Dust Framework

Bemutatkozás

Kedves Loránd, 39 éves rendszertervező, programozó matematikus vagyok. Eddigi pályafutásom során két alapvető feladatkörben dolgoztam: meglévő rendszerek elemzése és moduláris átszervezése, illetve új rendszer megtervezése kiforratlan igények alapján.

Egy kétfős cég tagjaként országos hibanaplózó és követő rendszert írtunk akkor, amikor a mai internet még nem is létezett, a hibajegyek továbbítására email-t használtunk.

A Cygron Kft. munkatársaként egy döntéstámogató rendszer elemzése és átstrukturálása volt a feladatom, később több szálon futó adatelemző komponenst terveztem, majd fejlesztését vezettem. A cég adatbányász termékeivel kétszer volt a Comdex számítástechnikai kiállítás döntőse Las Vegasban, 1999-ben a Personal Productivity kategória győztese lett.

Az Ulyssys Kft. munkatársaként egy korábbi dokumentumkezelő rendszerből emeltem ki az általános ügyviteli szolgáltatás szolgáltatásokat, majd erre épültek (és épülnek jelenleg is) a cég hatáskörébe tartozó ügyviteli rendszerek, amelyek között legfontosabb az IIER: a magyar mezőgazdaság országos támogatási és felügyeleti informatikai rendszere. Ezt követően az én feladatom volt a közös szolgáltatásréteg kialakítása, moduljainak tervezése és megvalósítása (törzs karbantartás, iktató, levél, bizonylat, jegyzőkönyv kezelés, munkafolyamatok, adat export/import keret, ...)

Az alábbiakban ismertetett terv igen nagyra törő célokat, és újszerű megközelítéseket tartalmaz, de remélem a bemutatkozásom elég alapot szolgáltat ezek megfontolására.

Feladat

Probléma

Huszonöt éve ültem először számítógép elé programozási céllal. Azóta a számítástechnika elképesztő sebességgel fejlődött minden tekintetben: számítási sebesség, memóriakapacitás, háttértárak mérete, kommunikációs sávszélesség (majd az internet), perifériák (monitor, grafikus felületek, hang), ... területén. Húsz évvel ezelőtt egy teljes vállalatot irányító informatikai rendszer gépeinek összesített kapacitása ma egy telefonban is kevés lenne. Ma huszonöt dollárba kerül, és hitelkártya (vagy éppen pendrive) méretben megvalósítható egy olyan számítógép, amely minden paraméterében meghaladja az én tíz évvel ezelőtti csúcskategóriás fejlesztőgépemet.

Mindezek dacára ma is egyre újabb gépeket vásárolunk, okkal panaszkodunk a lassúságukra, a fenti eredmények mintha nem váltak volna "felhasználói elégedettséggé". Programozóként még rosszabb a helyzet: a programozás kreativitása eltűnőben van, a "piac" lényegében azonos feladatok repetitív megoldását kívánja egyre újabb, és a korábbiakkal sosem kompatibilis környezetekben: futtató rendszerek, programozási nyelvek, szolgáltatáskészletek. A hatalmas, és állandóan fejlesztett eszköztárak mélységeinek ismerete egyre nagyobb súllyal esik latba az eredeti programozói képességekkel szemben, ma nem alkotók, hanem betanított informatikai szalagmunkások adják a szakma túlnyomó többségét. Semmi nincs "kész", nem lehet továbblépni azzal, hogy ezt a feladatot megoldottuk, folyamatosan jönnek ki az új hardver eszközök, operációs rendszerek, ügyviteli programok, szórakoztató alkalmazások – nem a valós szükséglet, hanem a várt nyereség szerint.

Mindez elképzelhetetlen mennyiségű erőforrást szív fel és pazarol el. Tekintetbe véve azt, hogy erőforrásaink végesek, ez egy alapvető problémát jelent, továbbá nem megszünteti, csak vagyoni alapra helyezi és tovább növeli a távolságot az emberek között.

Ez egy valós, kritikus probléma, amelynek okát, illetve megoldásának tervét ismertetem.

Ok

A jelenség oka az, hogy az informatika világa alapvetően eltér minden általunk megszokott környezettől. Az informatika „alapeleme” az információ: adat, esemény, algoritmus, amely sokkal inkább hasonlít egy gondolatra, mint egy tárgyra. A két legfontosabb különbség: a *korlátlan másolhatóság* és az akár *végtelen élettartam*.

A tárgyi világban minden egyes széket, fúrógépet vagy kiskanalat egyenként el kell készíteni. Az informatika „termékei” (szöveg, film, programkód) ha egyszer létrejöttek, az eredeti „példánnyal” azonos értékben, végtelen számban másolhatók, ezek az eredetivel azonos módon, tőle függetlenül használhatók.

A tárgyak a használat során, de anyaguk anélkül is fárad, öregszik, idővel cserére szorul. Ezzel szemben az informatika termékei, amíg létezik az őket tárolni, használni (megjeleníteni, végrehajtani) képes környezet, örök életűek – más szóval: életciklusuk nem használatukhoz (megtekintés, lefuttatás) és nem is a létrehozásuktól eltelt időhöz kötődik, hanem az őket tároló, megjelenítő környezethez.

A világunkat ma alapvetően meghatározó üzleti gondolkodás ezt a két tulajdonságot kezelni képtelen, mert minden eszköze erőforrások kinyerésére, termékek folyamatos előállítására épül. Fogalmai között nem szerepel a feladat végleges megoldása, kizárólag az állandó igény folyamatos kielégítése (esetleg új igények „feltárása”, nem ritkán generálása) – amely viszont az informatika alaptörvényeivel ellentétes. Ez pontosan látszik az üzlet és a gondolati értékek „frontvonalán” (a számítástechnikától függetlenül is): szerzői jogok, szabadalmak kérdésében.

A korai számítástechnika üzleti oldala ezért a hardverrel egységet alkotó programokban és adatokban gondolkodott: hatalmas és igen drága gépek készültek, a programok írása pedig néhány magasan képzett varázsló feladata volt. Mindaddig, amíg nem jött egy ember, aki képes volt üzleti modellt alkotni az erre köztudottan alkalmatlan informatikára, és céget alapított a hardvertől elválasztott számítógépes programok létrehozására: ez volt Bill Gates és a Microsoft (Steve Jobs, az örök „nagy ellenfél” kijelentése szerint).

Az üzleti modell lényege, hogy elválasztja az adatokat és programokat az őket kezelni képes hardvertől, ugyanakkor az új verziókkal (operációs rendszer, nyelv, felhasználói programok, szabványos, tehát cserélhető és egyre fejlettebb hardverek) visszahozza az „avulás” fogalmát: „régi program”, „régi gép”, így az informatikai eszköztár „inflálódik”. A befektetett pénz tehát meg fog térülni, az operációs rendszer, a szövegszerkesztő, stb. többé nem egyszeri megoldott feladat, hanem állandóan kifizettethető szolgáltatás, a programozás nem egyetemi kutatás többé, hanem ugyanolyan folyamatosan üzemeltethető gyár, mint amelyik mondjuk cipőket állít elő.

Ennek köszönhetjük azt, hogy ma otthoni, hétköznapi, a családi költségvetésbe illeszthető eszköz a számítógép, hogy létrejött a világháló, amelynek elérhetősége oly természetes a számunkra. Ezek a tényezők gyökeresen átalakították, és ma még fel sem fogott mértékben megváltoztathatják egész életünket, gondolkodásunkat (bár a folyamat irányítóinak nyilván nem ez lebegett a szeme előtt, hanem hatalmas vagyont szereztek vele).

Ennek látható ára az, amit folyamatosan kifizetünk az új számítógépes alkatrészek, operációs rendszerek, felhasználói programok, játékok, ma már táblagépek, telefonok... megvásárlása során.

Rejtett költsége pedig mindaz az energia és alapanyag, amely az új gépek hihetetlen tömegének előállításához szükséges. Veszteség az, hogy az elavultnak tekintett gépeinket nem adhatjuk tovább a kevésbé tehetőseknek, fejlődő régióknak (tehát nem csökkenthetik a távolságot közöttünk), mert ők sem tudják mire használni: már nem létezik a 3.1-es Windowszal, betárcsázós modemmel elérhető világ. Az ezt futtatni képes gép már értéktelen, veszélyes hulladékként kezelendő szemét, amelyben lassan összegyűlik a világ egyre szűkebb ritkaföldfém-készlete...

A közösség számára hátrányos következményekkel együtt tudnánk élni különösebb korlátok nélkül, viszont az utóbbi költség lehetetlenné teszi a folytatást, váltásra kényszerít minket.

Cél

Cél egy olyan informatikai környezet kialakítása, amely szembe képes fordulni az üzleti megközelítéssel, az információt (adatot és algoritmusokat, programokat egyaránt) saját törvényei szerint kezelni, elfogadni és kihasználni fizikai korlátoktól való függetlenségét, megoszthatóságát. Megalkotni azt a környezetet, amely szabványosítja az információ megjelenítését és felhasználását minden embercsoport számára (hely és nyelv függetlenség, lokalizáció lehetősége), amely a jelenlegi kereteknél jóval mélyebb szinten absztrahálja az információ környezetét (hardver, operációs rendszer, futtató rendszer, szolgáltatások, nyelv), ezáltal lehetőséget biztosít a lényeges tartalom megjelenítésére és használatára ma elavultnak számító eszközökön is. Amennyiben ez a cél megvalósítható, kettős hatást gyakorol világunkra és gondolkodásunkra.

Képes lenne megállítani az informatika mai, állandóan és véleményem szerint jelentős eredmények nélküli, viszont hatalmas erőforrásokat felemésztő malmait, lehetőséget adna arra, hogy a ma „hulladékként" kezelt eszközeink felgyorsíthassák a ma elmaradott régiók számára is a XXI század beköszöntét, továbbá arra, hogy a „fejlett világ" leállhasson az értelmetlen fejlődéssel, mert a valós szükségletek kielégítéséhez megfelelő képességű eszközök ugyanis hatalmas tömegben és olcsó előállíthatóak.

Másrészt bizonyíthatná, hogy az üzleti szemlélet, amely a leghatékonyabbnak bizonyult egy feltáratlan világ meghódítására, képtelen kezelni, és természeténél fogva rettenetes pazarlást indukál egy véges, már belakott világ esetén – ez nem csak az informatikára igaz, hanem egész bolygónkra is alkalmazható állítás. Segítségével bizonyíthatnánk, hogy képesek vagyunk globális együttműködésre, valódi és hosszú távú megoldások keresésére és alkalmazására, először az informatika terén. Ha itt sikerrel járunk, talán nagyobb magabiztossággal lépünk tovább a globális emberi civilizáció együttműködésen és nem értelmetlen versengésen alapuló átszervezése felé.

Szerkezet

A továbbiakban megkísérlem bemutatni a tervezett rendszer alapelveit. Ezek során egy-egy meglehetősen filozofikus felvetéstől igyekszem eljutni a konkrét megvalósítási terv ismertetéséig anélkül, hogy száraz szakszöveggé változna az egész. A konkrét tervek, és a kapcsolódó forráskódok a dust-framework.org honlapon elérhetőek a szakértők számára.

Entitások

Minden számítógépes rendszer feladata alapvetően az, hogy „lenyomatot" hozzon létre az őt körülvevő „valóság" (vagy képzelt világ) szereplőiről, eseményeiről, majd ezeket a lenyomatokat tulajdonságaiknak megfelelően kezelje. Így reprezentálja személyünket egy előfizetéseket nyilvántartó rendszer a közszolgáltatóknál, egy földrajzi helyet vagy egy csővezetéket egy térinformatikai alkalmazás, de ugyanezt végzi egy játékprogram is a szabályok által alkotott keretek között létező szereplőkkel.

Ezt a kapcsolatot írja le a legelterjedtebb, objektum orientált programozási módszer, melyben a nyelv eszközkészletének részét képezi a szereplők leírása tulajdonságaik és szolgáltatásaik segítségével (osztályok), a rendszer pedig ezen osztályok példányaival (objektumokkal) dolgozik, amelyek tehát a „külvilág" szereplőit és eseményeit reprezentálják.

Tekintettel arra, hogy ezek a tulajdonság és szolgáltatás halmazok gyakran hierarchikus szerkezetűek (egy kerékpárnak és egy autónak vannak közös, „jármű" tulajdonságai és vannak egyedi, a másikra nem jellemző sajátosságai), az osztályok között tartalmazási kapcsolat: öröklődés alakítható ki (a „Jármű" osztály a „Kerékpár" és az „Autó" osztály közös őse).

Egyed és tulajdonságok

A megközelítés alapvető problémája az, hogy nem jól reprezentálja a külvilágot, összekapcsolja a megjelenített szereplő „létezését” (objektum példány) a „tulajdonságaival” (osztály). Ezzel szemben például egy ember, mint „objektum” létezése során a rá élethosszig jellemző tulajdonságokon túl átmeneti tulajdonsághalmazokkal is rendelkezik. Lesz tanuló, beosztott, vezető, telefon tulajdonos, háztulajdonos, autóvezető, … Ezek a tulajdonságok, szerepkörök adott pillanatban hozzákapcsolódnak, esetleg egy későbbi időpontban ismét megszűnnek, de amíg jelen vannak, az adott egyed „mint ilyen” szereplő elérhető, megszólítható.

Bonyolítja a helyzetet, hogy a tulajdonsághalmazok definíciói változhatnak az egyed létezése során, lehetnek olyan tulajdonságcsoportok is, amelyek definíciói a létrehozáskor még nem is léteztek, illetve idegen környezetből származnak és nem voltak ismertek.

Az elvi problémák eredménye a jelenleg létrejött heterogén, az egységes kezelés reményével sem kecsegtető helyzet, amelyben a „informatika korában” lakcímünknek nincs egyértelmű adatgazdája, számos rendszerben van nyilvántartva, és például egy költözés során minden szolgáltató, bank, újság előfizetés, … nyilvántartásának frissítéséről gondoskodnunk kell. Vagy vegyük a leghivatalosabb címnyilvántartót, a postát, amelynek rendszerében a *nekünk* küldött tartalmon a feladó a *lakcímet* tünteti fel, éppen azt az adatot, amelyet megváltoztathatunk. Ugyanígy a megváltoztatható *telefonszámunkat* adjuk meg ismerőseinknek, pedig ők nem a lecserélhető vonalat/SIM kártyát szeretnék elérni, hanem *személyünket*. Természetesen mindkét esetben van mód az átirányításra, de általában a küldő ilyenkor sem kap értesítést arról, hogy *minket* már nem *itt* lehet elérni...

A helyzet megoldása a két komponens szétválasztása. Az „egyedet” az Entitás fogalma jelöli, amely egy-egy kapcsolatban áll a külvilág szereplőjével (ember, ház, autó), amelynek egy vagy több tulajdonsághalmaza, Aspektusa van. Ezek közül van egy kiemelt „elsődleges” aspektus, amely az entitást az életciklusán át végig jellemzi, de ezen felül rendelkezhet további aspektusokkal is.

Az aspektus lényegében megfelel az objektum orientált megközelítés „objektum” fogalmának: jellemzőket és szolgáltatásokat tartalmaz és egy konkrét külvilágbeli szereplőhöz kapcsolódik, tartalmát az aspektus „Típusa” írja le, amely az „osztály” fogalmának felel meg. Az öröklődés pedig a típusok közötti kapcsolatok segítségével írható le. Ennek megfelelően mondhatjuk, hogy létezik egy „Név” aspektus, majd úgy adjuk meg a Személy és az Utca aspektusokat, hogy mindkettőnek kötelezően igényli a Név aspektus meglétét. Egy adott személy, illetve utca megadásakor létrejön az ő Név aspektusuk is, amikor név szerint keresünk, vagy egy térképen szeretnénk megjeleníteni az utcát és a személyünket, azt a közösen ismert és használható „Név” aspektus felhasználásával tehetjük meg.

Az entitást közvetlenül nem érjük el, mindig valamilyen aspektusán keresztül. Így például személyünket a Név aspektus alapján, vagy autónk tulajdonosaként, munkahelyi pozíciónk alapján, és így tovább. Különös szerepet tölt be bizonyos aspektusok esetén az egyedi azonosító tulajdonság (például egy autót az alvázszáma) vagy tulajdonságok (egy helyiséget az épület-emelet-ajtó sorozat). Az elsődleges aspektus típusok esetén ilyen egyedi azonosító megadása kötelező, ez az adat az aspektus példány teljes életén keresztül változatlan kell legyen, akár publikus, akár belső technikai azonosítóként működik.

A Személy típusnak is van ilyen azonosítója, amelyet egymással megosztunk, bár minden felületen a megszokott Név szerepel. Amikor telefonhívást bonyolítunk, a Név alapján a hívott egyedi azonosítóját küldjük el a telefonhálózatnak, amely vissza tudja keresni a kapcsolódó telefonszámot, ezzel pedig végrehajtja a hívást. Ez ma nem így működik? Dehogynem, csak a telefonszám játssza a publikus adat (név) szerepét, az egyedi azonosítás pedig az előfizető egyedi azonosítója (IMSI kódja), amelyet mi nem ismerünk, viszont a rendszer valójában ezt használja a kapcsolat létrehozására.

A telefonszám azért szükséges, mert ehhez létezik egységes, globális, hierarchikus, az egyediséget biztosító kezelési szabályrendszer, míg a személyi azonosítókra nincs ilyen szabvány. Egy ilyen rendszer létrehozásának semmilyen technikai akadálya nincs, elvégre a globális telefonrendszer, az internet domain név szabványai, vagy bármelyik email kiszolgáló ugyanezt a szolgáltatást nyújtja. Kizárólag az üzleti-politikai szemlélet, és az általa generált téves motivációk teszik lehetetlenné, sőt, személyes biztonsági szempontból kifejezetten aggályossá egy ilyen rendszer kialakítását. Ezzel viszont hihetetlen mennyiségű ismételt fejlesztést, felhasználói kényelmetlenséget, sokszoros adminisztrációt okoz – ezek költsége azonban az ügyfelekre hárítható, tehát a probléma megoldásában a felelős szereplő ellenérdekelt.

Életciklus

A külvilágban az egyedek életciklusa általában egyértelmű: az élőlények megszületnek és elpusztulnak, a tárgyakat elkészítjük, használjuk, majd végül megválunk tőlük, a földrajzi környezetünk meg „ott van”.

Ezek informatikai lenyomatának életciklusa már sokkal bonyolultabb. Jelen programozási gyakorlat szerint adatszerkezeteket, objektumokat hozunk létre a számítógép memóriájában. Ez a memória törlődik a gép kikapcsolásával, tehát valamilyen külső, tápellátástól független tárolóba kell menteni és onnan visszaolvasni az entitás adattartalmát. Ez lehet egy ismert formátumú fájl a saját gépünkön, valamilyen tároló szolgáltató (adatbázis-kezelő, LDAP szerver, de ilyen például az ismét más módon elérhető elektronikus postaládánk is). Innen töltjük fel az objektum példány adatait, jellemzően a rendszerünkre nézve egyedi programkódok segítségével (mert függenek az objektum leírásától, az aktuális fájl formátumtól, vagy a használt tároló elérő felületétől).

Mivel semmilyen szabvány nem létezik ezen művelet leírására, gyakorlatilag ahhoz szokunk hozzá, hogy minden rendszer egymástól független lenyomatait tartalmazza azonos entitásoknak, ezek minden, korábban említett mellékhatásával – éppen az számít meglepetésnek, ha egy rendszer belépési adataival egy másik („barát”) rendszer használható (OpenID, Facebook, Twitter, Gmail belépés). Szinte általános jelenség az is, hogy ezzel a felhasználó is visszaél: szerinte biztonságos álnéven jelentkezik be, vagy akár több azonosítót készít magának.

Összesítve: a jelenlegi informatikai környezet a „lenyomatok” kezelése szempontjából hasonlít arra az időre, amikor a hardver és a szoftver egymástól elválaszthatatlan volt, csak most a hardver szerepét átvették az adatainkat kezelő, egymással silány kapcsolatban lévő szoftverek. Használati szempontból semmi különbség nincs egy valóban „hardver” (papír) telefonkönyv, és egy olyan „modern” telefonkészülék között, amely csak a saját gyártó-függő formátumában hajlandó exportálni az ismerőseink listáját, szánalmasan szűkös szolgáltatáskészletet nyújt – eszközeink közötti folyamatos szinkronizációról, új kiegészítő adatok felvételéről már nem is beszélve. Mindezeknek nem technikai akadálya van, hanem az, hogy természetesen minden gyártó a saját további termékeinek értékesítésében érdekelt. A közös szabványok kidolgozása és használata kifejezett hátrányt jelentene számára. Természetesen mindez tovább gondolható az internet elvileg „szabványos” eszközeire: HTML, CSS, JavaScript „nyelvjárások”, beépülő modulok garmadája, …

Cél az, hogy az egyedek életciklusára nézve egyértelmű, közös, globális szabványokat alkossunk. Ismétlem, ennek technikai akadálya nincs, hozadéka óriási, ezért az összes megfelelően tehetős globális vállalkozás (gyárak, szállító rendszerek, nemzetközi szervezetek) kidolgozza és megvalósítja a maga módján belső rendszereiben (bejelentkezés, adathozzáférés, eszköznyilvántartás, naptárak, erőforrások, feladatok, ...). A feladat tehát megoldható, sőt megoldott, ugyanakkor természetesen nem egyszerű.

Az entitás egy külvilági „egyed” olyan informatikai lenyomata, amelyen műveletek végezhetők, tehát egy adott számítástechnikai eszköz (nem feltétlenül „gép”) operatív memóriájában van, adatai és szolgáltatásai programkódok, illetve ezeken keresztül a felhasználó számára elérhetők. Hogy kerül oda? „Előhívás” révén. Nem létezik olyan művelet, hogy én „létrehozok” egy entitást, majd

„betöltöm” az adatait, mert olyan nincs, hogy egy entitás „üres”, nincs kapcsolatban pontosan egy külvilági egyeddel. A keretrendszer feladata az, hogy az általam megadott adatok segítségével (aspektusok adatmezőire kiadott szűrés után, esetleg listából választással) megtalálja a kívánt egyed elsődleges tároló környezetét, onnan az adatait felolvassa, és a kitöltött entitást a hívónak visszaadja.

A kulcs a globális, közös típuslista az aspektusokra nézve. Ez elsőre ijesztőnek hangzik, de jelen tapasztalataim arra engednek következtetni, hogy életünk valós folyamatai, szükségletei, történései tárolásához és kezeléséhez szükséges eszközkészlet megfelelő szerkezetben kezelve meglepően egyszerű, főleg úgy, hogy a keret lehetőséget ad mind a típusok listájának, mind a meglévő típusok deklarációjának (ésszerű) módosítására.

A típusleírás ugyanis tartalmazza az aspektus példány eléréséhez szükséges információt – ez lehet egy konkrét szolgáltató gép címe, vagy egy olyan adat, amelyet elosztott kiszolgáló komponensek képesek értelmezni. Utóbbira tökéletes, és elég megbízhatóan működő példa a bármely megfelelően regisztrált gép azonosítására alkalmas domain név.

Ilyen módon egy személy egyedi azonosítóját megadva ugyanúgy, ahogy a domain név „eltalál” egy géphez, a típusleírás segítségével ez az azonosító megtalálja azt az egy vagy több kiszolgálót, amely az ő adatait tárolja, betölti, és hozzám eljuttatja.

Mindez nem jelenti azt, hogy minden adatot egyetlen rendszerben kellene tárolni, sőt. Ha valakit a lakcíme alapján szeretnék megtalálni, a lakcím aspektus adatait töltöm ki, amely a lakcím nyilvántartó adatbázisához juttat engem. Ott viszont a tulajdonos, jelen esetben egy személy egyedi azonosítója érhető el, amely alapján a személy már az előbbi úton elérhető. Ugyanígy tárolhat egy egészségügyi intézmény leletek, beavatkozások listáját, és érheti el a fenti módon a személy adatait.

A közös tárolás előnye és hátránya egyben az egyetlen egyed – egyetlen „fő” lenyomat elv: az egyidejű módosítást technikai megoldásokkal (zárolás, feloldás) ki kell zárni, az ennek ellenére (például kommunikációs hiba miatt) bekövetkező ütközéseket le kell kezelni. Ez azonban szintén nem új probléma, nem könnyű, de számos rendszerben született rá megfelelő szintű megoldás.

Mit jelent az, hogy „fő” lenyomat? Ez az az információ csomag, amely az aspektus elsődleges tároló rendszerének háttértárolóin van (tehát az ottani gépek leállítása esetén is megmarad). Az itteni kiszolgáló kérésre közvetlenül eléri és a keret megfelelő komponensei segítségével előhívja ezt az adattartalmat saját memóriájába („lenyomatot készít a fő lenyomatról”), majd továbbítja az igénylőnek. Természetesen ezen a ponton mód van tetszőleges jogosultsági ellenőrzés végrehajtására, a kérés, egyes adatok vagy szolgáltatások megtagadására.

A folyamat több pontján mód van a lenyomatok átmeneti rögzítésére, vagyis az entitás gyorsító-tárazására a szokásos szolgáltatásokkal (a jogosultságok figyelembe vételével), különösen állandó hálózati kapcsolattal nem rendelkező eszközök esetén.

A „lenyomat” sajátos példája, amikor időbeli változást, történetet rögzít – erre a típus vagy az egyed beállításainak megfelelően az elsődleges tárolón, vagy ahhoz kapcsolt naplózó eszközön van mód; ilyenkor megfelelő kéréssel korábbi állapotok is előhívhatók a rendszerből. Ez szükséges lehet például hozzáférési, jogosultsági kérdések utólagos, történeti ellenőrzése során, vagy időbeli folyamatok (forgalom, időjárás, …) elemzésénél.

Kommunikáció

Gyakorlatilag minden számítógépes program működése különböző komponensek közötti kommunikáción alapul: lényegében a kódrészletek paramétereket számítanak ki vagy állítanak össze, majd azokat műveleti paraméterek formájában átadják más komponenseknek. Ez alól nem kivétel még az egyetlen függvényből álló „spagetti” kód sem, mert az is hívja a futtató rendszer szolgáltatásait (indítási paraméterek beolvasása, képernyőre írás, memória műveletek, …).

Az objektum orientált megközelítés annyit tesz ehhez, hogy minden szolgáltatásnak ad egy aktuális adatkörnyezetet, amelyet a szolgáltatás „sajátjának” tekint, és az üzenetküldés (függvény hívás) címzettje nem a programkód, hanem egy adott objektum példány, amelynek környezetében az üzenet kezelésére megadott algoritmus végrehajtandó.

Ez a megközelítés, hasonlóan a korábbiakhoz, két súlyos kötöttséget vezet be: az adott programozási nyelv részévé teszi az üzenet küldését (a függvény neve és paraméterezése), illetve az üzenet címzettjét egyaránt forráskód szinten rögzíti; a függvényhívás azonnal gépi kód szinten vezérlésátadást jelent a megcímzett szolgáltatásnak. Ez kézenfekvő megoldásnak tűnik, azonban súlyos mellékhatásokkal jár.

Feltételezi, hogy a megcímzett objektum és azon belül a végrehajtandó kód a hívóval azonos futtató környezetben van. Ez már az alapvető operációs rendszer hívásoknál sem igaz, ahol védelmi szint módosítást jelent a vezérlés átadása, így a rendszer alkotói kénytelenek egy „kliensoldali” hívható függvény „burkot” kialakítani, amelyet a programozó meghívhat (így a rendszer hívások „normál hívásnak” látszanak), belül pedig kivételkezelési mechanizmus révén „jutnak be” a rendszer mag szintre.

Külön kategóriát jelent az, amikor az objektum nincs a hívó gépén (szerver-kliens rendszerek, elosztott szerver környezet, „felhő”). Ilyenkor az előző trükkhöz hasonló megoldáshoz, proxy objektumokhoz fordulnak. Ezek automatikusan generált forráskódú objektumok, amelyek „úgy tesznek, mintha” a megcímzett objektumok lennének, de csupán az adataik (vagy azok egy része) jelenik meg a kliens gépen, a függvényhívások pedig a kommunikációs réteg hívásaira „fordítják le” a hívó kéréseit, illetve továbbítják számára a távoli objektum válaszait.

Hosszú élettartamú rendszerek esetén ráadásul fel kell készülni a komponensek adattartalmának, szolgáltatáskészletének és azok paramétereinek változására – erre az elkészült és lefordított kódok semmilyen segítséget nem adnak, amiből számos probléma keletkezik.

Bizonyos esetekben szükség lehet a kommunikáció naplózására, monitorozására – erre a közvetlen hívások miatt szintén semmilyen alapértelmezett lehetőség nincs, helyi részleges megoldások kialakítása lehetséges, de az ide befektetett pluszmunka tovább növeli a verzió módosítások mellékhatásait (ez a réteg ugyanúgy érzékeny a szolgáltatási felület módosítására).

A címzett rögzítése megint csak súlyos következményekkel jár: a programrészlet kód szinten összekapcsolódik a partner objektumokkal, amelyekkel egyébként semmilyen más kapcsolata nincs, csak az, hogy bizonyos típusú és tartalmú üzeneteket kap tőlük vagy küld nekik. Így kötődik össze egy rendszer az adatbázis kezelővel, pedig csak annyit szeretne, hogy adatai perzisztens módon, a saját rendszer futásától független életciklussal rendelkező adatokkal dolgozhasson (de azok lehetnének szövegfájlokban, LDAP szerverben, registry-ben, vagy bárhol máshol); a helyi fájlrendszer állományaival dolgozik, pedig olvashatna FTP-ről, vagy konzolról is parancsokat; email üzenetet küld, pedig SMS vagy Skype küldés is kézenfekvő lenne, ugyanazokat a szolgáltatásokat használná.

Ennek megkerülésére léteznek szabványok és eszközök a különféle programozási környezetekben: Java nyelvben az interface („üres” szolgáltatási felület definíció) és a class (adatok és szolgáltatások) – de azért könnyű belekavarodni egy idő után a többféle absztrakt osztály és interfész lehetőségeibe. Ide tartoznak a konfiguráció alapú környezetek, ahol a használó kódban csak az absztrakt szolgáltatásra hivatkozunk (interfész), a konkrét megvalósító osztály pedig egy keretrendszer konfigurációjában kerül megadásra, és tőle kérhető el; ilyen például a Spring a Java nyelv környezetében.

A felsorolt jelenségek és megoldási trükkök arra utalnak, hogy alapvető gond van a komponensek közötti kommunikáció = függvényhívás megközelítéssel. A Dust teljes mértékben szakít ezzel a gondolattal: a komponensek egymás között kizárólag a Dust által biztosított üzenetküldés lehetőségét használhatják. Ennek alapfogalmai a „Csatorna” és az „Üzenet”.

Az Üzenet lényegében ugyanolyan entitás, mint egy külvilági egyed leírója, egy vagy több aspektussal rendelkezhet, az aspektusok típusa az egyed aspektusaihoz hasonlóan definiálható,

ugyanazzal az eszközkészlettel, és ugyanaz a komponens halmaz is kezeli őket. Ennek oka, hogy bár természetük szerint általában ideiglenesen léteznek és utaznak két egyedet leíró entitás között, bizonyos esetekben egyedként viselkedhetnek az eredeti tervektől függetlenül, például üzenet naplózás során ezek is perzisztens tárolóba menthetők, onnan visszaolvashatók, Undo/Redo funkcióknál ideiglenesen tárolhatók, stb.

A Csatorna a típuson belül deklarálható egy vagy több „belépési pont”, amelynek „belső oldalára” az adott típushoz írt feldolgozó kód kapcsolható – ez tehát hasonló egy publikus, külvilág által hívható „függvénynek”. Ugyanakkor ennek a „függvénynek” több típusú üzenet küldhető ugyanazon a csatornán keresztül, a feldolgozó kód feladata ezek elkülönítése, amennyiben több kapott típus engedélyezett. Másrészt, a Csatorna példány önálló objektum, vagyis biztosított a „hívás”, mint kontextus ismerete: a hívó csatornát nyit a hívott komponens felé, majd ezen keresztül egy vagy több üzenetet küld. Mindkét komponensnek lehetősége van adatokat tárolni a „csatornában”, így annak állapota és futási környezete (jogosultság, nyelv, forgalomszámlálás, átirányítás, …) a csatorna fennmaradásáig biztosított.

A hívó nem kód, hanem deklaráció szinten hivatkozik az általa futás közben használni kívánt csatornákra. Ennek megfelelően mód van arra, hogy az entitások külső konfigurációs eszköz segítségével kapcsolódjanak össze, illetve kapcsolatuk egyes paraméterei szintén külső adatként legyen meghatározható (bár természetesen továbbra is mód van a csatornák célpontjainak és paramétereinek kód szintű megadására). Így Dust környezetben alapértelmezett, hogy egy futtatható rendszer komponensei kizárólag az adott környezetbe történő telepítés során kapcsolódjanak össze az ott megadott módon a környezetben elérhető szolgáltató komponensek példányaiként.

Kontextusok

A fenti felsorolásból bizonyára feltűnt, de egyértelművé kell tenni: egy számítógépes rendszer elméletileg három típusú adatcsomaggal dolgozik:

- „külvilági egyed” rendszerbeli lenyomatát tartalmazó, vele műveleteket végző entitás;
- „rendszer komponens”: adatbázis kapcsolat, felhasználói felület, naplózó, üzenettovábbító rendszer elemei;
- „ideiglenes entitások”: események, üzenetek adattartalmát átmenetileg, a kommunikáció idejére tároló entitások.

A Dust környezetében ezek között semmilyen különbség nincs. Egy alkalmazás szempontjából ugyanolyan perzisztens adatobjektum egy „Személy” entitás, mint az „Adatbázis kapcsolat”, ugyanolyan eszközökkel írható le, és ugyanazok a keret komponensek kezelik mindegyiket.

A kiszolgáló környezet objektumaira viszont jellemző, hogy eltérő életciklussal rendelkeznek.

- Egyes szolgáltató objektumok csak a velük végzett művelet idejéig léteznek (például egy konkrét adatbázis lekérdezés).
- Más objektumok „statikusak”, tehát adott futtató környezetben csak egy példány létezik belőlük – ilyen például az adatbázis hozzáférést kezelő objektum.
- Megint mások köztes megosztottsággal rendelkeznek, például egy szerver-kliens rendszerben az aktuális kliens hozzáférési jogosultságait hordozó adatbázis kapcsolat, amely az első kérésnél keletkezik, és a kliens kapcsolat fennállás alatt ugyanaz a kapcsolat példány érhető el a kliens számára, amikor lekérdezéseket hoz létre.

Ezen változatok közül az „egyedi” és a „statikus” általában a nyelvi környezet által biztosítottak: alapértelmezésben az objektumok nem megosztottak, a rá való hivatkozások átadásával, tárolásával adhatók kézből kézbe; megfelelő kulcsszó használatával az objektum statikussá tehető, és azonos példány érhető el minden hivatkozó számára. Utána már bonyolultabb, környezetfüggő megoldások

következnek (adott futtató szálra egyedi „ThreadLocal” adat, függvényben statikusnak deklarált, az első híváskor értéket kapó változó, webes környezetben a kapcsolat (Session) objektumban tárolt objektum. Sokféle eszköz, azonos feladatra.

A Dust esetében nincs mód programkód szinten befolyásolni a kontextusokat, nincsenek statikus objektumok és függvények – ez a feladat a konkrét alkalmazás deklarációjának során megoldandó. A tevékenységek végrehajtása során az új objektum létrejöttét / adott kontextusban elérhető példány elérését kizárólag a konfiguráció befolyásolja.

Például: az alkalmazás komponens listájában szerepel egy adatbázis hozzáférés entitás deklaráció. Ez típusánál fogva „perzisztens tároló”, tehát kérhető tőle entitások előhívása. Tartalmaz ezen túl konkrét lekérdezés deklarációkat, amelyek adott komponenseket hívnak elő a megfelelő táblákból. A „PerzisztensTároló” típus tartalmaz egy „alapértelmezett tároló” referenciát, amely erre az entitásra mutat. Adatelőhíváskor csatornát kell nyitni a „PerzisztensTároló” felé, amelynek címzettje alapértelmezésben ez az entitás. A csatorna a hívó kontextusban elérhető felhasználó jogosultságainak megfelelő kapcsolatot nyit az adatbázisra, amely a csatorna létezéséig fennmarad, ebben egy vagy több előhíváshoz megfelelő átmeneti lekérdezés objektumok keletkeznek és futnak le.

Deklaratív eszközök

A korábbiakból kitűnhetett, hogy véleményem szerint a jelenleg programozáshoz alkalmazott eszközök (tervező eszközök, programozási nyelvek és eszközkészletek) üzleti szempontból helyes, de a valós igényekkel homlokegyenest ellenkező megközelítéseket alkalmaznak.

Igyekeznek minden lehetséges eszközzel magukhoz kötni a programozókat: folyamatosan fejlesztett eszközkészletek, állandóan bővülő – pontosabban hízó szolgáltatáskészletek, amelyek állandó tanulásra és újraimplementálásra kényszerítik, a csak ott használható tapasztalatok révén pedig „röghöz kötik” őket.

A meglévő megoldások folyamatosan elavulnak, de a környezetek állandóan bővülő piacán hordozhatatlanok is, ilyen környezetben még egy megbízható keresztplatformos megoldás sem kialakítható (ahol egy közös „fedőréteg” szolgáltatásait használva lehetne általános kódot írni, amely beilleszthető különféle futtató környezetekben létrehozott alkalmazásokba).

A létrehozott programkódok jelentős része már számtalan alkalommal megvalósított feladat újbóli lekódolását jelenti. Akár adott cégen belül is létezik már implementáció a kért szolgáltatásra, de kettővel korábbi komponenskészletre lett megvalósítva egy éve – ma már „elavult”. Gyakorlatilag nem is éri meg hosszú távra gondolkozva újrahasznosítható komponenseket fejleszteni (hiába örök probléma a tesztelés-hibajavítás-továbbfejlesztés ciklus járulékos költsége), mert mire elég jók lesznek, már ódivatúak.

Az a töredék kód, amely valóban egyedi az aktuális feladathoz, többségében olyan dolgokat tartalmaz, amelyek nem az algoritmus megvalósításai: típus definíciók, felhasználói felületek szerkezetét és eseménykapcsolatait leíró kódok, szerviz funkciók (betöltés, kiírás, naplózás, …) Mindezeket a választott nyelv, és például felhasználói felület, adatbázis-kezelő, ... eszköztár elemeit felhasználva kell megírni, bár tartalmilag semmi közük az adott környezethez, ugyanaz az adatszerkezet, ugyanaz a felhasználói felület tökéletesen megfelelő lenne egy másik nyelven, nem webes hanem asztali program számára, nem adatbázisban, hanem XML fájlban tárolt adatokra – de ez érdektelen, mert az alkalmazott környezet elvárja, hogy mindezek forráskódban (adatbázis scriptekben, HTML/JavaScript kódokban, stb...) jöjjenek létre.

Jól láthatóvá teszik a problémát a divatos szoftver tervező eszközök, amelyek „projektjei” gyakorlatilag szoftverek, bennük megtervezhetők az adat és program komponensek (jellemzők halmaza, adattípusok, ellenőrzések, kommunikáció, akár felhasználói felületek, stb.) Jellemző, hogy a drágább eszközök a célnyelvi forráskódot is képesek legenerálni, és bizonyos szintig szinkronban tartani az alkalmazás tervével (forráskódból tervet előállítani, kódból a tervet, tervből a kódot

frissíteni) – persze viszonylag szűk határok között, nem beszélve a változat követésről, esetleges párhuzamos munkából származó ütközésekről...

A Dust szakít ezzel a megközelítéssel. Alapelve, hogy amelyik programkód generálható, vagy minden gondolkodás nélkül megírható, annak nem is szabad megszületnie, mert a forrásként használt adathalmaz, például egy program adatszerkezeteinek vagy felhasználói felületeinek terve a futtató környezetben is beolvasható, felhasználható; a kapcsolódó szolgáltató objektum hierarchia (például a felhasználói felület struktúrája és eseménykezelő rutinjai) futásidőben felépíthető pontosan úgy, ahogy azt a generált forráskód tenné. Kivétel e szabály alól azon forráskódok halmaza, amely

- arra szolgál, hogy a programozó az algoritmus lekódolása közben hivatkozni tudjon a tervben szereplő adatokra, mezőkre és szolgáltatásokra – tehát konstansok listái;
- olyan adatokat tartalmaz, amelyeket a futtató rendszer betöltése során használ – tehát még nem képes külső adatforrások beolvasására (például elsődleges adatforrás egy adatbázis, de a hozzáférési beállítások nem tárolhatók az adatbázisban).

Függetlenség a környezettől

Természetesen ilyenkor mindenkinek drága tervezőeszközök hatalmas és titkos szerkezetű projekt fájljai jutnak eszébe, vagy a számtalan különféle eszköz örökké egyedi konfigurációs formátumai, vastag kézikönyvei, és a mágikus mellékhatások, alapértelmezett értékek, amik adott esetben csak fórumok bogarászásával, keret forráskód olvasással derülnek ki. Itt XML, ott JSON, máshol valami properties fájl formátum, esetleg bináris, szerkesztőprogrammal karbantartható beállítások, LDAP szerverben vagy registryben tárolt tartalom; esetleg különféle komponensek különböző megoldásokkal... Nyilván nem garantálható előre, hogy a Dust esetében minden azonnal kézenfekvő lesz, viszont néhány alapelv sokat segít a tisztább struktúrák létrehozásában.

A Dust esetében nincs köztes objektum kezelés: a külső adatforrás tartalma maga az entitás aspektus halmaza, amelyet egy általános író/olvasó komponens kezel és alakít át a rendszer által kezelhető, operatív memóriában tárolt adatszerkezetté. Ennek megfelelően nincs rögzített formátum vagy tároló, a rendszer a konfigurációjában megadott tárolóból és beállításokkal fogja előhívni a komponensek beállításait. Stream tárolás (helyi vagy hálózaton keresztül, megadott protokollon keresztül elérhető fájlok) esetén a beállításokhoz tartozik a szintaxis leírása, amely szerint az aktuális fájl olvasható (így a rendszer képes alkalmazkodni egy külső adatforráshoz, illetve rendelkezésre állnak a közismert szöveges vagy akár bináris formátumok konverterei). Bármelyik komponens konfigurációja bármilyen adatforrásból elérhető, sőt, mód nyílik több rétegű konfigurációra is: az alkalmazás alapértelmezett beállításai, fölötte olyan beállítások, amelyek a telepítéskor az adott gépre vonatkozóan felülírják az eredetieket, amit kiegészíthetnek a felhasználó egyedi módosításai.

Nincsenek kódban rögzített alapértelmezések, az adatok egyértelműen a konfiguráción keresztül állíthatók. Nincsenek rejtett, ismeretlen beállítások, ugyanis a típusok, így a bennük kezelt attribútum halmaz, azok típusai, esetleges dokumentációjuk a típus leíráson keresztül elérhető, amely ugyanolyan konfigurációs objektumhierarchia, mint a rendszer bármely más típusai.

Az adatok és módosítások helye, a változások terjedési iránya egyértelmű. Kódból soha nem keletkezik konfiguráció. A fejlesztés első lépése a típuskészlet kialakítása a megfelelő tervező eszközzel, ez a készülő komponens („Unit”) objektumhierarchiája, amit a megadott perzisztens tárolóba mentünk. Ebből generálódnak a megfelelő forrás fájlok (referencia konstansok és függvény deklarációk, amiket a programozónak le kell kódolni – ez utóbbiak lesznek a csatorna belépési pontok, az érkező üzenetek feldolgozó kódjai). A generált forrásokat a programozó nem módosíthatja, a rendszer szempontjából rögzítettnek kell tekinteni őket, frissíteni a típuskészlet módosítása után szükséges.

Ezen felül a futtató rendszer betöltő forráskódját generálja a tervező eszköz, amikor alkalmazást hozunk létre. Ide kerülnek azok a futtatható forráskódok, amelyek a Dust keret betöltése során szükséges komponenseket (memória kezelés, elsődleges perzisztens tároló, stream esetén a konfigurációs állományokban használt szintaxis) inicializálja.

Algoritmusok

A fenti eszközökkel biztosítható, hogy a komponensek, és a belőlük felépített alkalmazás adatstruktúrája, szerkezeti felépítése, beállításai és a kezelt adatok programkód írása nélkül, a keretrendszer által kezelt deklaratív eszközök segítségével üzemeltethető. A forráskód gyakorlatilag egyetlen feladata a komponensekhez érkező üzeneteket kezelő kód megírása, amelyben a beérkező adatokat feldolgozza, illetve szükség esetén más komponenseket megszólít.

Ez azt jelenti, hogy a komponensek egymást soha nem látják közvetlenül, programozási nyelv szintjén, csupán a számukra a felhasznált komponensek deklarációja alapján generált forráskódok (konstansok) kötik össze őket. Ezeket a konstansokat és a futás során előállított paramétereket használja a Dust környezetében írt kód arra, hogy a Dust szolgáltatási felületén adatokat olvasson és módosítson, illetve üzeneteket küldjön más komponenseknek.

Ez a felépítés a programozás tevékenységének túlnyomó részét a deklaratív eszközökre terheli: kényszeríti a programozót arra, hogy

- a feladatát tényleg önálló funkcionális egységekre bontsa,
- utánanézzen, hogy ezek az egységek léteznek-e már (és ha igen, azokat használja vagy szükség esetén továbbfejlessze),
- a valóban kifejlesztendő komponenseket alaposan megtervezze (hiszen csak a deklaráción keresztül éri el azokat)

Ezáltal valóban átgondolt rendszerek létrehozását indukálja, amelyekben a deklarációban testet öltő terv a lényegi érték, az üzenetkezelő kódok mindig a tervben pontosan deklarált feladatok adott környezetben (nyelven, eszközkészlettel) létrehozott megvalósításai.

Következésképpen a Dust a lehető legnagyobb szabadságot biztosítja a megvalósítás környezetének megválasztásához: amennyiben a Dust kernel egy futtató környezetben létrehozható, illetve a Dust API (amely nagyjából tíz függvényt jelent) az adott nyelven megvalósításra kerül, onnantól a Dust használható.

Ettől függetlenül természetesen a Dust konkrét megvalósításához szükséges egy programozási nyelv kiválasztása, ez a (nem objektum orientált) C nyelv. A döntés oka az, hogy C fordító szinte minden környezetben elérhető a minimális képességű beágyazott rendszerektől a legnagyobb gépekig. A Dust komponens-orientált környezetet biztosít, így szükségtelenné teszi az objektum orientáció nyelvi eszközeit, azok csak felesleges akadályt képeznek; komponensekbe csomagolja azokat a szolgáltatásokat, amelyek ma a legtöbb nyelv eszközkészletét jelentik (felhasználói felületek kezelése, szokásos fájl, konzol, képernyő, memória műveletek, stb.) Így az algoritmus megvalósításához lényegében egy „ismert script-jellegű szintaxis” a legmegfelelőbb - a pedig C egy minimális nyelv, a lehető legszűkebb eszközkészlettel – ideális választás.

A Dust megközelítésmódja még egy szokatlan lehetőséget rejt. Minden futtatható programkód gyakorlatilag művelet objektumok halmaza a memóriában, amelyek a rendszer szolgáltatásait (memória hozzáférés, vezérlés átadás, aritmetikai és logikai műveletek) használják. Maga a forráskód ennek egy megadott nyelven (jelen esetben a C programozási nyelven) fájlba mentett változata, amelyet egy fordítóprogram először művelet hierarchiává konvertál, majd (optimalizálás után) a gépen futtatható bináris kóddá alakít. Erre a lehetőségre épülnek természetesen a szintaxis szerinti színező komponensek, a forráskódok tartalmi elemzését végző, átrendezését lehetővé tévő fejlesztőkörnyezetek is.

A forráskódhoz tehát létezik és biztosan generálható vele végrehajtás során ekvivalens (bár várhatóan a fordítási optimalizáció miatt lassabb), művelet objektumokat tartalmazó hierarchia. Ugyanígy tehát bármi, ami C programozási eszközökkel, a Dust komponenseinek használatával megvalósítható, az ilyen végrehajtó komponens entitásokat tartalmazó hierarchia segítségével, tehát deklarációs eszközökkel is megvalósítható.

Nyilván nem arról van szó, hogy ez a lehetőség kiváltaná az algoritmusok forrásnyelven történő programozását – viszont alkalmas

- a forráskódok vizualizációjára, így tartalmi ellenőrzésére;
- egyszerű alapértelmezett, fejlesztés alatti átmeneti algoritmusok kialakítására programozás nélkül (figyelmeztető jelzések, naplózás, ideiglenes adatgenerálás, stb);
- egyszerű ellenőrzések megadására (adatbevitel formátum, értékkészlet, adatmezők egyszerű függésvizsgálata), amelyek például távoli kliensen, közvetlenül a felhasználói felületen, fordítás nélkül is futtathatók;
- C forráskód interpretálására (fordított felhasználás: egy olyan környezetben, ahol a C fordító nem elérhető, mégis lehetséges a C forrás „lefuttatása” azáltal, hogy a kód végrehajtó entitások hierarchiájává alakítható);
- arra, hogy a rendszer maga alakítson ki műveleteket, „programozza saját magát”: például adaptálódjon a rajta áthaladó forgalomhoz, algoritmusokat hozzon létre tanuló algoritmusok végeredményének felhasználásával, stb.

Futtató környezet

A programozó a feladata megoldása során algoritmusokat fejleszt ki, amelyek az aktuális nyelv és eszközkészlet komponenseit felhasználva valósítanak meg részfeladatokat, talán már ezredszer, majd ezekből a komponensekből a kért környezetben elérhető alkalmazást épít fel. A cél természetesen a minél rövidebb fejlesztési idő, ezért azok a környezetek értékesebbek, amelyek több részfeladatot vesznek le a programozó válláról.

Ez lehet a Java megközelítése, amely a több operációs rendszeren megvalósított Java futtató környezet révén homogén felületet teremt a jellemző programozási feladatok számára – így viszont egy egyszerű összeadáshoz is kell a sok tíz MB méretű teljes JRE. Ennek elkerülésére változatokat hoztak létre belőle (J2ME, J2SE, J2EE), amitől ezek rögtön különböző futtató környezetek lettek: a J2ME nem biztos, hogy minden komponenst tartalmaz, amelyet a J2SE-ben megírt Java programom igényel. Továbbá így sem tartalmaz mindent, tehát kellenek kiegészítő komponensek (hálózati kapcsolatok, adatbázis, levelezés, stb.), illetve több verzió van (1.3.1, 1.4, 5, 6, 7) amelyek lényegesen eltérnek egymástól.

Lehetséges egy szűk nyelv (pl. C++) használata, de ez nem elég hatékony, számtalan kiegészítő komponens könyvtárat érdemes használni; itt egyes cégek megpróbálják lefedni az általuk megcélzott teljes szegmenst (például: a Borland adatbázis-kezelő csomag a tároló szerver elérésétől az objektum kiolvasáson keresztül a felhasználói felület kontrollokig lefedi az igényeket), ebből is van számos változat – ezeket öröm használni, amíg minden igényünket kielégítik, de bajba kerülünk, ha egy ideális felhasználható komponens nem az általunk választott környezetben készült.

A programfejlesztés jelen pillanatban egyre növekvő sebességgel kering egy ördögi körben. A fejlesztő-futtató környezetek igyekeznek minél naprakészebbek és teljesebben lenni, ezért folyamatosan változnak. A változások miatt a megírt kódok elavulnak, használhatatlanná válnak, újra kell írni őket. Az újraírás időigényét minimálisra kell csökkenteni – ezért hatékonyabbá kell tenni a környezetet, vagyis tovább kell frissíteni, okosítani azt.

A történet további kellemetlen része, hogy a futtató környezetek általában meghatározzák az alkalmazás jellegét: gépre telepített asztali alkalmazás vagy webes vékonyklienses; adatbázissal

vagy adatfájlokkal dolgozik, Windows vagy Linux környezetben, esetleg beágyazott rendszerben kell futnia (telefon, háztartási eszköz, szórakoztató elektronika). Ezek mind meghatározzák az alkalmazható nyelvek körét, a futtató környezet kód által elérhető része sem szabványos. Vagyis: ugyanazokat az ezerszer megoldott feladatokat újra és újra le kell kódolni az aktuális futtató rendszerhez adaptálva.

A Dust ebből a spirálból az ellenkező irányban tör ki. Minimalizálja a megírandó kód mennyiségét, a létrehozott kódok kizárólag deklaráció szinten kötődnek egymáshoz, az alkalmazás valójában csak a futtató gépen áll össze az ott rendelkezésre álló komponensekből, de ettől függetlenül az elvárásoknak megfelelően képes működni. A futtató kernel csak a valóban elengedhetetlen minimális komponenseket tartalmazza, így a mérete bármilyen környezetre adaptálhatóvá teheti. Az interpreter lehetőséget ad olyan nyelven megírt forráskód futtatására, amelyhez az adott rendszerben nem is érhető el fordító (bár természetesen ez csak áthidaló megoldás lehet).

A rendszer kritikus része természetesen az a szegmens, amelyet az operációs rendszer tesz a futó program alá: a gép aktuális hardver eszközei: memória, háttértárak, perifériák (billentyűzet, egér, monitor, hang, …) A rögzített perifériákkal valójában meg is köti a számítógép lehetőségeit: az USB portok segítségével ma már számos monitor, egér és billentyűzet köthető fizikailag egy számítógéphez, amely a monitorok kezelésére alkalmas is – mégsem lehet több munkaállomásként használni egy gépet, mert a perifériák nem oszthatók meg munkafolyamatok között.

A Dust egyszerűségénél fogva erre is lehetőséget teremt. Egy egér gyakorlatilag egy esemény forrás komponens, amely az egér hardver eseményeire figyel, és továbbítja azokat. Természetesen van mód több egér, billentyűzet bekötésére, amelyek helyzet illetve karakter eseményeket generálnak; mód van így bonyolultabb eszközök integrálására (például két kamerával vagy speciális eszközzel 3D helyzet információt közlő „egér”; a néző pozíciójának ismeretében 3D „monitor” integrációjára). Mindezek a meghajtó kernel komponensek operációs rendszer szintjén vagy közvetlenül a hardverre kötve is megvalósíthatók, a többi alrendszer (például a felhasználói felület kezelő komponens) ezek esemény csatornáit „hallgatják”, következésképpen operációs rendszertől függetlenek, vagy akár a megfelelő meghajtók megléte esetén operációs rendszer nélkül, közvetlenül a hardveren futtathatók. Ez gyakorlatilag minden Dust alapon megvalósított komponensere igaz, ahol a futtató rendszeren biztosítható minden igényelt komponens megléte (legalább emulátor szinten).

Felhasználói felület

A felhasználói felület a programozás állatorvosi lova: minden korábban említett kóros jelenség tünetét magán hordozza.

Először is, a fejlesztés elején el kell dönteni, hogy asztali vagy webes alkalmazást fejlesztünk – bár a komponensek túlnyomó többsége számára nincs jelentősége, bármelyik környezetben azonos szolgáltatásokat kell nyújtaniuk.

Ez a döntés meghatározza, hogy milyen eszközökkel (nyelvek, eszköztárak) dolgozunk. Az asztali megvalósítás általában operációs rendszer és GUI eszköztár választást is jelent, amelytől már csak a rendszer jelenős újrafejlesztésével lehet elszakadni; webes környezetben szerveroldali eszköztárak választása jelent hasonló, gyakorlatilag végleges kötöttséget.

A felhasználói felület szerkezete túlnyomó részben forráskódok formájában alakítható ki: az eszköztár objektumaiból összeállított hierarchia az, amit aztán a futtató környezet megjelenít a felületen, ezek felelősek az események kezeléséért és az adattartalom megjelenítéséért. Asztali esetben ez a programozási nyelv objektumait jelenti, webes környezetben HTML / CSS nyelv az alap, de a szerveroldali komponensek tovább bonyolíthatják a történetet: JSP/ASP lapok, az eszköztárak saját konfigurációs eszközei, esetleg kódgenerálással kiegészítve. Önálló osztályt képez a GWT, amelyet Java nyelven lehet programozni, azonban ezt HTML/JavaScript forrásra fordítja le az eszköztár.

A programozási alapelvek között fontos helyen szerepel a megjelenítés és az üzleti logika (ellenőrzések, műveletek) szétválasztása – viszont rettentő nehéz elkerülni, hogy az egyébként is forráskód formájában létrehozott felhasználói felület kódjába mégis bekerüljön valami ilyen tevékenység. Külön említendő, hogy az üzleti logika megvalósításánál nincs is helye, „belépési pontja" annak az eseménynek, hogy valamilyen adat megváltozott, aminek hatására ellenőrzést kell futtatni; nincs egyértelmű „parancs felület" (a felhasználói felület gombjai) – ezek az esemény kapcsolatok kizárólag a felhasználói felület forráskódjában, eseménykezelők formájában valósíthatók meg.

Az a tény, hogy ma az alkalmazások webes felhasználói felületet kapnak, önmagában teljes abszurditás. Tény, hogy a különféle operációs rendszerek és futtató környezetek gyártói ellenérdekeltek voltak abban, hogy egyetlen közös szabványt hozzanak létre arra a teljesen kézenfekvő feladatra, hogy egy alkalmazás felhasználói felülete az üzleti logikától elválasztva, akár másik gépen kell megjelenjen. Ezért azt a megoldást választották, hogy egy eredetileg statikus tartalom megjelenítésre kialakított szabványt, a HTML nyelvet bővítették egyre extrémebb lehetőségekkel, mellé tették a CSS és a JavaScript különféle változatait, amelyeket minden böngésző kicsit másként implementál. Mindez lehetőséget ad a lehető legkuszább, ám működő felhasználói felületek kialakítására, gyakorlatilag lehetetlenné teszi a homogén, megbízható megjelenést, és elvi képtelenséggé a tartós, hosszú távon működőképes komponensek kialakítását. Utóbbiakkal értelme sincs kísérletezni, az egyetlen értelmes választás valamilyen „nagy" cég vagy fejlesztőcsoport remélhetőleg megbízható, eléggé böngészőfüggetlen és elfogadható sebességgel frissített komponenscsomagjához kötődni, azokat használni.

A Dust számára mindez a totális képtelenség nem létezik. A felhasználói felület természetesen nem létezik forráskód formájában, kizárólag a megfelelő típusú entitások hierarchiája alkotja. Ezek az alkalmazáshoz kapcsolódó beállítások között, perzisztens tárolóban érhetők el, betöltéskor entitás példányok keletkeznek belőlük, amelyet az aktuális megjelenítő környezet (ez nyilván a futtató hardver, operációs rendszer vagy eszközkészlet) kontroll komponensei valósítanak meg. Az ő feladatuk a lehető legjobban alkalmazkodni a konfigurációban leírtakhoz, kezelni a felhasználói eseményeket (egér, billentyűzet). A rajtuk keletkező események (adatmódosítás, akció) konfigurációnak megfelelően kötődik a megjelenített entitás példányok adatmezőihez és parancsaihoz.

Így Dust alatt a felület nem kötődik a megjelenítéshez használt eszközkészlethez, független az üzleti logikától akár azonos, akár távoli gépen fut. A megközelítés lehetővé teszi különleges megjelenítő környezetekben használni ugyanazt a GUI deklarációt. Lehetséges például egy szöveges konzolon leírni a szerkezetét, lehetőséget adni a panelek hierarchiájában történő navigációra, adatok módosítására, parancsok kiadására. Ugyanígy mód van a szerkezet felolvasására, hangvezérléssel az eredetivel egyenrangú kezelőfelületet biztosítani a gyengén látók számára. Lehetséges a különféle felhasználói felületek térbeli megjelenítése és a velük való interakció: 3D szemüveggel az asztalra „vetíteni" az alkalmazás kezelőfelületét, kamerával követni a kézmozdulatokat, így a virtuális gombok megnyomhatók lesznek. Mindez nem igényel semmilyen egyedi fejlesztést, pusztán annak következménye, hogy a felület deklaratív eszközökkel került kialakításra.

További két fontos tulajdonság: a GUI deklaráció nem tartalmaz megjeleníthető szövegeket, csak hivatkozásokat – a szövegek különálló tárolása révén az alkalmazás lokalizálása, más nyelvi változatának elkészítése analóg az első változat elkészítésével, csak ezt a különálló nyelvi objektumtárat kell a célnyelvre lefordítani.

A Dust a felhasználói felület deklarációjára és megjelenítésére ugyanazokat az eszközöket használja, mint amikkel a statikus formázott szövegeket kezeli – a kettő között technikai különbség nincs, csak a GUI esetén aktív kontrollok is elhelyezhetők a szövegben. Így egy aktív felhasználói felület az aktuális adataival ugyanúgy elmenthető, kinyomtatható, mint egy formázott dokumentum.

Küldetés

A Dust keretrendszer létrehozásának kettős célja van. Egyrészt, amennyiben sikerül működőképessé tenni, egyértelmű bizonyítékot szolgáltat arra, hogy a jelen hiedelem, amely szerint a piaci, pénzügyi versenyhelyzet optimális megoldások kialakulását motiválja, több mint tévedés: szándékos csúsztatás. Valójában azt garantálja, hogy bármilyen „szolgáltatást nyújtó szereplő" a saját profitját igyekszik maximalizálni minden eszközzel, így nincs tekintettel a környezeti hatásokra, a költségeket pedig a lehető legtágabb módon externalizálja. Tökéletesen ellenérdekelt egy megoldható probléma megoldásában, mert az saját piacának megszűnését jelentené. Továbbá soha nem érdekelt a termékeinek egyenlő elérhetőségének biztosításában, hanem éppen ellenkezőleg: fizetőképesség szerint szelektál, és csak a tehetősek érdekeit veszi figyelembe – következésképpen az általa birtokolt és használt tudományos és technológiai vívmányokat természetszerűleg az emberi társadalom vagyoni helyzetbeli széttagoltságának növelésére használja.

Mindez a jelenkori globális gazdasági-pénzügyi szemlélet természetes és elidegeníthetetlen következménye. Az állítás indoklása nem célja ennek a dokumentumnak, de elérhető a Hajnalvilág elemzésben. Következésképpen a jelen sokrétű (emberi, kulturális, társadalmi, ökológiai, erőforrás, gazdasági és politikai) válságainak megoldása reménytelen fajunk együttműködési rendszerének újraszervezése nélkül, amelynek nem képezi részét a „pénz" fogalma.

A pénz központi szerepet játszott az emberi civilizáció hatalmas sebességű fejlődésében, azonban mára elfelejtettük, hogy valójában csupán eszköz arra, hogy valós erőforrásainkat (természeti kincsek, a felhasználásukhoz szükséges technológia, infrastruktúra, szaktudás és emberi munkaerő) képesek legyünk egységes rendszerben kezelni, tartalékolni, átcsoportosítani, nagyobb közösségeket érintő célok érdekében mozgósítani. Ennek mind a mai napig a leghatékonyabb eszköze az érték absztrakció, amely az egyedi csere ügyletek helyett egy közvetítő eszköz, a pénz felhalmozására és fókuszált felhasználására ad módot.

Ma azonban van egy ennél hatékonyabb eszköz a kezünkben, ez pedig a globális, és gyakorlati szempontból egyidejűnek tekinthető informatikai hálózat. Ez lehetőséget ad nekünk arra, hogy az absztrakció jelenlegi kényszerén (ahol mindent, az emberi életet, egészséget, szabadságot és jövőt is homogén módon, ráfordítható pénzösszeg formájában kezelünk) túllépve újra a valósággal legyünk képesek foglalkozni.

Végső soron mindannyiunknak vannak jogos fizikai létszükségletei, amelyek kielégítése nem csak a bolygó szerencsés felén élő szerencsés csoportoknak elidegeníthetetlen emberi joga; és mindannyiunk természetes igénye fenntartó és fenntartható életmódot folytató közösségek aktív, megbecsült tagjaként létezni véges életünk során. Ehhez valójában az szükséges, hogy pontosan képesek legyünk látni a rendszerben elérhető erőforrásokat és igényeket, teljesen átlátható felületen legyünk képesek közös célokat kialakítani, azok között fontossági sorrendet felállítani, az elérésükhöz szükséges feladathierarchiát megtervezni, az elérhető erőforrásokat hozzárendelni, és végrehajtani őket. Ez alapvetően egy informatikai feladat, amin csak felesleges, gátló, hibás motivációkat beépítő teher a pénz, közgazdaságtan és politika ezernyi eszköze és szakértője. Technológiánk és tudásunk pedig, fajunk történelmében első alkalommal ma rendelkezik is azzal a képességgel, hogy ezt a lépést globális méretben megtegye, és ezt az informatikai rendszert kialakítva új alapokra helyezze önmaga megszervezését.

A Dust által létrehozható informatikai hálózat megbízható, és alapelvei révén gyakorlatilag bárki számára elérhetővé tehető, egyedi lenyomatait képes hordozni minden rendelkezésünkre álló erőforrásnak és igénynek, beleértve saját személyünket, szaktudásunkat, foglaltsági táblázatunkat és céljainkat. Egy működő Dust alapú rendszer az előrelépés lehetősége fajunk számára a jelenlegi eszközeinkkel és gondolkodásmódunkkal megoldhatatlan válságból.