt adatokhoz és szolgáltatásokhoz csak a platform interfészén keresztül lehet hozzáférni, ennek ellenőrzése kernel szinten történik, megfelelő adatstruktúra által szabályozott módon. Ennek megfelelően a jogosultság ellenőrzése, az elérés naplózása megkerülhetetlen, a rendszer élete során bármikor módosítható, és forráskód „alatti” szinten üzemel.

*A fenti szempontok teljesítése komoly erőfeszítést igényel, ennek általában komoly ára van a szolgáltatásokat nyújtó platform használója oldalán.*



## ÖSSZEGZÉS

8. **Függőség** – A Dust Platform használatához a leíró nyelv alapfogalmait kell elfogadni, minden más ezen a nyelven készül. A kernel implementáció maga is Dust alapú, lecserélhető bármilyen ekvivalens szolgáltatásra.
9. **Méret** – A kernel elsődleges célja a minimális méret és szolgáltatáskészlet; minden más külső modulokban valósul meg, amelyek létrehozása egyszerű feladat. A Dust Platform homogén környezetet ad a beágyazott rendszerektől a globális szolgáltatásokig.
10. **Betanulás** – A Dust Platform egyértelmű vonalat húz az adatszerkezet szinten rögzített, bejárható, dokumentálható „Mit?” és a forráskódon megvalósított „Hogyan?” között; az előbbire egyetlen, hatékony módszertant ad. Egy Dust rendszer „önmaga magyarázata”.
11. **Teljesítmény** – A kernel implementációja az átláthatóságra törekszik, messze nem optimális; viszont a Neumann architektúra tanúsága szerint ez előny, mert később javítható. A végletekig kitekintve: a Sun megkísérelte a JRE hardver implementációját, de annak mérete és fejlődése miatt ez nem volt életképes. A Dust kernel minimális és lezártnak tekinthető szolgáltatáskészlete viszont felveti ugyanezt a kérdést.

A felsorolt szempontok mindegyikéhez pontosan tudható, a több vagy kevesebb a jobb. A mai módszertanok esetén ezek ellentétes irányba hatnak, az optimum egy elfogadható kompromisszum a feltételrendszer tagjai között. A Dust Platform felépítése során a kijelölt cél minden feltétel egyidejű teljesítése volt, amely az évek során kialakította a dolgozatban leírt szerkezetet, a fenti lista pedig nem ideák felsorolása, hanem egy működő rendszer jellemzése.

### 5.3 JELEN ÁLLAPOT

A dolgozatban részletezett példák a DustCompact eszköztárra épülnek, amely az ERPort ETL rendszer fejlesztéséhez létrehozott részleges, működőképese Dust Platform implementáció Java és JavaScript környezetben.

#### 5.3.1 A DOLGOZAT HÁTTERÉT ADÓ MUNKA

A jelenleg használatban lévő informatikai módszertanok elemzése, a belső ellentmondások (mint például az objektum, a kontextus, a komponensek közötti

## ÖSSZEGZÉS

kommunikáció vagy a binding) feltárása, azok hatásának elemzése és alternatív, hatékonyabb megoldások keresése húsz év fejlesztői, rendszertervezői munka háttérében zajlott. Számos részfeladat megoldása és kísérleti változatok után létrehoztam az itt bemutatott DustCompact Java (túlnyomó részben JRE 1.3) implementációt, amely egyebek mellett az alábbi szolgáltatásokat nyújtja:

- Rugalmas, forráskód nélküli adat és szolgáltató komponens kezelő kernel.
- Egy JSON szintaxist használó leíró nyelv, annak író és olvasó komponensei. Ilyen JSON fájlok használatával tetszőleges adat és szolgáltatás tölthető a kernelbe.
- Java szintaxis és API, amely révén Java forráskód integrálható a definiált szolgáltatások végrehajtására.
- Futtató környezet, amely egy „bootloader” kódrész után betölti a kernel működéséhez szükséges modulokat (a kernel maga is a fenti eszközökre épül), majd a saját alkalmazás adatleírását is betölti, elindítja. Ezzel bármilyen teszt alkalmazás létrehozható.
- Segéd szolgáltatások, például adat bejárás, állománykezelés, adatszűrés, ...
- JavaScript middleware, amely példát ad az egyedi kódolás nélküli, mégis teljes körű adatkezelésre egy másodlagos (böngésző) környezetben.
- Külső eszközök integrációja, például Apache Poi (Excel import/export); MVEL (kifejezés kiértékelés); J2EE (Java servlet kiszolgálás); Jetty (servlet container implementáció); JDBC MySQL (adatbázis kezelés).

A mesterképzés tananyaga és a nemzetközi aktuális szakirodalom áttekintése segített a tanulságok és alapelvek összegzésében, az elméleti váz megformálásában, amelyre a dolgozat szerkezete és szövege épül, illetve kijelölte az összehasonlítás szempontrendszerét.

A mellékletben a Hello, World! alkalmazás különféle megvalósításain keresztül bemutatom a fő elveket és a megoldás rugalmasságát; az adatbázis kapcsolatot megvalósító komponens fejlesztésének és forráskódjának elemzésével pedig azt, ahogy az egyszerű eszközöket komplex feladat megoldására, majd az elkészült kódok további finomítására használom.

### 5.3.2 A DUSTCOMPACT IMPLEMENTÁCIÓ JELLEMZŐI

- Ez alapvetően egy Java fejlesztői segédlet, ezért a célnyelven sokkal több szolgáltatás érhető el, mint egy ideális, nyelvfüggetlen Dust implementáció esetén. Ezt a szűk fejlesztési idő és a projekt során változó elképzelések hatékonyabb követése tette szükségessé.
- Egyfajta „nyelvfüggetlenséget” jelent szerveroldalon, hogy a kernel implementációban Java 1.3 szintaxist használtam, bár a „utility” komponenseknél a tömörebb írásmód miatt visszatértem a Java 5-ben bevezetett kényelmesebb szolgáltatások használatára (enum, iterációk). A verziófüggő elemeket (például a különféle collection implementációk) interfészek alá rendeztem és különálló projektben implementáltam.
- A fejlesztés vége felé már érződött néhány kellemetlen átfedés a „Java segédlet” és a Dust elvei között. Ilyen például a process / visitor Java nyelvi és Dust szolgáltatás szintű támogatása, a programozási konstrukciók (feldolgozó stack, switch) Dust szintű támogatásának hiánya, amelyre sajnos nem volt idő.
- Elkészült egy korlátozott JavaScript „middleware” amely egy általános DustCompact szerver – JavaScript kliens architektúra szolgáltatásait tartalmazza (kiegészítve az ERPort speciális igényeinek GUI kerettől független részével). Feladata a szerverrel való adat és parancs kapcsolat, a válaszok frissítése, illetve a Meta adatokból automatikusan generálható felhasználói felülethez szükséges adatok kibontása. Alapvető cél volt, hogy a middleware teljesen elszigetelje egymástól a szerveroldali adatrepresentációt, és azokat az adatokat, amelyek a GUI viselkedéséhez szükségesek. Ez nagyon egyszerűvé tette az adatrepresentáció módosítását (erre volt példa).
- ReactJS általános szolgáltatások: lapozható grid, status report, automatikusan generált adاتمegjelenítés és szerkeszthető adatlap. A kapcsolódó kódokat Roncz László kollégám készítette.

## ÖSSZEGZÉS

### 5.3.3 MINTA ALKALMAZÁS: HELLO WORLD!

A „Hello, World!” a Test01 projekt alatt létrehozott konfiguráció sorozat, feladata az alkalmazás építés alapfogalmainak bemutatása, parancssori program.

Az első változat arra példa, hogy a tipikus Hello, World! Dust Platform környezetében nem igényel programozást – viszont a platform alapfogalmait muszáj megérteni és használni. Az eddigiek alapján természetesen egyszerű a feladat: az `Application initMessages` listájára egy „Hello, World!” szöveget tartalmazó `ProcessLine` parancsot kell felvenni, amelynek címzettje a konzol kimenetet fedő, `TextHandler` szolgáltatást nyújtó entitás.

A második lépés a saját forráskód használatát mutatja be. Az `Application` kiegészül egy lokális `Unit` és `Module` deklarációval, amelyben létrehozzuk a `Test01` típust (ez a `greeting` paramétert tartalmazza, hogy a saját konfiguráció szintű paraméter beállítását és forráskódból történő használatát megmutassa), és szolgáltatást (amely a `Processor` szolgáltatást implementálja). A forráskódban látható a paraméter beolvasása, és a feldolgozási lánc következő eleméhez továbbítása. A futtatáshoz most az `InitMessages` lista egy paraméter nélküli `Process` parancsot küld egy `Test01` entitásnak, ahol a feldolgozási lánc következő eleme a konzol kimenet.

A harmadik változat ugyanezt a megoldást webes környezetbe helyezi át. Az alkalmazás egy `Jetty` server szolgáltatást tartalmaz (ez az `AutoInit` miatt magától indul). A megjelenített lapon a `Test Process` HTML link az általános `api/cmd` dispatcher servletnek küldi a `Test01` entitásnak szóló, paraméter nélküli `Process` parancsot. A kimenet most nem a konzolra, hanem a `Jetty` server servlet `response stream`-jére van állítva, tehát a válasz a böngészőben jelenik meg. A változat érdekessége, hogy a `Test01` kódja egyáltalán nem változott meg, viszont teljesen új környezetbe került. A weblapon egyéb diagnosztikai felületek is szerepelnek, ezek a teszt alkalmazás szempontjából nem relevánsak.

### 5.3.4 ÉLES HASZNÁLAT: ERPORT, EGY WEBES ETL RENDSZER

Az ERPort alkalmazás megrendelője SAP tanácsadó, megvalósítására egy háromfős startup cég tett kísérletet. A rendszer bemutatható szinten működőképes, azonban az erőforrások nem voltak arányban a versenytársak méretével.

Egy informatikai tanácsadó számára jellemző feladat azonos célokra létrehozott, de teljesen eltérő eszközökkel és adatkészlettel megvalósított informatikai rendszerek integrációja, amikor például egy nagyobb cég felvásárol vele azonos piacon, hasonló termék vagy szolgáltatás palettával rendelkező egy vagy több kisebb vállalkozást. A cégek mindegyike rendelkezik a saját adatainak kezelésére alkalmas informatikai rendszerrel. A beolvasztani kívánt cégek rendszereinek szerkezetét és kapcsolati hálózatát fel kell tární, módszertant alkotni a központi struktúrába való migrációra, amelyet végre kell hajtani – közben viszont fenn kell tartani a napi működést. A feladat igen összetett, sok informatikai és kommunikációs problémát kell kezelni, a tanácsadónál hatalmas mennyiségű, folyamatos frissítésre szoruló adathalmaz keletkezik.

A DustCompact környezetre épülő ERPort feladata a következő részekből áll:

- *Állománykezelés:* a tanácsadónak számtalan állományt küldenek. Ezek egy része nem feldolgozandó általában bináris tartalom: képek, leírások, szerződések, stb., amelyeket kereshető, elrendezhető módon tárolni kell. Tudni kell a feltöltés idejét, képesnek kell lenni azonos névvel feltöltött állományok helyes kezelésére (a korábbi állomány nem felülírható, az is fontos információ, hogy mikor milyen adat állt rendelkezésre). Ezeket az állományokat adatelemekhez (termékek, személyek, cégek) kell tudni rendelni n-m kapcsolatban; onnan elérhetővé kell tenni őket.
- *Adatimport:* bizonyos állományok adattáblákat tartalmaznak. Ezeket (Excel munkalapok, csv exportok, stb.) az állományokat feltöltés után feldolgozza, az oszlopok neveit mezőnévként használva adattípusokat hoz létre, majd a további sorokat ilyen típusú entitások formájában tölti be és menti el. Az import felkészült arra, hogy a táblázat nem homogén (fejlécek, csoportosítások, stb.), illetve a címet tartalmazó adatokból képes Google geocoding segítségével térkép koordinátákat visszanyerni.

## ÖSSZEGZÉS

- *Adatszerkesztés*: a betöltött adatokat lapozható táblázatok formájában megjeleníti, a kiválasztott sorhoz adatlapot tud megjeleníteni. Az adatlap szerkeszthető, a módosítás menthető.
- *Transzformáció*: az ERPort tervezésekor meghatározott cél struktúrák (Product, Brand, Employee, stb.) feltölthetők az importált/módosított adatokkal megadott szabályok szerint, itt mód van közvetlen értékmásolásra, kifejezéssel megadott számított érték betöltésére vagy konstans adat megadására. A művelet beállításait tartalmazó entitás természetesen tárolódik, így a tevékenység megismételhető.
- *Export*: tetszőleges típus(ok) tartalma Excel állományba exportálható; a művelet eredménye a szerveren tárolódik, onnan bármikor letölthető.
- *Speciális nézetek*: a fejlesztés során felmerült megjelenítési módszerek (helyek megmutatása zoomolható, mozgatható Google Maps panelen; adatkocka alapú többdimenziós összesítés megjelenítése „foam” panelen, naptár és idővonal kezelés). Ezek az aktuális változatban a felület átalakítása miatt nem elérhetőek.

## 5.4 FEJLESZTÉSI TERVEK

### 5.4.1 DUST PLATFORM

A DustCompact implementáció igen hatékony fejlesztési eszköztár lett, azonban elérte a határait, erős a feszültség a Dust Platform általános, konfiguratív komponensei és a Java szintű implementáció között. A közeljövőben ez a fejlesztési vonal lezárásra kerül a jelenlegi állapot kisebb kiterjesztéseivel (tag kezelés, adatbázis perzisztencia integráció, az ERPort struktúrájának letisztítása és demo szintű véglegesítése), így korlátainak ismeretében fejlesztési alapként használható lesz.

Érdemi továbblépés azonban a jelenlegi, Java nyelvre épülő implementáció átdolgozása egy olyan megközelítésre, amely ezt a függést minimalizálja. Ennek legfontosabb komponenseit a következő fejezetek írják le.

### 5.4.2 PÁRHUZAMOS VÉGREHAJTÁS KEZELÉS

Jelenleg a DustCompact egy szálon fut, a szinkronizációt bizonyos kritikus pontokon (lazy loading tárolók Factory hívásai) Java szinten kezeli. Ez az eszköztár kiegészíthető a szerver oldalon egy aszinkron job management támogatással, ugyanígy a szerver-kliens kapcsolat az aszinkron válasz lehetőségével. A JavaScript middleware már teljesen elválasztja a kliens oldali kódot a szerverrel való kommunikációtól, itt elhelyezhető egy olyan háttér kommunikáció, amely kezeli a szerver „folyamatban” válaszát, időnként újra kérdez, ennek alapján egy folyamat státusz megjelenítésre képes (akár több háttér művelet egyidejű kezelésével); az eredeti callback függvényt akkor hívja vissza, amikor a szervertől a művelet eredménye megérkezik. Ez természetesen kiegészíthető a művelet megszakításának képességével, amely szerver és kliens oldalon egyaránt megfelelően kezelhető.

Mindez azonban komoly „felhasználó oldali kód” megírását követeli, amelyben könnyű hibázni. Jelenleg elemzés és kidolgozás alatt áll az a módszertan, amely a Dust Platform kernel rétegében megbízhatóan, a platformok közötti esetleges különbségeket elfedő módon, kizárólag az entitás, üzenetkezelés, kontextus fogalmainak használatával képes leírni a párhuzamos működés szabályait. Ennek implementációja már túlmutat a DustCompact keretein.

### 5.4.3 KONFIGURATÍV KIFEJEZÉSEK, ALGORITMUSOK

A kernel, toolkit és ERPort komponensek megvalósítása már számos alkalommal ütközött a konfiguratív algoritmus kezelés hiányába. Jellemző példa a parserek működése: a beolvasandó alapvetően hierarchikus, rekurzív feldolgozást igénylő folyamat aktuális állapotát egy feldolgozó stack reprezentálja. Ez az ára annak, hogy a vezérlés nem a parser, hanem az eseményforrás oldalán van (például stream olvasás) – a nyereség a memóriaigény csökkenés, illetve az olvasó komponenstől való függetlenség. Ez a nyelvi konstrukció számos helyen jelenik meg a kódban, amely arra utal, hogy szerencsésebb lenne entitásokkal lefedni, ez hosszabb távon ismét a rendszer átláthatóságát, hordozhatóságát növelné.

## ÖSSZEGZÉS

Másik ilyen jelenség a plugin-jellegű feldolgozás, amikor valamilyen feltétel alapján dől el az, hogy az aktuális elemtől a feldolgozást milyen komponens végzi. A parserek itt is megjelennek, de ilyen például a web szerver kérés elosztó komponense. Itt egy „Switch” konstrukció lenne szükséges.

A kifejezések tárolására, kiértékelésére szintén már szükség van, a DustCompact esetén a megoldást az MVEL engine integrációja jelentette (ez template és script engine szerepben is használható), így működik a számított mezők kezelése az ERPort import során. A megoldás hátránya viszont, hogy például alkalmatlan dinamikusan építhető logikai kifejezések struktúrájának visszafejtésére, amelyre az SQL where feltétel generálásnál szükség lenne. A következő lépésben elsődleges fontosságú egy saját, entitás alapú kifejezéskezelő komponens létrehozása, akár még a DustCompact környezetben is.

### 5.4.4 FELHASZNÁLÓI FELÜLET

A szövegkezelés teljes eszköztárának referencia szintű implementációja szükséges, a következők szerint.

- Az encoding probléma teljes kizárása (ezt a feladatot az entitás tárolók kötelesek intézni, az ő feladatuk a szövegek tároló szintű helyes kódolása, illetve a kernel beállított kódolása közötti konverzió).
- A szöveg entitások nyelv beállításának, és az aktuális kontextus nyelvének kezelése, a megfelelő változat elérése.
- A szöveg hierarchikus szerkezetének kezelése, konstansok és adatok szöveggé alakítása, formázási beállítások lehetővé tétele, layout (egyszerű eszközökkel).
- Az interaktív felhasználói felület alapl működéseit fedő fogalmak és szolgáltatások kialakítása, referencia implementáció egy eszköztárral (például Java Swing); a hordozhatóság ellenőrzése miatt összevetni egy alternatív környezettel (SWT, GWT).



### 5.4.5 A JAVA KÖRNYEZET ELHAGYÁSA

A Dust Platform eddig felsorolt eszközei is elég erősek: homogén tudásreprezentáció, bármikor cserélhető alapkomponeensek, az adatstruktúra elszakad a nyelvtől, maguk az adatok a forrás rendszerektől: az adatszótár és komponensek összeállítása után a rendszer szolgáltatásai azonnal működőképesek. Mindez azonban akkor válik igazán érdekessé, amikor az elkészült rendszer platformok között is hordozhatóvá válik; demonstrálható a portolás nélküli analóg, azonnal frissülő viselkedés különféle eszközökön. Ehhez alapvető szükséglet a kernel további vékonyítása, hogy maga is konfiguratív algoritmusokra épüljön (amennyire csak lehetséges). Ennek nyilván komoly ára lesz a teljesítményben, de a veszteség inkább átmeneti: forráskód, bytecode generálható az algoritmusok entitás alapú leírásából, ami akár egy fordítóprogram bemenete is lehet.

Első cél a „vékony felületeken” való megjelenés, tehát mobil platformok (Android, iPhone, WinPhone) támogatása, mert itt a kliens szolgáltatások nem elsődlegesek: a kernel implementáció, algoritmus, szövegkezelő és GUI fedő modul segítségével minden Dust Platform alapú alkalmazás vékony (de a kifejezés és algoritmus modulokkal akár teljes, vastag) kliense megvalósítható.

A következő lépés a konnektor modulok készletének és tudásának bővítése:

- Vállalati környezet: adatbázis, LDAP, beléptető és jogosultság kezelés, üzenet csatornák;
- Integráció: adat és eseménykezelő rendszerek (mail, Sykpe, Slack, Trello, ...)

Ezzel párhuzamosan, a vékony kliensekből kiindulva, az aktuális környezeten elérhető eszköztárak modulokba csomagolásával a nagyobb platformok „desktop” eszköztárainak fejlesztése (natív .NET, iOS, Linux alkalmazások létrehozásának támogatása).

## 6 MELLÉKLETEK

### 6.1 DUSTCOMPACT PROJEKT SZERKEZET

<p><b>DustComp</b></p> <p>A DustCompact környezet közös, megosztott projektje, ezt minden használó projekt include-olja. A működéshez elég a com.dustcomp csomag négy osztálya, az benne levő további csomagok speciális kiegészítőket tartalmaznak.</p> <ul style="list-style-type: none"> <li>• Api: a kernel által használt unitok azonosítóinak nevei, illetve gyakran használt utasítássorok helper függvényekbe csomagolva.</li> <li>• Coll: gyűjtemény osztályok és segédletek. A Platform terveivel ellentétben itt Java nyelvi támogatást kaptak.</li> <li>• Shared: speciális, de gyakran használt unitok azonosítói és helper függvények</li> <li>• Utils: bonyolultabb Java szolgáltatásokhoz készített helper függvények</li> </ul> <p>A Platform célkitűzéseivel szemben ez a réteg “vastag” lett, a következő implementációban cél a kiegészítő csomagok konfiguratív eszközökre és generált forrásokra cserélése.</p>	 <pre> DustComp [erport design t2] ├── src │   ├── com.dustcomp │   │   ├── api │   │   │   ├── DustCompApiHacks.java │   │   │   ├── DustCompApiNames.java │   │   │   ├── DustCompApiRuntimeException.java │   │   │   └── DustCompApiUtils.java │   │   ├── coll │   │   │   ├── DustCompCollectionReader.java │   │   │   ├── DustCompCollections.java │   │   │   ├── DustCompCollectionsUtils.java │   │   │   ├── DustCompUtilsLazyCreator.java │   │   │   ├── DustCompUtilsPool.java │   │   │   ├── DustCompUtilsStack.java │   │   │   └── DustCompUtilsTagger.java │   │   ├── shared │   │   │   ├── DustCompGeocoderNames.java │   │   │   ├── DustCompGuiNames.java │   │   │   ├── DustCompSerializeNames.java │   │   │   ├── DustCompStreamNames.java │   │   │   ├── DustCompStreamUtils.java │   │   │   └── DustCompUtilsNames.java │   │   ├── utils │   │   │   ├── DustCompUtilsDateFormatter.java │   │   │   ├── DustCompUtilsDumpExplorer.java │   │   │   ├── DustCompUtilsSerializer.java │   │   │   ├── DustCompUtilsStream.java │   │   │   ├── DustCompUtilsTimestampProvider.java │   │   │   ├── DustCompComponents.java │   │   │   ├── DustCompNames.java │   │   │   ├── DustCompServices.java │   │   │   └── DustCompUtils.java │   │   └── poc │   │       ├── com.dustcomp.poc │   │       │   ├── DustCompPocUtilsConfig.java │   │       │   └── placholder.txt │   │       ├── JRE System Library [jre1.8.0_111] │   │       ├── deployment │   │       └── info │   │           ├── gui.txt │   │           ├── ideas.txt │   │           └── persistence.txt </pre>
---	---

## MELLÉKLETEK

### DustCompKernelBase

A kernel implementációja: azonosítók kezelése, felhő, entitás, modell, adatok, hívás és kontextus, a statikus réteg inicializálása.

Tartalmaz helper típusokat is, amelyek közvetlen adathozzáférést igényelnek (filter, scanner, builder, stb.) A binary, data és meta csomagok a rétegek elválasztása céljából készültek, de valódi határt nem jelentenek.

Hosszabb távon a kernel szegmenseit külön unitba, akár külön modulokba is érdemes lesz majd rendezni, de a jelen implementációnak ez nem volt célja.



























Érdekesség, hogy a fő projektek Java 1.3 szintaxissal készültek, így régebbi platformokon is képesek futni. A verzió szempontjából kritikus elemek (gyűjtemény osztályok, stream kezelés, ...) DustComp szinten interfészekkel vannak lefedve, és külön projektben kerültek megvalósításra.

```

v > DustCompKernelBase [erport design 12]
v src
  v com.dustcomp.dust.kernel
    v binary
      > DustCompBinaryLogicManager.java
      > DustCompBinaryModule.java
    v data
      > DustCompDataBuilder.java
      > DustCompDataBuilderBase.java
      > DustCompDataBuilderRelay.java
      > DustCompDataBuilderRow.java
      > DustCompDataBuilderTuple.java
      > DustCompDataCloud.java
      > DustCompDataEntity.java
      > DustCompDataFilter2.java
      > DustCompDataModel.java
      > DustCompDataPathResolver.java
      > DustCompDataScanner.java
      > DustCompDataUtilClusterLoc.java
      > DustCompDataUtilCubeBuilder.java
      > DustCompDataUtilDrop.java
      > DustCompDataUtilFilter.java
      > DustCompDataUtilFinder.java
      > DustCompDataUtilTagTreeBuilder.java
      > DustCompKernelInitializerBase.java
      > DustCompKernelInitializerDefault.java
    v meta
      > DustCompMetaService.java
      > DustCompMetaType.java
      > DustCompKernel.java
      > DustCompKernelCallDispatcher.java
      > DustCompKernelCallManager.java
      > DustCompKernelComponents.java
      > DustCompKernelIdManager.java
      > DustCompKernelMain.java
      > DustCompKernelNames.java
      > DustCompKernelUtils.java
  v test
    v com.dustcomp.dust.kernel
      > DustCompKernelTest.java
      > DustCompKernelTest2.java
      > DustCompKernelTestComponents.java
      > DustCompKernelTestEcho.java
  > JRE System Library [jre1.8.0_111]

```

## MELLÉKLETEK

<p><b>DustCompKernelJava7</b></p> <p>A Java verziótól függő kernel osztályok Java 7 implementációja.</p>	<ul style="list-style-type: none"> <li>▼  &gt; DustCompKernelJava_7 [erport design t2]</li> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.stdjava7 <ul style="list-style-type: none"> <li>&gt;  DustCompJava7CollFactory.java</li> <li>&gt;  DustCompJava7StreamProvider.java</li> <li>&gt;  DustCompJava7StreamUtils.java</li> </ul> </li> <li>&gt;  JRE System Library [jre1.8.0_111]</li> </ul> </li> </ul>
<p><b>DustCompUtils</b></p> <p>Általános “helper” szolgáltatások implementációi, például az Application kódja, amely az InitMessages lista elemeit meghívja, itt található. (Innentől minden Java kód a Unit – Module – LogicAssignment beállítások révén épül a rendszerbe)</p>	<ul style="list-style-type: none"> <li>▼  &gt; DustCompUtils [erport design t2]</li> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.dust.utils <ul style="list-style-type: none"> <li>&gt;  DustCompApplication.java</li> <li>&gt;  DustCompEntityBuilder.java</li> <li>&gt;  DustCompFormatCsvParserSimple.java</li> <li>&gt;  DustCompFormatJsonComponents.java</li> <li>&gt;  DustCompFormatJsonParser.java</li> <li>&gt;  DustCompFormatJsonWriter.java</li> <li>&gt;  DustCompStreamString.java</li> </ul> </li> <li>&gt;  JRE System Library [J2SE-1.5]</li> <li>&gt;  MetaEdit.unit.json</li> <li>&gt;  test2.json</li> </ul> </li> </ul>
<p><b>DustCompMvelWrapper</b></p> <p>Kifejezés kiértékelő modul, az MVEL engine integrációja, jelenleg az Expression üzemel, az MVEL egyébként script és template kezelésre is alkalmas.</p>	<ul style="list-style-type: none"> <li>▼  &gt; DustCompMvelWrapper [erport design t2]</li> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.dust.mvel <ul style="list-style-type: none"> <li>&gt;  DustCompMvelExpression.java</li> <li>&gt;  DustCompMvelNames.java</li> </ul> </li> <li>&gt;  JRE System Library [jre1.8.0_111]</li> <li>&gt;  Referenced Libraries</li> </ul> </li> </ul>

## MELLÉKLETEK

<p><b>DustCompServlet</b></p> <p>A J2EE servlet szolgáltatások integrációja. Ez a komponens képes egy J2EE “burkot” építeni bármely Dust Platform alkalmazás fölé, a webről érkező kéréseket üzenetké alakítja, a válaszokat megfelelően formázza.</p> <p>Minden servlet közös őssztályból származik, amely a http parancsokat, logolást, paraméterek olvasását kezeli, a leszármazott választ, esetleges hibaüzenetét a hívónak továbbítja (JSON parser és formatter segítségével). A mögöttes objektumok többsége már csak entitásokat lát.</p>	<ul style="list-style-type: none"> <li>▼  DustCompServlet [erport design 12] <ul style="list-style-type: none"> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.dust.servlet <ul style="list-style-type: none"> <li>&gt;  DustCompServletBaseContextListener.java</li> <li>&gt;  DustCompServletBaseServlet.java</li> <li>&gt;  DustCompServletCommandSimple.java</li> <li>&gt;  DustCompServletComponents.java</li> <li>&gt;  DustCompServletEntityGetter.java</li> <li>&gt;  DustCompServletEntityUploaderSimple.java</li> <li>&gt;  DustCompServletFileDownloadSimple.java</li> <li>&gt;  DustCompServletFileUploadSimple.java</li> <li>&gt;  DustCompServletLauncher.java</li> <li>&gt;  DustCompServletNames.java</li> <li>&gt;  DustCompServletUtils.java</li> </ul> </li> <li>▼  test <ul style="list-style-type: none"> <li>&gt;  TestServlet.java</li> </ul> </li> </ul> </li> <li>&gt;  JRE System Library [JavaSE-1.8]</li> <li>&gt;  Referenced Libraries</li> <li>&gt;  build</li> <li>&gt;  WebContent</li> </ul> </li> </ul>
<p><b>DustCompJettyWrapper</b></p> <p>A könnyebb integráció kedvéért a Jetty alkalmazáserverver kapott egy modult, így külön telepítés nélkül tudunk webes szolgáltatást nyújtani.</p>	<ul style="list-style-type: none"> <li>▼  DustCompJettyWrapper [erport design 12] <ul style="list-style-type: none"> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.dust.servlet.jetty <ul style="list-style-type: none"> <li>&gt;  DustCompJettyComponents.java</li> <li>&gt;  DustCompJettyServer.java</li> </ul> </li> </ul> </li> <li>&gt;  JRE System Library [jre1.8.0_111]</li> <li>&gt;  Referenced Libraries</li> </ul> </li> </ul>
<p><b>DustCompPoiWrapper</b></p> <p>Az Office állományok kezelését végző Apache csomag modulja. Jelenleg az Excel olvasás (általános, illetve egy ERPort specifikus változat), üres és template alapú export támogatott.</p>	<ul style="list-style-type: none"> <li>▼  DustCompPoiWrapper [erport design 12] <ul style="list-style-type: none"> <li>▼  src <ul style="list-style-type: none"> <li>▼  com.dustcomp.dust.io.poi <ul style="list-style-type: none"> <li>&gt;  DustCompPoiComponents.java</li> <li>&gt;  DustCompPoiExport.java</li> <li>&gt;  DustCompPoiExportTemplate.java</li> <li>&gt;  DustCompPoiImport.java</li> <li>&gt;  DustCompPoiImportERPort.java</li> <li>&gt;  DustCompPoiImportSax.java</li> <li>&gt;  DustCompPoiNames.java</li> <li>&gt;  DustCompPoiUtils.java</li> </ul> </li> </ul> </li> <li>&gt;  JRE System Library [jre1.8.0_111]</li> <li>&gt;  Referenced Libraries</li> </ul> </li> </ul>

## MELLÉKLETEK

<p><b>DustCompDBMySQL</b></p> <p>MySQL JDBC elérést lehetővé tevő modul, egyelőre demonstrációs célokból, fejlesztés alatt áll.</p>	<ul style="list-style-type: none"> <li>▼ DustCompDBMySQL [erport design 12] <ul style="list-style-type: none"> <li>▼ src <ul style="list-style-type: none"> <li>▼ com.dustcomp.dust.db.mysql51 <ul style="list-style-type: none"> <li>&gt; DustCompJdbcComponents.java</li> <li>&gt; DustCompJdbcEavStore.java</li> <li>&gt; DustCompJdbcEntityNames.java</li> <li>&gt; DustCompJdbcNames.java</li> <li>&gt; DustCompJdbcServer.java</li> <li>&gt; DustCompJdbcUtils.java</li> </ul> </li> <li>&gt; JRE System Library [jre1.8.0_111]</li> <li>&gt; Referenced Libraries <ul style="list-style-type: none"> <li>tablescripts.sql</li> <li>tablescripts2.sql</li> <li>todo.txt</li> <li>utils.sql</li> </ul> </li> </ul> </li> </ul> </li> </ul>
<p><b>DustCompGeocoding</b></p> <p>A Google Geocoding API támogatást integráló “proof of concept” szintű modul. Jelenleg az API felhasználási feltételeinek változása miatt korlátozottan üzemképes.</p>	<ul style="list-style-type: none"> <li>▼ DustCompGeocoding [erport design 12] <ul style="list-style-type: none"> <li>▼ src <ul style="list-style-type: none"> <li>▼ com.dustcomp.dust.geocoding <ul style="list-style-type: none"> <li>&gt; DustGeocoder.java</li> <li>&gt; DustGeocoderComponents.java</li> <li>&gt; DustGeocoderNames.java</li> </ul> </li> <li>&gt; JRE System Library [JavaSE-1.8]</li> <li>&gt; Referenced Libraries</li> </ul> </li> </ul> </li> </ul>
<p><b>Test01</b></p> <p>Teszt alkalmazások létrehozására szolgáló környezet. A projekthez minden korábbi modul projektje csatolva van, így bármilyen modul gyorsan tesztelhető; a konfigurációk *.json fájlok formájában láthatók a főkönyvtárban. A launch config beállításai között a - DustAppConfig=&lt;fájlnév&gt; beállítással lehet a megfelelőt kiválasztani, ennek hiányában az appConfig.json indul.</p>	<ul style="list-style-type: none"> <li>▼ Test01 [erport design 12] <ul style="list-style-type: none"> <li>▼ src <ul style="list-style-type: none"> <li>▼ test <ul style="list-style-type: none"> <li>&gt; Test01.java</li> <li>&gt; TestSlack.java</li> </ul> </li> <li>&gt; JRE System Library [jre1.8.0_111]</li> <li>&gt; Referenced Libraries</li> <li>&gt; customData</li> <li>&gt; upload</li> <li>&gt; uploadedData</li> <li>&gt; webroot <ul style="list-style-type: none"> <li>appConfig2.json</li> <li>DustCompTest01.launch</li> <li>DustGeocoderTest.launch</li> <li>DustServlet-Zsolt.launch</li> <li>geocoderTest.json</li> <li>helloWorld.json</li> <li>helloWorld2.json</li> <li>helloWorld3.json</li> <li>helloWorld4.json</li> <li>jetty.xml</li> <li>sqlTest.json</li> <li>test.json</li> <li>test.txt</li> <li>testOut.json</li> <li>testPoi.json</li> </ul> </li> </ul> </li> </ul> </li> </ul>

## 6.2 HELLO, WORLD!

### 6.2.1 KIZÁRÓLAG KONFIGURÁCIÓ

A szokásos Hello, World! alkalmazás a Dust Platformban. Ez a szoftver csak adatszerkezet: látható, hogy az alkalmazás hogyan épül fel a kernel komponenseiből. Az entitás a külső objektum `typeId`, `id`, és `Models` mezőkkel; a `Models`-en belüli objektumok az entitás modelljei, a mező név: érték párokat a `Values` objektum tartalmazza.

*helloWorld.json*

```
{
  "typeId": "Dust:Utils:Application",
  "id": "HelloWorld",
  "Models": [
    {
      "typeId": "Dust:Utils:Identified",
      "Values": {
        "id": "HelloWorld"
      }
    }, {
      "typeId": "Dust:Utils:Application",
      "Values": {
        "InitMessages": [
          {
            "typeId": "Dust:Utils:Message",
            "Models": [
              {
                "typeId": "Dust:Utils:Message",
                "Values": {
                  "command": "Dust:Utils:TextHandler!ProcessLine",
                  "Target": {
                    "typeId": "Dust:Stream:ProviderSystem"
                  }
                }
              }, {
                "typeId": "Dust:Utils:Variant",
                "Values": {
                  "Text": "Hello, world from Dust!"
                }
              }
            ]
          }
        ]
      }
    }
  ]
}
```

## 6.2.2 SAJÁT FORRÁSKÓD

A kód egy saját fejlesztésű `Processor` szolgáltatás, ezért implementálja a `DustUtilsProcessor` interfészt, a kód a `dust_utils_processor_process()` függvénybe kerül: beolvassuk a „greeting” paramétert, rákapcsolódunk a `TextHandler` szolgáltatást nyújtó kimenő csatornára, és elküldjük a `ProcessLine` parancsot. A Dust Platform érdekessége az, hogy minden kernel komponens is ugyanilyen építőkocka formájában kerül megvalósításra, példaként az adatbázis kapcsolat komponens látható a 6.4.5 fejezetben.

### *Test01.java*

```
public class Test01 implements DustCompApiNames, DustUtilsProcessor {
    DustIdentifier FLD_GREETING =
        DustCompUtils.getPathBuilder().simpleId("greeting");

    @Override
    public void dust_utils_processor_process() throws Exception {
        Object greeting;

        greeting = DustCompApiUtils.getValue(FLD_GREETING);

        DustCompApiUtils.initRelay(TEXTHANDLER_CMDS_ALL);
        DustCompApiUtils.setMsgVariant(CMD_PROCESSLINE, VT_TEXT, greeting,
true);
    }
}
```

---

```
<terminated> DustCompTest01 [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (2016. okt. 26. 12:15:33)
Hello World 2
DustCompact application with custom business logic implemented in Java
Hello, World! (from greeting parameter)
```

Ábra 6 Hello, World! (2) kimenet – a szöveget saját kód állítja elő



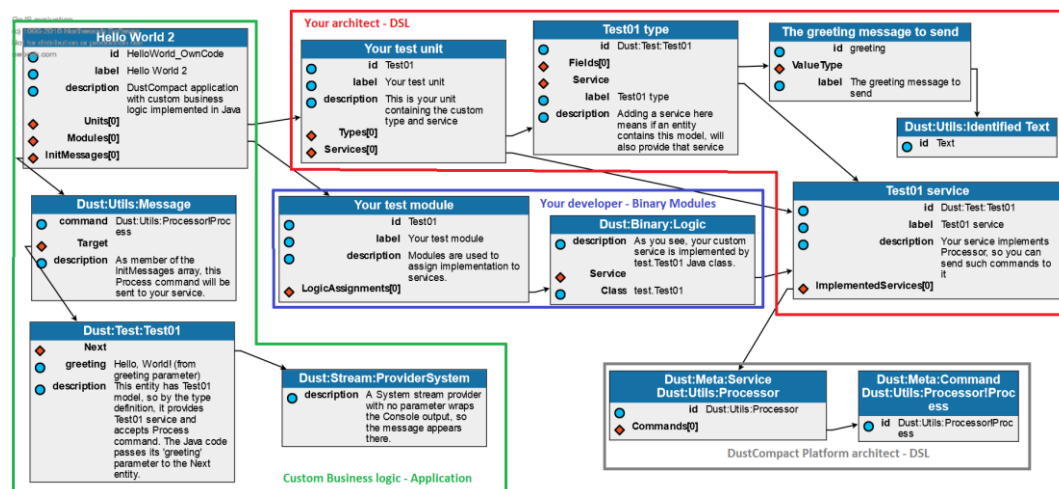
### 6.2.3 INTEGRÁCIÓ

Pontosan hogyan kapcsolódik a saját forráskód az alkalmazáshoz?

Az Alkalmazás építés fejezetben található az Application kiegészítésének leírása. Az előző tesztben az `InitMessages` listán szereplő `Message` entitás tartalmazott mindent, hiszen csak rendszer komponenseket használtunk. Most viszont el kell készíteni egy `Unit`-ot, amely bemutatja a kernelnek a `Test01` típust – ebben bevezetjük a „greeting” mezőt, illetve a `Test01` szolgáltatást hozzá kötjük (így amikor `Test01` elsődleges típusú entitást hozunk létre, ahhoz a kernel a `Test01` szolgáltatást automatikusan hozzárendeli. A `Test01` szolgáltatás definíciójában megadjuk, hogy implementálja a `Processor` szolgáltatást.

Ezzel az alkalmazásuk deklarációját el is tudjuk készíteni, azonban még nem tud futni. Ehhez szükség van egy `Modul`-ra, amely a `Test01` szolgáltatáshoz a saját Java osztályunkat hozzáköti. Amikor a `Test01` entitás kap egy `Process` parancsot, az első hívásnál a kernel ennek alapján létrehoz egy `Test01` objektumot, hozzáköti az entitáshoz, és meghívja a `process` függvényt.

A `Unit` és `Modul` entitást felvesszük az `Application Units` illetve `Modules` listájára, és kész vagyunk. Mindez tehát entitások kezelésével, tehát adatszerkezet műveletekkel történt. Az aktuális állapot megtekintéséhez a `Dust Platform` rendelkezik egy egyszerű webes gráf rajzoló komponenssel, amely az új szerkezetet áttekinthetőbb formában képes megjeleníteni.



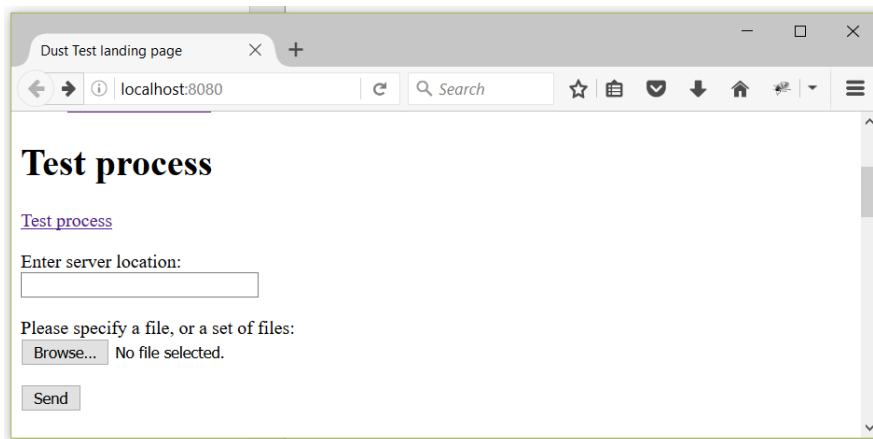
Ábra 7 Hello, World! (2) - entitás szerkezet grafikus exportja, feliratozva

### 6.2.4 MOZGATÁS WEB SZOLGÁLTATÁS ALÁ

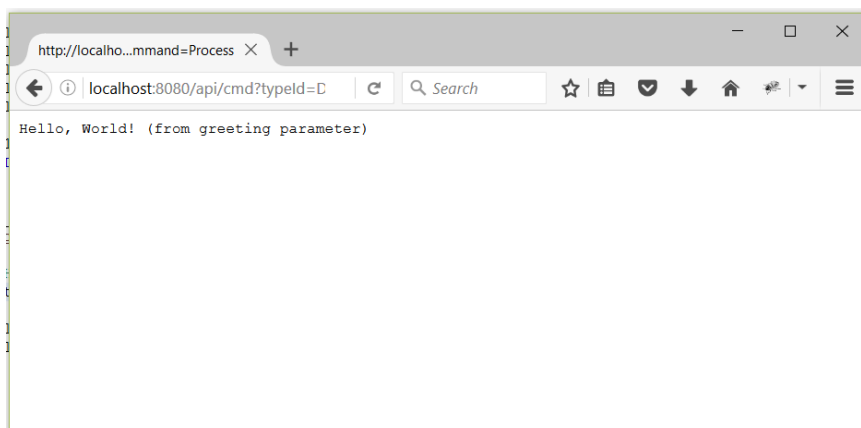
A Dust Platform szigorú modularitása lehetővé teszi alapvető döntések utólagos megváltoztatását. A harmadik változatban az előző szolgáltatás eredményét nem konzolon, hanem webes felületen szeretnénk megmutatni. Ehhez természetesen csak az alkalmazást leíró entitást kell kiegészíteni, illetve létrehozni az `index.html`-t, amelynek ide kapcsolódó része:

```
<h1>Test process</h1>
<a
  href="api/cmd?typeId=Dust:Test:Test01&id=Test01&Service=Dust:Utils:Processor&command=Process">Test process</a>
```

A Servlet modul által biztosított `api/cmd` szolgáltatás paraméterei a cél entitás azonosítása (típus és egyedi azonosító), illetve a paraméterek nélküli parancs. Paraméterezett parancsok esetén ugyanerre a címre POST-tal kell küldeni az üzenet entitást a szokott JSON formátumban.



Ábra 8 Hello, World! (3) - a webes szolgáltatás a Test process linkkel



Ábra 9 Hello, World! (3) - a linkre kattintva megérkezik a greeting szöveg

## MELLÉKLETEK

A konfiguráció a HelloWorld2.json másolata, itt a változásokat mutatom be. Az alkalmazás entitásai közé bekerült a felkonfigurált Servlet:Server (amelyen jelen beállítások szerint a Jetty modul implementál).

```
"Data": [
  {
    "typeId": "Dust:Servlet:Server",
    "id": "appServer",
    "Models": [
      {
        "typeId": "Dust:Servlet:Server",
        "Values": {
          "portPublic": 8080,
          "Handlers": [
            {
              "typeId": "Dust:Servlet:ContextResource",
              "Models": [
                {
                  "typeId": "Dust:Servlet:ContextResource",
                  "Values": {
                    "path": "webroot",
                    "welcomeFiles": [
                      "index.html"
                    ]
                  }
                }
              ]
            }
          ]
        }
      }
    ]
  },
  ],
```

A Test01 entitás (a mi Hello World komponensünk) kimenete, vagyis a feldolgozó csatorna következő eleme a Servlet StreamHandler entitása (amely természetesen TextHandler) – ezért kerül most a böngésző válaszába az üzenetünk, bár a kódon semmit sem változtattunk.

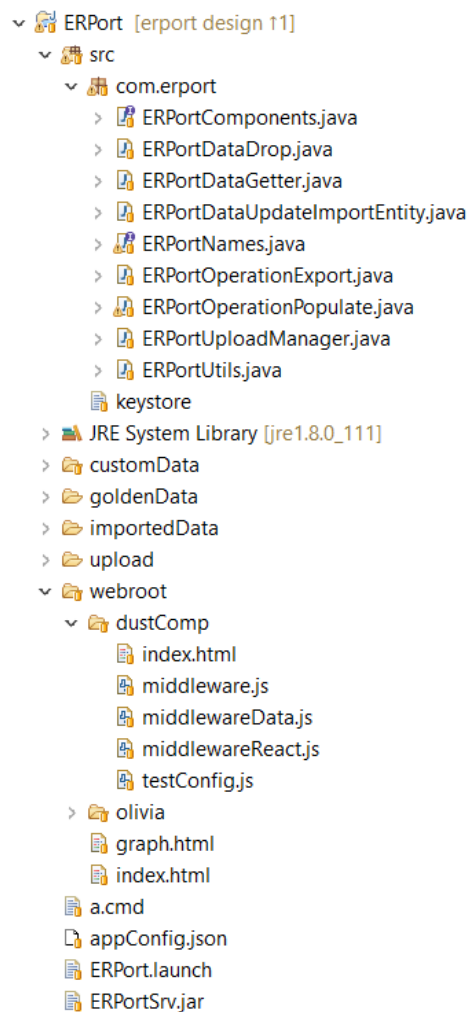
```
{
  "typeId": "Dust:Test:Test01",
  "id": "Test01",
  "Models": [
    {
      "typeId": "Dust:Test:Test01",
      "Values": {
        "greeting": "Hello, World! (from greeting parameter)"
      }
    },
    {
      "typeId": "Dust:Utils:Chain",
      "Values": {
        "Next": {
          "typeId": "Dust:Stream:Handler",
          "id": "Servlet"
        }
      }
    }
  ]
}
```

## 6.3 ERPORT

### 6.3.1 PROJEKT

Az alkalmazás Java projektje az alábbi ábrán látható 9 forráskódból áll, ezek valósítják meg az ERPort specifikus működéseket, minden egyébért a megfelelően konfigurált DustCompact elemek felelősek. Ezen túl az Excel importáló osztályt a Poi projektbe kellett „delegálni”, mert nem volt idő az összes kívánt képességet fedő konfigurálható elem kialakítására, ezért ERPort specifikus, Poi függő átmeneti forráskód született a feladat megoldására.

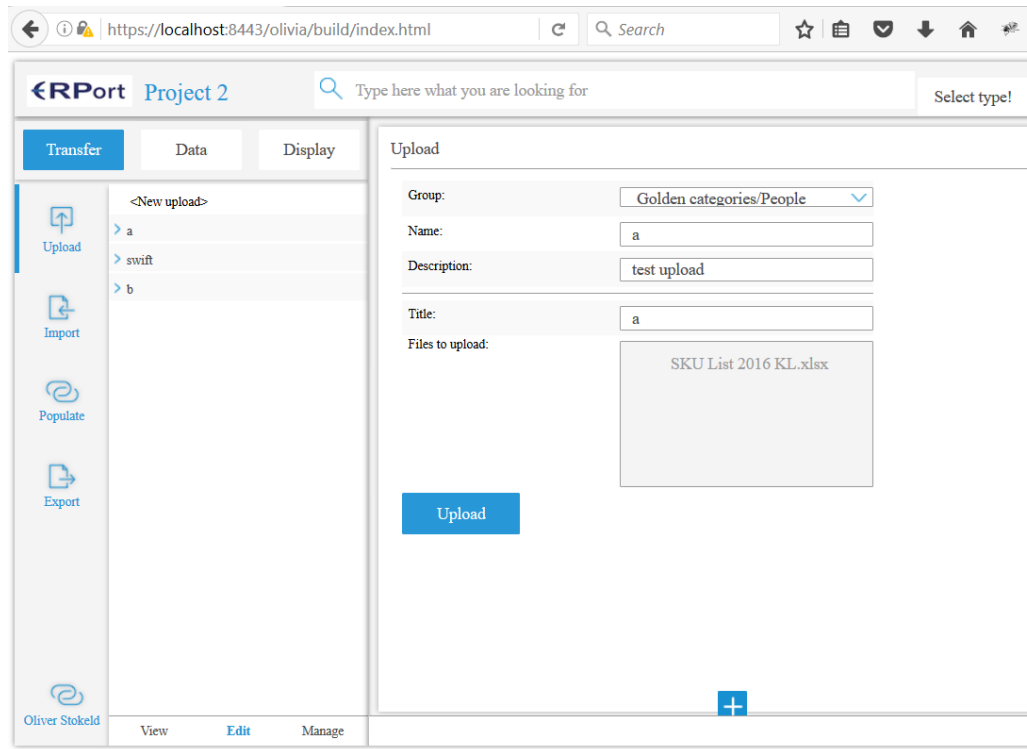
A felhasználói felület kerete ReactJS alapú, de az interakciót már az általános middleware réteg valósítja meg. ERPorttól független a lapozható táblázat, a szerkeszthető automatikus entitás adatlap, és az állapotkezelés is.



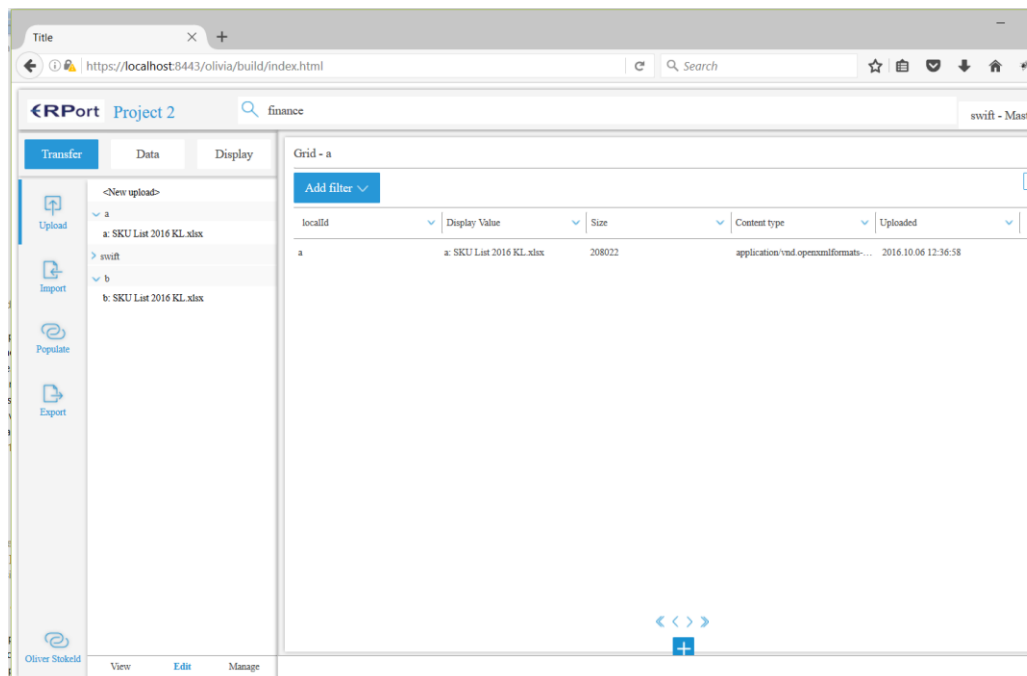
**Ábra 10 Az ERPort projekt összes Java forráskódja**

## MELLÉKLETEK

### 6.3.2 KÉPERNYŐKÉPEK



Ábra 11 Excel állomány feltöltése az ERPort tudásbázisába



Ábra 12 A feltöltött állományok megjelenítése táblázatban

## MELLÉKLETEK

Brand	NOM D'AFFICHAGE	TYPE DE BOITE AUX LETTRES	ALIAS
BK	Admin Bykilian	✓ Brand	in
BK	Ali KHOKHER	✓ NOM D'AFFICHAGE	
BK	Amanda AHMED	✓ TYPE DE BOITE AUX LETTRES	anda.ahmed
BK	Aigoud NADRCHINA	ADRESSE DE MESSAGERIE	stant.comptable
BK	Boutique CAMBON	✓ ALIAS	itique.cambon
BK	Boutique Lugano	ADRESSES DE MESSAGERIE	itique.lugano
BK	Boutique Mayfair London	✓	itique.mayfairlondon
BK	Bruno VERDIER	Utilisateur	bruno.verdier
BK	Camille TENDREL	Utilisateur	camille.tendrel
BK	Capucine CHANSAREL	Utilisateur	capucine.chansarel
BK	Caroline MAS	Utilisateur	caroline.mas
BK	Cesar	Utilisateur	cesar
BK	Christopher HOPKINSON	Utilisateur	Christopher
BK	Clarisse LAVAT	Utilisateur	clarisse.lavat

Ábra 13 Az Excel tábla adatai lapozható táblázatban

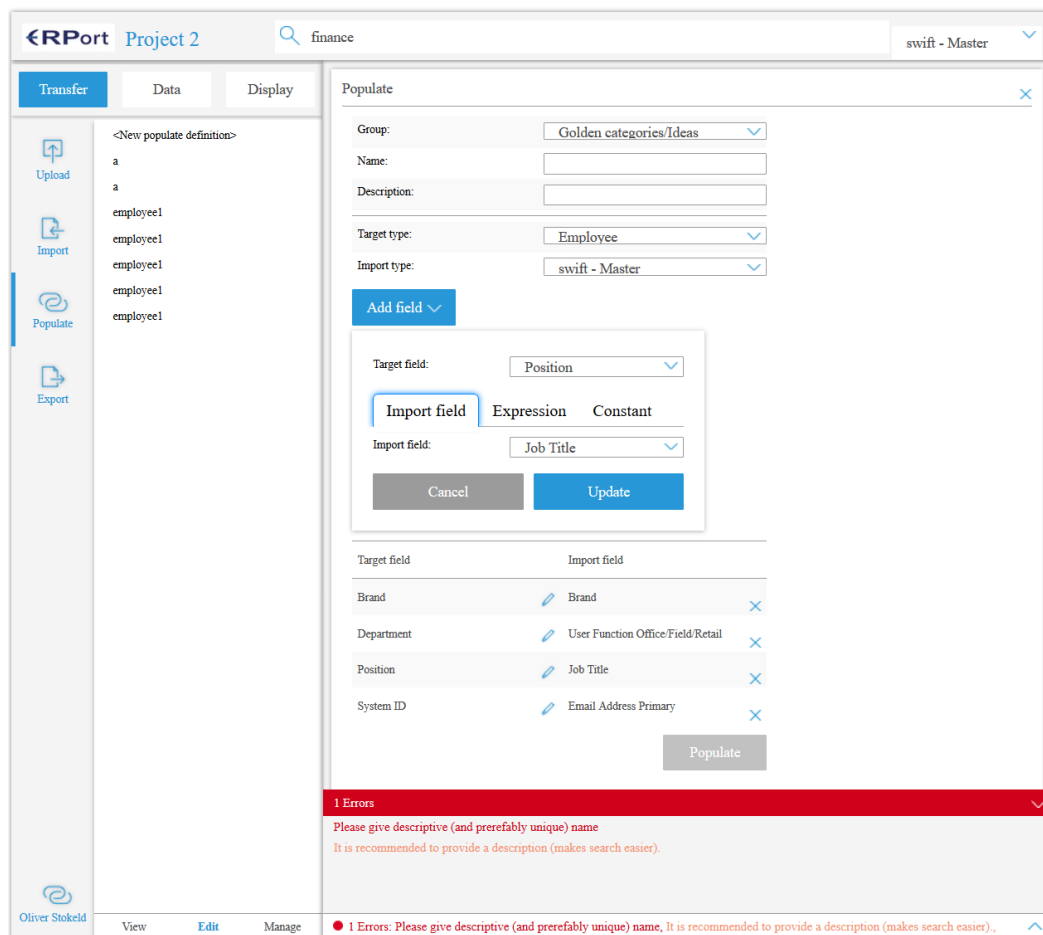
A feltöltéskor létrehozott típus, az ilyen típusú entitások teljesen ugyanolyanok, mint a rendszer bármely fejlesztés során létrehozott típusa és entitása, az alapszolgáltatások ugyanúgy működnek. „Manage” módban az entitás sorokban törlés gombok jelennek meg.

Category	SubCategory	ProductLine	Code	Description
FREDERIC MAL...	hn	Musc Ravageur	N01V50	Musc Ravageur 50ml
FREDERIC MAL...	1	Musc Ravageur	N01V100	Musc Ravageur 10...
FREDERIC MAL...		Musc Ravageur	N01VTEST	Testeur Musc Rava...
FREDERIC MAL...		Musc Ravageur		Mus Ravageur 10ml
FREDERIC MAL...		Musc Ravageur	N3201V11	Musc Ravageur 3...
FREDERIC MAL...		Musc Ravageur	N01V035	Musc Ravageur 3...
FREDERIC MAL...	ikj	Fleur de Cassie	N02V50	Fleur de Cassie 50ml
FREDERIC MAL...		Fleur de Cassie	N02V100	Fleur de Cassie 10...
FREDERIC MAL...		Fleur de Cassie	N02VTEST	Testeur Fleur de C...
FREDERIC MAL...		Fleur de Cassie	N02V10	Fleur de Cassie 10ml
FREDERIC MAL...		Fleur de Cassie	N3202V10	Fleur de Cassie 3X...

Ábra 14 Automatikus szerkeszthető adatlap, állapot kezelés

## MELLÉKLETEK

A szerkesztés módban aktiválható adatlap az entitás modelljeinek típusa alapján automatikusan épül fel. Teszt kedvéért minden mező kapott egy „Warning” jelzést, ha nincs kitöltve (ha Error lenne, a Save gomb szürke lenne).



ERPProject 2 finance swift - Master

Transfer Data Display

Upload Import Populate Export

<New populate definition>

a

a

employee1

employee1

employee1

employee1

employee1

Populate

Group: Golden categories/Ideas

Name:

Description:

Target type: Employee

Import type: swift - Master

Add field

Target field: Position

Import field

Expression

Constant

Import field: Job Title

Cancel Update

Target field	Import field
Brand	Brand
Department	User Function Office/Field/Retail
Position	Job Title
System ID	Email Address Primary

Populate

1 Errors

Please give descriptive (and preferably unique) name  
It is recommended to provide a description (makes search easier).

Oliver Stokeld View Edit Manage 1 Errors: Please give descriptive (and preferably unique) name, It is recommended to provide a description (makes search easier).

Ábra 15 Populate paraméterezés, állapotkezelés

Minden művelet, így a Populate (az importált táblákból másik típushoz tartozó entitások létrehozása) szintén egy entitás, a benne lévő mező másoló műveletek „gyermek entitások” a parancson belül. Következésképpen a Populate parancs szerkesztése az entitás szerkesztővel is megvalósítható (bár itt a fejlesztési feladatok időzítése miatt ez egyedi panelt kapott). Az állapot megjelenítését viszont ugyanaz a komponens végzi itt is, mint az entitás szerkesztő esetén.

A kiadott parancs entitásként mentődik a serveren, és bekerül a baloldali listába, kattintással aktiválható – ehhez egyedi kódolásra nincs szükség, a keret és a middleware alapszolgáltatásai végzik.

## 6.4 ADATBÁZIS KEZELÉS

A következő JSON példák az is látható, hogy egy valós modul már komoly méreteket ölthet. A platformból nagyon hiányzik a közvetlen entitás szerkesztő (amely elkészült ugyan az ERPort környezetében, de a felhasznált JavaScript eszköztár nincs összhangban a további célokkal). A forráskód kivonatok a platform jellegzetességeire mutatnak példákat.

### 6.4.1 EAV TÁBLÁK

```
CREATE TABLE `dustEavEntity` (
  `entityid` bigint(20) NOT NULL AUTO_INCREMENT,
  `primarymodeltype` varchar(60) NOT NULL,
  `uniqueid` varchar(200),
  PRIMARY KEY (`entityid`),
  UNIQUE KEY `idx_entity` (`primarymodeltype`, `uniqueid`),
  KEY `idx_primarytype` (`primarymodeltype`)
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8;

CREATE TABLE `dustEavModel` (
  `entityid` bigint(20) NOT NULL,
  `modeltype` varchar(60) NOT NULL,
  UNIQUE KEY `idx_entitymodels` (`entityid`, `modeltype`),
  KEY `idx_entitymodels1` (`entityid`),
  KEY `idx_entitymodels2` (`modeltype`),
  KEY `fk_entitymodels_entities` (`entityid`),
  CONSTRAINT `fk_entitymodels_entities` FOREIGN KEY (`entityid`)
REFERENCES `dustEavEntity` (`entityid`) ON DELETE CASCADE ON UPDATE NO
ACTION
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8;

CREATE TABLE `dustEavValue` (
  `entityid` bigint(20) NOT NULL,
  `modeltype` varchar(60) NOT NULL,
  `attr` varchar(60) NOT NULL,
  `arrIdx` int NOT NULL,
  `valStr` varchar(4000) DEFAULT NULL,
  `valRef` bigint(20) DEFAULT NULL,
  `valInt` bigint(20) DEFAULT NULL,
  `valFloat` float DEFAULT NULL,
  `valBool` tinyint DEFAULT NULL,
  `valDate` datetime DEFAULT NULL,
  KEY `idx_entitydata2` (`entityid`, `modeltype`, `attr`),
  KEY `fk_entitydata_model` (`entityid`, `modeltype`),
  CONSTRAINT `fk_entitydata_model` FOREIGN KEY (`entityid`, `modeltype`)
REFERENCES `dustEavModel` (`entityid`, `modeltype`) ON DELETE CASCADE ON
UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `dustStreamContent` (
  `entityid` bigint(20) NOT NULL,
  `streamContent` mediumblob DEFAULT NULL,
  KEY `idx_entitystream` (`entityid`),
  CONSTRAINT `fk_entitystream` FOREIGN KEY (`entityid`) REFERENCES
`dustEavEntity` (`entityid`) ON DELETE CASCADE ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```



## 6.4.2 UNIT KONFIGURÁCIÓ

```

{
  "typeId": "Dust:Meta:Unit",
  "id": "Database",
  "Models": [
    {
      "typeId": "Dust:Meta:Unit",
      "Values": {
        "Types": [
          {
            "typeId": "Dust:Meta:Type",
            "id": "Dust:Database:DatabaseServer",
            "Models": [
              {
                "typeId": "Dust:Meta:Type",
                "Values": {
                  "mode": {
                    "typeId": "Dust:Utils:Identified",
                    "id": "PrimaryOptional"
                  },
                  "Service": {
                    "typeId": "Dust:Meta:Service",
                    "id": "Dust:Database:DatabaseServer"
                  },
                  "Fields": [
                    {
                      "typeId": "Dust:Meta:FieldDef",
                      "id": "driver",
                      "Models": [
                        {
                          "typeId": "Dust:Meta:FieldDef",
                          "Values": {
                            "ValueType": {
                              "id": "Text",
                              "typeId": "Dust:Utils:Identified"
                            }
                          }
                        }
                      ]
                    },
                    {
                      "typeId": "Dust:Meta:FieldDef",
                      "id": "dbPath",
                      "Models": [
                        {
                          "typeId": "Dust:Meta:FieldDef",
                          "Values": {
                            "ValueType": {
                              "id": "Text",
                              "typeId": "Dust:Utils:Identified"
                            }
                          }
                        }
                      ]
                    },
                    {
                      "typeId": "Dust:Meta:FieldDef",
                      "id": "connTimeoutSec",
                      "Models": [
                        {
                          "typeId": "Dust:Meta:FieldDef",
                          "Values": {

```

## MELLÉKLETEK

```

        "ValueType":{
            "id":"Text",
        }
    "typeId":"Dust:Utils:Identified"
    }
    }
    ]
    },
    {
        "typeId":"Dust:Meta:FieldDef",
        "id":"DefaultAccount",
        "Models":[
            {
                "typeId":"Dust:Meta:FieldDef",
                "Values":{
                    "ValueType":{
                        "id":"Link",
                    }
                }
            }
        ]
    },
    {
        "typeId":"Dust:Utils:Identified"
    },
    {
        "RelType":"Dust:Access:User"
    }
    }
    ]
    }
    ]
    }
    ]
    },
    {
        "typeId":"Dust:Meta:Type",
        "id":"Dust:Database:EntityStore",
        "Models":[
            {
                "typeId":"Dust:Meta:Type",
                "Values":{
                    "mode":{
                        "typeId":"Dust:Utils:Identified",
                        "id":"PrimaryOptional"
                    },
                    "Service":{
                        "typeId":"Dust:Meta:Service",
                        "id":"Dust:Database:EntityStore"
                    },
                    "Fields":[
                        {
                            "typeId":"Dust:Meta:FieldDef",
                            "id":"DbServer",
                            "Models":[
                                {
                                    "typeId":"Dust:Meta:FieldDef",
                                    "Values":{
                                        "ValueType":{
                                            "id":"Link",
                                        }
                                    }
                                }
                            ]
                        }
                    ]
                }
            }
        ]
    },
    {
        "typeId":"Dust:Utils:Identified"
    },
    {
        "RelType":"Dust:Database:DatabaseServer"
    }
    }
    ]
    }

```

## MELLÉKLETEK

```
[
    ],
    "Services": [
        {
            "typeId": "Dust:Meta:Service",
            "id": "Dust:Database:DatabaseServer",
            "Models": [
                {
                    "typeId": "Dust:Meta:Service",
                    "Values": {
                        "ImplementedServices": [
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:TextHandler"
                            },
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:Block"
                            },
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:Stateful"
                            }
                        ]
                    }
                }
            ]
        },
        {
            "typeId": "Dust:Meta:Service",
            "id": "Dust:Database:EntityStore",
            "Models": [
                {
                    "typeId": "Dust:Meta:Service",
                    "Values": {
                        "ImplementedServices": [
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:Explorer"
                            },
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:Stateful"
                            },
                            {
                                "typeId": "Dust:Meta:Service",
                                "id": "Dust:Utils:Processor"
                            }
                        ]
                    }
                }
            ]
        }
    ]
}
```

## 6.4.3 MODUL KONFIGURÁCIÓ

```

{
  "typeId": "Dust:Binary:Module",
  "id": "JdbcMySQL",
  "Models": [
    {
      "typeId": "Dust:Binary:Module",
      "Values": {
        "LogicAssignments": [
          {
            "typeId": "Dust:Binary:Logic",
            "Models": [
              {
                "typeId": "Dust:Binary:Logic",
                "Values": {
                  "Service": {
                    "typeId": "Dust:Meta:Service",
                    "id": "Dust:Database:DatabaseServer"
                  }
                },
                "Class": "com.dustcomp.dust.db.mysql51.DustCompJdbcServer"
              }
            ],
            "Class": "com.dustcomp.dust.db.mysql51.DustCompJdbcServer"
          },
          {
            "typeId": "Dust:Utils:Tagged",
            "Values": {
              "Tags": [
                {
                  "typeId": "Dust:Utils:Tag",
                  "id": "Dust_Binary_LogicFlags_AutoInit"
                }
              ]
            }
          }
        ]
      },
      "Class": "com.dustcomp.dust.db.mysql51.DustCompJdbcEavStore"
    }
  ]
}

```

#### 6.4.4 TESZT ALKALMAZÁS KONFIGURÁCIÓ

Itt látható a JSON konfiguráció „kreatív alkalmazása” a fejlesztés során. „Elrontott” mezőnevekkel félre lehet tenni különféle változatokat, könnyű változtatni a teszt esetek között. Itt például `InitMessagesX` – a konnektor tesztelése az első `select` utasítással; `EntityBuilderX` – az `EAVStore` tesztelése, a valós entitás létrehozás helyett JSON kimenet. Az is látható, hogy a felhasználót a `RootEntities` listában deklarálom, a konnektorban csak hivatkozom rá. Ez éles környezetben külön forrásokba helyezhető, akár hozzáférésvédelemmel is ellátható, a rá hivatkozó részrendszer nem látja az felhasználói adatokat.

```
{
  "typeId": "Dust:Utils:Application",
  "id": "SQLTest",
  "Models": [
    {
      "typeId": "Dust:Utils:Application",
      "Values": {
        "RootEntities": [
          {
            "typeId": "Dust:Access:User",
            "id": "erport",
            "Models": [
              {
                "typeId": "Dust:Utils:Identified",
                "Values": {
                  "id": "erport"
                }
              }, {
                "typeId": "Dust:Access:User",
                "Values": {
                  "password": "very5ecret",
                  "testMulti": [
                    "aa",
                    "bbb",
                    "cccc"
                  ],
                  "Roles": [
                    {
                      "typeId": "Dust:Access:Role",
                      "id": "ERPortUser"
                    }
                  ]
                }
              }
            ]
          }
        ]
      }, {
        "typeId": "Dust:Utils:FormatJsonWriter",
        "numId": "121",
        "Models": [
          {
            "typeId": "Dust:Utils:Chain",
            "Values": {
              "Next": {
                "typeId": "Dust:Stream:Handler",
                "Models": [

```

## MELLÉKLETEK

```
{
  "typeId": "Dust:Utils:Chain",
  "Values": {
    "Next": {
      "typeId": "Dust:Stream:ProviderSystem"
    }
  }
}
]
}
}
]
}, {
  "typeId": "Dust:Database:DatabaseServer",
  "numId": "111",
  "Models": [
    {
      "typeId": "Dust:Database:DatabaseServer",
      "Values": {
        "driver": "com.mysql.jdbc.Driver",
        "dbPath": "jdbc:mysql://localhost:3306/erport",
        "useSsl": "false",
        "autoReconnect": "true",
        "DefaultAccount": {
          "typeId": "Dust:Access:User",
          "id": "erport"
        }
      }
    }
  ]
}, {
  "typeId": "Dust:Database:EntityStore",
  "numId": "112",
  "Models": [
    {
      "typeId": "Dust:Database:EntityStore",
      "Values": {
        "DbServer": {
          "typeId": "Dust:Database:DatabaseServer",
          "numId": "111"
        },
        "EntityBuilder": {
          "typeId": "Dust:Utils:FormatJsonWriter",
          "numId": "121"
        },
        "EntityBuilderX": {
          "typeId": "Dust:Utils:DataBuilder"
        }
      }
    }
  ]
}, {
  "typeId": "Dust:Utils:Scanner",
  "numId": "113",
  "Models": [
    {
      "typeId": "Dust:Utils:Chain",
      "Values": {
        "Next": {
          "typeId": "Dust:Database:EntityStore",
          "numId": "112"
        }
      }
    }
  ]
}
```

## MELLÉKLETEK

```

    }
    }, {
      "typeId": "Dust:Utils:Scanner",
      "Values": {
        "numId": "false"
      }
    }
  ]
}
],
"InitMessages": [
  {
    "typeId": "Dust:Utils:Message",
    "Models": [
      {
        "typeId": "Dust:Utils:Message",
        "Values": {
          "command": "Dust:Utils:Processor!Process",
          "Target": {
            "typeId": "Dust:Utils:Scanner",
            "numId": "113"
          }
        }
      }
    ],
    {
      "typeId": "Dust:Utils:Variant",
      "Values": {
        "Link": {
          "typeId": "Dust:Access:User",
          "id": "erport"
        }
      }
    }
  ]
}
],
"InitMessagesX": [
  {
    "typeId": "Dust:Utils:Message",
    "Models": [
      {
        "typeId": "Dust:Utils:Message",
        "Values": {
          "command": "Dust:Utils:TextHandler!ProcessLine",
          "Target": {
            "typeId": "Dust:Database:DatabaseServer",
            "numId": "111"
          }
        }
      }, {
        "typeId": "Dust:Utils:Variant",
        "Values": {
          "Text": "select * from dustEavEntity;"
        }
      }
    ]
  ]
}
]
}

```

### 6.4.5 SZERVER KONNEKTOR FORRÁS

A megvalósító Java osztály implementálja a szolgáltatásokhoz rendelt interfészeket. Stateful amiatt, hogy a konstans paraméterek és a kapcsolat inicializálása az egyéb műveletektől függetlenül, automatikusan megtörténhessen, Block kezelő, amelyet tranzakciók kezelésére használ, illetve TextHandler, SQL parancsokat fogad. „Gyors kiegészítés” keretében a ProcessLine parancs külön paramétereket kapott a prepared statementek kezeléséhez (így a sokszor ismételt parancsok SQL értelmezése elmarad, csak az új paramétereket kell kezelni).

```
public class DustCompJdbcServer implements DustUtilsStateful,
DustUtilsTextHandler, DustUtilsBlock, DustCompJdbcComponents {
```

A függvényben látható a működési paraméterek beolvasása és a driver inicializálás. A shutdown hook kezelése természetesen nincs jó helyen, a Stateful objektumok felszabadítását a kernelnek kell majd kezelnie, de ez az automatizmus komplex tervezést igényel (felszabadíthatóság paraméterezése, a foglalási sorrend kezelése, esetleges függések, különösen „lazy” inicializálás esetén). A valós kapcsolat zárás a release függvénybe került, átmenetileg a hívása helyben történik.

```
@Override
public void dust_utils_stateful_init() throws Exception {
    jdbcDriver = (String) DustCompApiUtils.getValue(FLD_DRIVER);
    dbUrl = (String) DustCompApiUtils.getValue(FLD_DBPATH);

    connProps = new Properties();
    connProps.setProperty("useSSL", (String)
DustCompApiUtils.getValue(FLD_USESSL));
    connProps.setProperty("autoReconnect", (String)
DustCompApiUtils.getValue(FLD_AUTORECONNECT));

    timeoutSec = DustCompApiHacks.getInt(DustIdentifier.ICTX_THIS,
FLD_CONNTIMEOUTSEC, 5);

    if (null != DustCompApiUtils.getValue(FLD_DEFACCOUNT)) {
        defAccount = new
Account(DustCompApiUtils.getValue(FLD_DEFACCOUNT, TYPE_IDENTIFIED,
FLD_ID), DustCompApiUtils.getValue(FLD_DEFACCOUNT, TYPE_ACCESS_USER,
FLD_PASSWORD));
    }

    if (!knownDrivers.contains(jdbcDriver)) {
        Class.forName(jdbcDriver);
        knownDrivers.add(jdbcDriver);
    }

    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
```



## MELLÉKLETEK

```
        try {
            dust_utils_stateful_release();
        } catch (Exception e) {
            DustCompApiUtils.wrapException(e);
        }
    }
});
}

@Override
public void dust_utils_stateful_release() throws Exception {
    if ((null != dbConn) && dbConn.conn.isValid(timeoutSec)) {
        if (!dbConn.conn.getAutoCommit()) {
            dbConn.conn.rollback();
        }
        dbConn.conn.close();
    }
}
```

Block parancsok „lefordítása” a tranzakciókezelés vezérléséhez.

```
@Override
public void dust_utils_block_start() throws Exception {
    getConn(false);
}

@Override
public void dust_utils_block_end() throws Exception {
    if ((null != dbConn) && !dbConn.conn.getAutoCommit()) {
        dbConn.conn.commit();
    }
}
```

Így lesz a TextHandler fogadó oldala SQL végrehajtó szolgáltatás...

```
void addCommandFragment(String str) throws Exception {
    sbCmd.append(str);
    boolean direct = (boolean)
DustCompApiUtils.getParamValue(FLD_SQLDIRECT);

    if (direct || str.endsWith(SQL_STATEMENT_TERMINATOR)) {
        str = sbCmd.toString();
        DustCompUtils.sbClear(sbCmd);
        DBConn dbConn = getConn(true);
        ResultSet rs = dbConn.execStatement(str);

        DustCompApiUtils.setParamValue(FLD_RESPONSE, rs);
    }
}

@Override
public void dust_utils_texthandler_processLine() throws Exception {
    addCommandFragment((String)
DustCompApiUtils.getParamValue(VAR_TEXT));
}

@Override
public void dust_utils_texthandler_processChar() throws Exception {
    addCommandFragment((String)
DustCompApiUtils.getParamValue(VAR_TEXT));
}
```

### 6.4.6 EAVSTORE FORRÁS

A megvalósító Java osztály itt is implementálja a szolgáltatásokhoz rendelt interfészeket. Explorer szerepben az történik, hogy a feldolgozó csatornán „előtte” álló komponens lényegében SAX bejárás eseményeket küld, ez lehet egy kernel Scanner, amely egy „élő” entitást jár be, de ugyanez érkezhethet egy JSON file parsertől, vagy akár egy ugyanilyen EAVStore-tól is. A csatornán következő entitás egy TextHandler, amelynek ez a szolgáltatás a kapott hívások és az aktuális állapot alapján SQL parancsokat küld.

```
public class DustCompJdbcEavStore implements DustUtilsExplorer,
DustUtilsProcessor, DustCompJdbcComponents, DustCompSerializeNames,
DustCompJdbcEntityNames {
```

A ProcessLine parancs elküldése az entitás TextHandler szolgáltatást megvalósító „kimenő” csatornájára (feltételezve, hogy ez Java ResultSet objektumot fog visszaadni; ha a hívó felkészült arra, hogy ez akár null is lehet, a TextHandler lehet akár egy konzol kiírás is).

```
    ResultSet execStmt(String sql) {
        DustCompApiUtils.setMsgValue(CMD_PROCESSLINE, VAR_TEXT, sql, true);
        return (ResultSet) DustCompApiUtils.getMsgValue(CMD_PROCESSLINE,
FLD_RESPONSE);
    }
```

Ugyanez a szolgáltatás PreparedStatement-ekkel kezelt sorok számára. Ez egy „gyors megoldás”, a ProcessLine ilyen módon való kiegészítése nem elegáns, a helyes megoldás a Message entitásba helyezett másik modell használata lenne. Az átalakítás pár percet vesz majd igénybe.

```
    void prepStmt(String sql, int parCount, Object[] pars) {
        DustCompApiUtils.setMsgValue(CMD_PROCESSLINE, VAR_TEXT, sql,
false);
        DustCompApiUtils.setMsgValue(CMD_PROCESSLINE, FLD_PREPDATACOUNT,
parCount, false);
        DustCompApiUtils.setMsgValue(CMD_PROCESSLINE, FLD_PREPDATAVALUES,
pars, false);
    }

    ResultSet execStmt(String sql, int parCount, Object[] pars) {
        prepStmt(sql, parCount, pars);
        DustCompApiUtils.sendMsg(CMD_PROCESSLINE);
        return (ResultSet) DustCompApiUtils.getMsgValue(CMD_PROCESSLINE,
FLD_RESPONSE);
    }
```

## MELLÉKLETEK

A Block üzenetek feladata kettős: követni az érkező adat szerkezetét, illetve az üzeneteket továbbítani. Például a BlockStart üzenet feldolgozása:

```
public void dust_utils_block_start() throws Exception {
    String bm = (String) DustCompApiUtils.getParamValue(FLD_BLOCKTYPE);

    if (SERIALIZE_BLOCKTYPE_MASTER.equals(bm)) {
        buildStack.clear();
        BuildItem bi = (BuildItem) buildStack.push();
        bi.setState(READ_ENTITY);

        initDbConn();

        DustCompApiUtils.sendMsg(CMD_BLOCK_START);
    } else {
        BuildItem bi = (BuildItem) buildStack.peek();

        switch (bi.getState()) {
            case READ_MODEL:
                if (bi.checkField(SERIALIZE_STR_VALUES)) {
                    bi.setState(READ_VALUE);
                    bi.setField(null);
                }
                break;
            case READ_VALUE:
                if (SERIALIZE_BLOCKTYPE_ARRAY.equals(bm)) {
                    bi.setSingle(false);
                } else {
                    System.out.println("Going down to level " +
buildStack.getSize());
                    ((BuildItem) buildStack.push()).setState(READ_ENTITY);
                }
                break;
        }
    }
}
```

Ugyanez az osztály valósítja meg az entitások beolvasását EAV adattáblákból, ez itt egy egyszerűsített változat. A Process üzenet paramétere az entitás belső azonosítója. A már látott prepared statement olvasással ez egy utasítás, a válasz ResultSet objektumban érkezik, a select order by gondoskodik arról, hogy az érkező sorok megfelelő sorrendben vannak. A feldolgozó csatorna következő eleme egy Explorer, amelynek most a „bejárás” SAX eseményeit fogjuk elküldeni. Látható, hogy valóban eseményeket ismerünk fel: a kulcs mezők változása egy új entitás, modell vagy mező kezdetét jelenti, ezért hívódnak az optCloseXXX függvények, majd az aktuális elem által hordozott új információ küldése következik.

## MELLÉKLETEK

```
@Override
public void dust_utils_processor_process() throws Exception {
    initDbConn();

    Object[] pars = new Object[1];
    pars[0] = DustCompApiUtils.getParamValue(FLD_STOREID);
    // ResultSet rs = execStmt(STMT_SELECTBYID, 1, pars);

    ResultSet rs = execStmt(STMT_SELECTBYIDSET.replace("%par1%", "13,
14"));

    Object val;
    Long l;
    String s;
    while (rs.next()) {
        if (null == eid) {
            DustCompApiUtils.initRelay(EXPLORER_CMDS_ALL,
FLD_ENTITYBUILDER);
            DustCompUtilsSerializer.manageContentBlock(true,
SERIALIZE_BLOCKTYPE_MASTER, false);
        }

        l = rs.getLong(1);
        if (!DustCompUtils.isEqual(eid, l)) {
            optCloseEntity();

            eid = l;
            modelType = attr = null;

            DustCompUtilsSerializer.manageContentBlock(true, null,
false);

            DustCompUtilsSerializer.sendFieldSimple(FLD_NUMID,
eid.toString());
            DustCompUtilsSerializer.sendFieldSimple(FLD_TYPEID,
rs.getObject(2));
            DustCompUtilsSerializer.sendFieldSimple(FLD_ID,
rs.getObject(3));

            DustCompUtilsSerializer.manageContentField(true,
SERIALIZE_STR_MODELS);
            DustCompUtilsSerializer.manageContentBlock(true,
SERIALIZE_BLOCKTYPE_ARRAY, false);
        }

        s = rs.getString(4);
        if (!DustCompUtils.isEqual(modelType, s)) {
            optCloseModel();

            modelType = s;
            attr = null;

            DustCompUtilsSerializer.manageContentBlock(true, null,
false);

            DustCompUtilsSerializer.sendFieldSimple(FLD_TYPEID,
modelType);

            DustCompUtilsSerializer.manageContentField(true,
SERIALIZE_STR_VALUES);
            DustCompUtilsSerializer.manageContentBlock(true,
SERIALIZE_BLOCKTYPE_SET, false);
        }
    }
}
```

## MELLÉKLETEK

```
s = rs.getString(5);
l = rs.getLong(6);
if (!DustCompUtils.isEqual(attr, s)) {
    optCloseAttr();

    attr = s;
    multi = SINGLEVALUE != l.intValue();

    DustCompUtilsSerializer.manageContentField(true, attr);
    if (multi) {
        DustCompUtilsSerializer.manageContentBlock(true,
SERIALIZE_BLOCKTYPE_ARRAY, false);
    }
}

val = rs.getObject(7);

if (null == val) {
    for (int i = 9; (null == val) && (i < EAVFIELDS.length);
++i) {
        val = rs.getObject(i);
    }
} else {
    l = rs.getLong(8);

    if (!rs.isNull()) {
        DustCompUtilsSerializer.manageContentBlock(true, null,
true);
        DustCompUtilsSerializer.sendFieldSimple(FLD_NUMID,
l.toString());
        DustCompUtilsSerializer.sendFieldSimple(FLD_TYPEID, val);
// TODO Remove this!
        DustCompUtilsSerializer.sendFieldSimple(FLD_REFMODEL,
val);
        DustCompUtilsSerializer.manageContentBlock(false, null,
true);
        val = null;
    }

    if (null != val) {
        DustCompUtilsSerializer.sendFieldContent(val);
    }
}

optCloseEntity();

if (null != eid) {
    DustCompUtilsSerializer.manageContentBlock(false,
SERIALIZE_BLOCKTYPE_MASTER, false);
}
}
```

## MELLÉKLETEK

Megjegyzés: bár ez a függvény a feladat gyorsan begépelhető, viszonylag átlátható megoldása, a Dust Platform elveinek ismeretében elég rossz, a fejlesztés lépései is láthatóvá válnak.

1. Ciklust futtat. Ez a lépés átkerülhetne a DB konnektor oldalára, amely a ResultSet-en iterálva az aktuális elemet tenné egy entitásba, és soronként hívná az EAVStore-t.
2. Ezzel megszűnne a kapcsolat az EAVStore és az adatbázis között, az „EAV formátum kezelő” megjelenhet általános szinten más adatforrások (Excel vagy CSV, ...) feldolgozása során. Viszont...
3. Egy olyan komponens, amely megadott mezőket figyel, ezek változása esetén előre definiált üzeneteket küld, nem igényel forráskódot! A „monitor” teljesen általános, konfiguratív algoritmus, az EAV tudás ennek egy konfigurációja, csupán adatszerkezet.

## 6.5 ÁBRAJEGYZÉK

Ábra 1 A tudás praktikus, „fehér doboz” modellje .....	13
Ábra 2 A tudásreprezentáció fejlődése .....	14
Ábra 3 A csoportvezető Margaret Hamilton és az AGC forráskódja .....	17
Ábra 4 Az eszközök és a virtuális világ fejlődése.....	18
Ábra 5 A szoftver alapelemei, egymásra hatásuk a rendszer élete során.....	23
Ábra 6 Hello, World! (2) kimenet – a szöveget saját kód állítja elő.....	80
Ábra 7 Hello, World! (2) - entitás szerkezet grafikus exportja, feliratozva....	81
Ábra 8 Hello, World! (3) - a webes szolgáltatás a Test process linkkel .....	82
Ábra 9 Hello, World! (3) - a linkre kattintva megérkezik a greeting szöveg..	82
Ábra 10 Az ERPort projekt összes Java forráskódja .....	84
Ábra 11 Excel állomány feltöltése az ERPort tudásbázisába.....	85
Ábra 12 A feltöltött állományok megjelenítése táblázatban .....	85
Ábra 13 Az Excel tábla adatai lapozható táblázatban .....	86
Ábra 14 Automatikus szerkeszthető adatlap, állapot kezelés .....	86
Ábra 15 Populate paraméterezés, állapotkezelés .....	87

## 6.6 CD MELLÉKLET TARTALMA

A szakdolgozat CD mellékletének szerkezete:

- KedvesLorand\_ST3BJ3\_Diploma\_DinamikusTudasreprezentacio.docx
- KedvesLorand\_ST3BJ3\_Diploma\_DinamikusTudasreprezentacio.pdf
- prezentaciok/
- projektek/
- hivatkozások/

## 6.7 IRODALOMJEGYZÉK

1. FOWLER, Martin. Intentional Software. [online]. 2009. [Hivatkozva: 2016. október 30.]. <http://www.martinfowler.com/bliki/IntentionalSoftware.html>
2. FOWLER, Martin. Microservices. [online]. 2014. [Hivatkozva: 2016. október 30.]. <http://www.martinfowler.com/articles/microservices.html>
3. GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
4. GARCÍA-BORGONÓN, Laura, et al. Software process modeling languages: A systematic literature review. *Information and Software Technology*, 2014, 56.2: 103-116.
5. General Electric, 2016, What is Predix Platform?. [online]. [Hivatkozva: 2016. október 26.]. <https://www.predix.io/docs#ZfI6Ip5V>
6. HENDERSON-SELLERS, Brian; GONZALEZ-PEREZ, Cesar. The rationale of powertype-based metamodeling to underpin software development methodologies. In: *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43*. Australian Computer Society, Inc., 2005. p. 7-16.
7. IBM, 2016, What is IBM Bluemix?. [online]. [Hivatkozva: 2016. október 26.]. <https://www.ibm.com/developerworks/cloud/library/cl-bluemixfoundry/>
8. MAKABEE, Hayim. Adaptable Design Up Front. 2015. [Online] [Hivatkozva: 2016. október 30.] <https://effectivesoftwaredesign.com/2015/12/24/talk-adaptable-design-up-front-slides-video/>
9. MEYEROVICH, Leo A.; RABKIN, Ariel S. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, 2013, 48.10: 1-18.



10. Microsoft Corporation, 2016, Why Azure?. [online]. [Hivatkozva: 2016. október 26.]. <https://azure.microsoft.com/en-gb/overview/what-is-azure/>
11. NAKAGAWA, Elisa Y., et al. The state of the art and future perspectives in systems of systems software architectures. In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. ACM, 2013. p. 13-20.
12. PARTRIDGE, Chris; GONZALEZ-PEREZ, Cesar; HENDERSON-SELLERS, Brian. Are conceptual models concept models?. In: *International Conference on Conceptual Modeling*. Springer Berlin Heidelberg, 2013. p. 96-105.
13. RUMPE, Bernhard. Executable Modeling with UML. A Vision or a Nightmare?. *arXiv preprint arXiv:1409.6597*, 2014.
14. SHANNON, Claude E. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 1938, 57.12: 713-723.
15. SIMONYI, Charles. The death of computer languages, the birth of intentional programming. In: *NATO Science Committee Conference*. 1995. p. 398-399.
16. VÉG, Csaba. *Alkalmazásfejlesztés a Unified Modeling Language szabványos jelöléseivel*. Logos 2000 Bt., 1999.
17. VERHAGEN, Wim JC, et al. A critical review of Knowledge-Based Engineering: An identification of research challenges. *Advanced Engineering Informatics*, 2012, 26.1: 5-15.
18. VON NEUMANN, John; GODFREY, Michael D. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 1993, 15.4: 27-75.
19. WESTFALL, Linda. *The certified software quality engineer handbook*. ASQ Quality Press, 2008.