Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Ferencz Endre

Bozóki Szilárd

AUTOMATIZÁLT TESZTADATBÁZIS-GENERÁLÁS JPA ALAPOKON

KONZULENSEK

Budai Péter

Dr. Goldschmidt Balázs

BUDAPEST, 2013

Tartalomjegyzék

Összefoglaló

A sokrétű felhasználói igényeket kielégítő szolgáltatásokat megvalósító informatikai rendszerek az összetett követelmények és integrációs lehetőségek száma miatt komplex, függőségekkel teli architektúrával rendelkeznek. Ilyen körülmények között mind a hagyományos, mind pedig az agilis szoftverfejlesztési módszertanok nagy hangsúlyt fektetnek a tesztelésre. Az on-line tartalmat előállító szerveroldali alkalmazások jelentős része valamilyen relációs adatbázist használ, ennek ellenére az objektum-relációs leképzésre alapuló adatintenzív alkalmazások működésének tesztelésére csak korlátozott eszközök állnak rendelkezésre, melyek nem képesek hatékonyan kiszolgálni a felmerült igényeket.

A fejlesztés és a tesztelés során használt adatok általában nagyban különböznek az éles rendszerben használtaktól, mivel sokszor adatvédelmi okokból nem is lehetséges az éles rendszer adatainak tesztjellegű használata. Másfelől - a korlátozott erőforrások miatt - értelemszerűen nem megvalósítható az összes lehetséges bemenet előállítása. Gondos tervezéssel előállítható olyan adatstruktúra, amely képes lefedni a szoftver használata során előforduló helyzeteket. A tesztadat-generálás problémájának azonban csak egy részlete a tényleges adatbázismezők kitöltése, mivel az egyes objektumok közötti kapcsolatok kialakítása nagyban befolyásolhatja egy alkalmazás működését.

A Java platform szabványosított objektum-relációs leképzését, a Java Persistence API-t használva mutatunk be platformfüggetlen megoldást a problémára. A fejlesztés korai fázisában megtervezett entitás osztályok alapján lehetővé tesszük a tesztadatok generálását és az ezek közötti kapcsolatok olyan formában történő kialakítását, hogy az így előálló adatkészlet minél hatékonyabban le tudja fedni a rendszer működését, beleértve mind az üzemszerű, mind a kivételes helyzeteket. Nagy hangsúlyt fektettünk az adatok minőségére. Biztosított az egyes adattípusok értéktartományának és eloszlásának finomhangolása. Az általunk javasolt eszköz hasznos lehet nemcsak a tesztelési, hanem a fejlesztési feladatok során is, ezáltal rövidítve a fejlesztési időt és növelve az elkészült termék minőségét.

Abstract

Current complex information technology systems satisfy a wide range of user requirements, while being full of interdependencies because of the integration possibilities. In such circumstances both the conventional and agile software development methodologies have a strong emphasis on testing. Most of the server side applications, which provide on-line content for their users, are based on a relational database, but the tools for testing of an object-relationship mapping based application are very limited and cannot satisfy the current needs.

Data used during testing and development might be significantly different from those used in live systems. Because of security and privacy reasons in most cases it is not possible to use samples from live systems. With careful planning it is possible to provide a data structure that is appropriate to cover most of the use cases. Completing the data fields however is only a small part of test data generation, the inter-object relationships are much harder to maintain, while they can even alter the program behaviour.

Using the Java Persistence API, the standardized object-relationship mapping on the Java platform, we present a database provider independent solution to this problem. The entity classes are designed in the early phases of a development project. Based on these classes we can generate sample data that contain suitable relationships to cover all the possible normal and exceptional situations. We placed a large emphasis on data quality, there are possibilities to configure data range and distribution. The tool presented is helpful not only for testing, but for development tasks too, enabling us to minimize the effort and to develop a product with good quality.

1 Bevezetés

Az informatika fejlődése és egyre nagyobb térnyerése révén az elérhető szolgáltatások és alkalmazások száma folyamatosan meredeken emelkedik. A hangsúly ráadásul egyre inkább az olyan termékeken van, amelyeket elosztottan, egyszerre több párhuzamos munkameneten keresztül lehet kezelni. Ebből kifolyólag folyamatosan növekszik az olyan alkalmazások száma is, amelyek adatbázist használnak adattárolás céljából.

Manapság két meghatározó technológia segítségével fedjük le a megnövekedett igényeket: objektumorientált elven építjük fel szoftvereinket, az adattárolásra pedig relációs adatbázisokat alkalmazunk. A két technológia közötti hatékony átjárást biztosítja az objektum-relációs leképzés (ORM).

A szoftverfejlesztésben egyre jobban kiéleződött a verseny, amiben óriási szerepet játszik a minőség és a költség. Ezen tényezők javítását megcélozva az egyes szoftverfejlesztési módszertanok nagy hangsúlyt fektetnek a hatékony tesztelési stratégiára.

A relációs adatbázison végzett műveletek tesztelését számos eszköz támogatja, ha alacsony szinten szükséges megoldani a problémát, de ha objektum-relációs leképzésen keresztül szeretnénk tesztelni, akkor már sokkal szerényebb a választék.

Az adatbázisban való adattárolást alkalmazó szoftverek esetén összetett feladat a megfelelő tesztadatok előállítása. További nehézséget jelent, hogy ha relációs adatbázist használunk, mivel ekkor a sémákból és a referenciális integritásból adódóan további megkötéseket is figyelembe kell venni. Néha az alkalmazáslogika is kihat a tárolt adatokra és további kényszereket hoz létre azokon. Ezen felül költséghatékonysági okokból nyilvánvalóan nem lehet az összes lehetséges adatot generálni. Ki kell választani azokat a releváns eseteket, amelyekre az alkalmazás építkezik, de nem szabad figyelmen kívül hagyni az olyan speciális helyzeteket, amelyek problémát okozhatnak az alkalmazás működésében. További fontos kérdés, hogy a tesztadatok létrehozását milyen mértékben lehet automatizálni a hatékonyság szem előtt tartásával.

Dolgozatunkban az automatizált tesztadat-generálás problematikájára mutatunk be – az objektum-relációs leképzés egyik megvalósítását szem előtt tartva – platformfüggetlen megoldást.

A megoldás kidolgozásához először megvizsgáltuk, hogy egy ilyen eszköz hogyan tudna hozzájárulni a leggyakrabban alkalmazott szoftverfejlesztési módszertanok hatékony kivitelezéséhez, majd meghatároztuk azokat az elsődleges és másodlagos célokat, amelyeket az eszköznek teljesítenie kell.

Felkutattuk a jelenleg elérhető megoldások alkalmazhatóságát az általunk felvetett problémára és elemeztük ezek előnyeit és hátrányait, hogy az implementáció során fel tudjuk használni ezeket a tapasztalatokat.

Végül megterveztünk és elkészítettünk egy olyan eszközt, amely teljes mértékben képes kiszolgálni az általunk felvetett igényeket és ellenőriztük a működését egy közepes komplexitású bemeneten.

2 Tesztadat-generálás

A tesztadat-generálás már a fejlesztési folyamat során is hasznos lehet. Megfelelő minőségű adatokkal történő fejlesztés esetén a hiba már nagyvalószínűséggel annak keletkezésekor felfedezhető, annak költsége minimalizálható. A jól definiált metodológiák szerint azonban sokkal átfogóbb tesztelésre van szükség.

Ebből következően napjainkban a tesztelés egyre hangsúlyosabb szerepet kap a szoftverfejlesztésben. Ebben a fejezetben arra a kérdésre keressük a választ, hogy egy új eszköz hogyan és milyen mértékben tudna ahhoz hozzájárulni, hogy a tesztelés még hatékonyabb, gyorsabb és pontosabb lehessen.

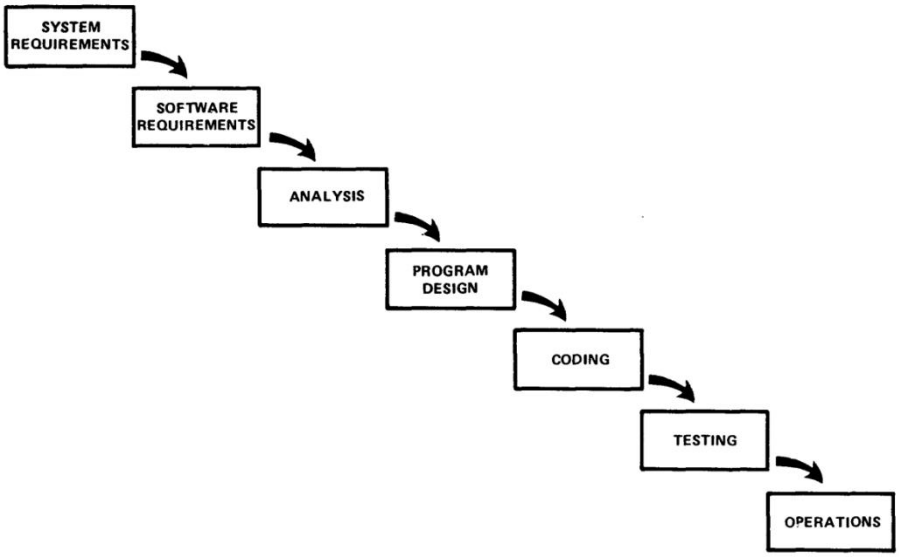
2.1 Tesztelés helye a szoftverfejlesztésben

A szoftvertesztelés egy olyan tevékenység, melynek célja, hogy valamilyen aspektusból releváns információt szerezzen egy adott szoftver vagy termék minőségéről. Erre a tevékenységre általában többször és több célból is szükség van egy projekt során. Mint azt ebben a szakaszban bemutatjuk, egyes módszertanok a gyakori tesztelést javasolják, míg más esetekben csak az átadás előtt kerül sor a végleges termék ellenőrzésére.

2.1.1 Hagyományos megközelítés

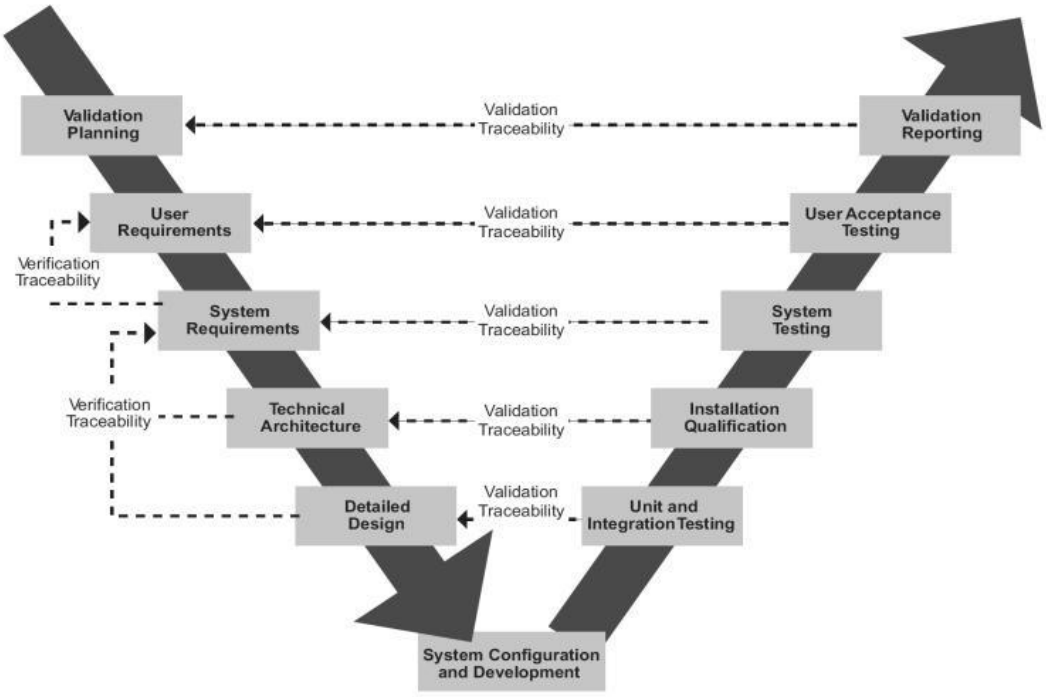
A szoftverfejlesztési módszertanok hagyományos (életciklus alapú) megközelítésében jól meghatározott fázisokra bomlik a fejlesztés folyamata. A konkrét fázisok és az azok közötti összefüggések már az egyes jól definiált módszertanoktól függ. A beható vizsgálathoz ezek közül hármat vizsgálunk meg: a vízesés modellt, a V-modellt és a Rational Unified Process (RUP) modellt.

A vízesés modell esetében külön fázis (1. ábra) van elkülönítve a tesztelés számára (1). A lefejlesztett programot ekkor kell behatóan vizsgálni, hogy megfeleljen a specifikációban foglaltaknak. Az ekkor lefolytatandó teljes körű vizsgálat során értelemszerűen szükség van olyan eszközökre, amelyek az egyes tesztesetekhez bemeneti adatot állítanak elő.



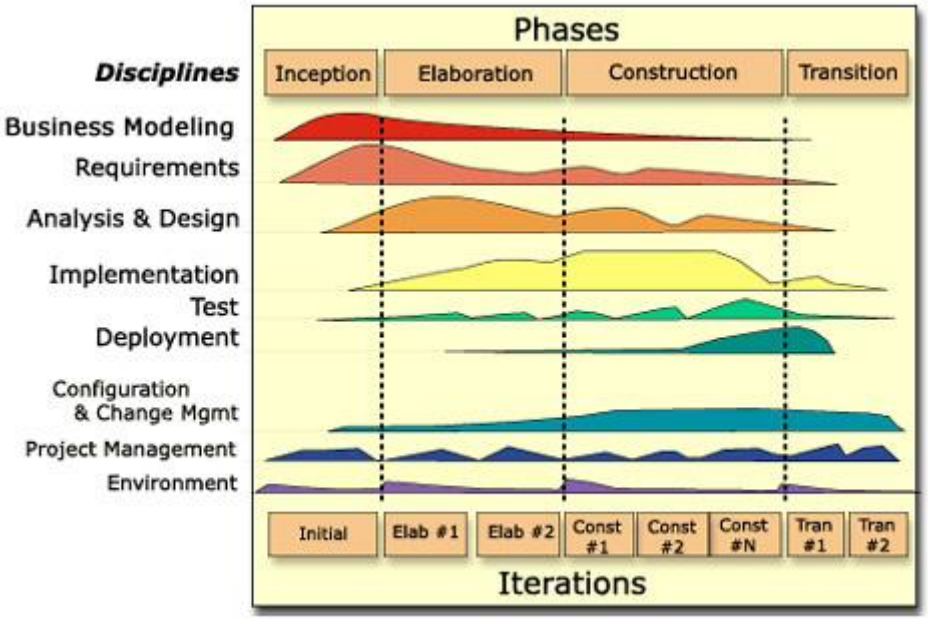
1. ábra Vízesés modell fázisai (1)

A V-modell (2) már nem elégszik meg az egyidejű átfogó teszteléssel, hanem minden egyes fázis végén az elkészült terméknek megfelelő teszteket kell futtatni (2. ábra). Véleményünk szerint az egységteszteléstől a felhasználói elfogadási tesztekig minden fázisban szükség lehet egy jó minőségű adatokkal feltöltött adatbázisra.



2. ábra A V-Model fázisai (2)

Az IBM által javasolt RUP iteratív modell (3) esetében minden iterációra meghatározott jellegű és minőségű tesztelés szükséges (3. ábra). A tesztadat-generálás az elaborációs fázistól válik szükségessé.



3. ábra RUP fázisai (3)

2.1.2 Agilis megközelítés

Az agilis módszertanok szerint a szoftverfejlesztés olyan iteratív és inkrementális folyamat, melynek során a követelmények és az erre adott megoldás önszerveződő és többfunkciós csapatok együttműködése során jön létre. Az agilis módszertanok esetében hangsúlyos szerepet kap a tesztelés.

Az agilis módszertanok közül a teszt-orientált fejlesztési módszertant (Test-driven development) emeljük ki. Ennek lényege, hogy a fejlesztők először megalkotják a kezdetben sikertelenül lefutó automatizált teszteseteket és csak ez után foglalkoznak a szoftvertermék kidolgozásával (4). Ebben az esetben a tesztadat-generálás fokozottan megjelenik, mivel a követelmények átfogó elemzése után már könnyen definiálhatóak az adatbázis entitások, amelyekre az adatgenerálást építeni lehet.

2.2 A szoftvertesztelés típusai

Láthatjuk, hogy a tesztelést a tárgyalt módszertanok kiemelten kezelik. Fontos megvizsgálni azonban, hogy a lehetséges teszttípusok közül melyek azok, amelyeket automatizált teszteléssel támogatni lehet.

2.2.1 Statikus és dinamikus tesztelés

A tesztelt komponens vizsgálata szempontjából megkülönböztetjük a Statikus és a Dinamikus tesztelést (5).

Statikus tesztelés esetén a komponenst nem futtatjuk, csak a forráskódot és a hozzátartozó egyéb leírókat vizsgáljuk a specifikáció, a követelmények és egyéb definiált szempontok, metrikák szerint. Egy tipikus statikus tesztelési technika a kódszemle, melynek során a forráskódot végignézik és ellenőrzik a résztvevők.

Dinamikus tesztelésről akkor beszélünk, mikor futás közben figyeljük meg az ellenőrzött komponenst.

Ezt a két teszttípust elkülönítve jól látható, hogy az általunk tervezendő eszközt csak dinamikus tesztek esetén lehet hatékonyan felhasználni.

2.2.2 Doboz megközelítés

A tesztelt komponensről rendelkezésre álló információk alapján megkülönböztetjük a fekete dobozos (funkcionális), fehér dobozos (strukturális) és a szürke (áttetsző) dobozos tesztelést.

Fekete dobozos tesztelés (6) esetén a tesztelt egység belső működéséről semmilyen információval nem rendelkezünk, kizárólag a ki- és bemeneti interfészeivel állunk kapcsolatban. Fekete dobozos tesztelés során tehát azt vizsgáljuk, hogy az adott bemenetekre előállított kimenet mennyiben egyezik az elvártakkal. A bemenetek és az elvárt kimenetek előállításánál a rendszer belső működéséről nem rendelkezünk információval, csak az interfészeket és a specifikációt ismerjük.

Fehér dobozos tesztelés (7) esetén a tesztelt egység teljes belső felépítésével tisztában vagyunk és azt a tesztelés során is képesek vagyunk megfigyelni. A dinamikus fehér dobozos tesztelés egyik technikája a kód lefedettségének mérése, amiben azt mérjük, hogy a tesztelés során beadott bementek feldolgozása közben az adott program melyik sora, utasítása vett rész.

Szürke doboz tesztelés esetén a Fekete és a Fehér dobozos megközelítés egyes elemei egyszerre kerülnek alkalmazásra. Ebben a megközelítésben a tesztelő ismeri a tesztelt egység bizonyos belső részeit, amit a célzott tesztesetek kidolgozásához felhasznál. A teszt lefuttatása azonban fekete doboz módjára történik, ugyanis

semmilyen belső információt nem tartunk nyílván a tesztelt egységről, csak a kimeneteket vizsgáljuk.

Tipikus példája egy szürke dobozos megközelítésnek egy adatbázist használó többrétegű alkalmazás tesztelése, amelyben az adatbázisban található adatokat a rendszer implementációjának függvényében beállítja a tesztelő, majd ez után a felhasználói felületen megvizsgálja, hogy a megfelelő módon kezeli-e a tesztelt egység az adatbázis adatait. Ez a példa a tisztán fekete doboz megközelítéstől ott tér el élesen, hogy az adatbázis ismerete és beállítása szükséges. A fehér dobozos teszteléstől ott tér el, hogy a futás közben már nem férünk hozzá sem a tesztelt egység belsejéhez, sem az adatbázishoz.

Az automatizált tesztadat-generálás tehát a szürke vagy a fehér dobozos tesztelést képes hatékonyan kiszolgálni.

2.2.3 Fejlesztési folyamat által meghatározott tesztelési szintek

A V-modell tárgyalásánál már említettük, hogy a tesztelés különböző szintjein egyaránt felhasználható a tesztadat-generálás. Ebben az alfejezetben azt mutatjuk be, hogy az egyes szinteken milyen módon lehet és érdemes ezt felhasználni.

Az egységtesztek esetén általában egy osztály funkcionalitását ellenőrizzük. Ezen a szinten leginkább az összetett üzleti logika futtatása során jelentkezik az összetett bemeneti adatok megadásának problematikája. Ilyen esetekben azonban roppant hasznos lehet egy olyan eszköz, amely a megadott paraméterekkel determinisztikusan képes létrehozni a bemeneteket.

Az integrációs teszt már összetett szoftverkomponenseket vizsgál összefüggéseiben. Egy ilyen szcenárió esetében kritikus, hogy az egyes komponensek viselkedése minél inkább megegyezzen az éles környezetével. Ebben lehet nagy segítségünkre egy olyan eszköz, amely a valódi adatokkal megegyező minőségű adatbázist képes előállítani.

A rendszerteszt során már teljesen összeállnak a rendszer egyes komponensei és ebben az esetben is érvényesek az integrációs teszteknél megfogalmazott állítások.

Az elfogadási teszteknél is kritikus lehet a megfelelő tesztadat, mivel sok esetben ekkor még nem állnak elő vagy nem elérhetőek egy éles rendszer által használt adatok.

2.3 Eszközök automatizált teszteléshez

A legtöbb gyakorlati teszteléssel foglalkozó szakirodalom (8) tényként kezeli, hogy a tesztek akkor tudnak igazán hasznosak lenni egy szoftverfejlesztési projektben, ha azok képesek rövid idő alatt automatikusan lefutni és értékelhető eredményt visszaadni.

A Gartner definíciója szerint (9) az automatizált tesztelés jelző illik minden olyan szoftverre vagy szolgáltatásra, ami támogatja a felügyelet nélküli tesztelési folyamatot, beleértve a funkcionális és a terheléses teszteket is. Az előnyök többek között a következőek: konzisztens eredmények és adatok, könnyű karbantarthatóság és hatékonyabb erőforrás felhasználás.

Összességében elmondható, hogy az automatizált tesztelés során a különböző szoftverek és eszközök a megfelelő beállítások után emberi beavatkozás nélkül működhetnek.

Az automatizált tesztelés során alkalmazható eszközök közül részletesebben az automatizált tesztadat-generálást támogatóakat fogjuk megvizsgálni.

2.4 Az automatizált tesztadat-generálást támogató eszközök

A tesztadat-generálás iránti igény már régóta jelen van a piacon. Láthattuk, hogy a szoftverfejlesztési módszertanok többsége nagy hangsúlyt fektet a tesztelésre, így természetes, hogy sok különböző termék érhető el változatos képességekkel.

Ebben az alfejezetben ezeknek az eszközöknek a képességeit mérjük fel, kiemelve az olyan funkciókat, amelyek relevánsak lehetnek az általunk megtervezett eszköz szempontjából is.

2.4.1 Generate Data

A Generate Data[1] egy online, főleg webes környezetre fókuszáló, alap szintű tesztadat-generátor. Ingyenes változata korlátozott számú sor generálását engedi.

A létrehozás során előre definiált alaptípusok közül lehet választani, melyeket esetenként további paraméterek beállításával testre lehet szabni. Ezen felül valódi

[1] http://www.generatedata.com/

mintákkal is rendelkezik, mint például beépített típusok esetén az országok nevei vagy a személynevek.

A kimeneti formátumokat tekintve támogatja a gyakorlatban használt legfőbb fájltípusokat (pl. XML, CSV, Excel, HTML, JSON, stb.)

Előnye, hogy használata nagyon egyszerű, de csak addig lehet jól használni, amíg csak a beépített típusokra van szükségünk. A mi szempontunkból további nagy előnye, hogy a támogatott programozási nyelvek esetén forráskódot is képes generálni JavaScript, Perl, PHP és Ruby nyelveken. Az előállított forráskód egyszerű tömböket vagy listákat tartalmaz.

A Generate Data hátránya, hogy az adattáblák közötti kapcsolatokat nem támogatja és az egyedi igényeket csak korlátozottan képes kiszolgálni.

2.4.2 Databasetestdata

A Databasetestdata[2] egy online alapszintű ingyenes adatgenerátor. Beépített típusokkal és esetenként hozzájuk tartozó valós mintákkal rendelkezik. A típusokat nem lehet tovább paraméterezni.

A weboldal háromféle kimeneti formátumot támogat (JSON, CSV, XML). Előnye, hogy egyszerűen lehet használni, viszont nagyon kezdetleges.

2.4.3 Mockaroo

A Mockaroo[3] is egy online alapszintű tesztadat-generátor. Változatos és helyenként paraméterezhető típuskészlettel rendelkezik, amelyhez megfelelő valódi minták is tartoznak.

Kimeneti formátumait tekintve kiemeljük, hogy az előző rendszerek képességeit felülmúlja az SQL kódgeneráló képességével. További előnye, hogy választékos és alapszinten paraméterezhető adattípusokkal rendelkezik.

A Mockaroo weboldalán egyébként a szerzők részletesen kifejtik (10) a megfelelő tesztadatok alkalmazásának fontosságát, mellyel mi is egyetértünk.

---

[2] http://www.databasetestdata.com/

[3] http://www.mockaroo.com/

2.4.4 Spawner

A Spawner[4] egy egyszerű offline használható adatgenerátor. Jelenleg hét adattípust támogat, de ezekhez összetett paraméterezhetőség tartozik.

Az alkalmazás képes feltölteni egy beállított MySql adatbázist a generált adatokkal. Fontos előnye, hogy nem webes alapokon készült, ezért lényegesen gyorsabb és hatékonyabb, mint az online adatgenerátorok.

2.4.5 Red Gate SQL Data Generator

A Red Gate SQL Data Generator[5] egy olyan alkalmazás, amely relációs adatbázisokat céloz meg. Rengeteg funkcióval rendelkezik, az eszköz kipróbálása során ezeknek csak egy részét sikerült teljes körűen felmérnünk.

Az adattípusokat tekintve választékos beépített típusokkal rendelkezik, és ehhez összetett paraméterezhetőség is társul.

A beállított adatbázissal már elinduláskor kapcsolatba lép, a generálás előtt feltérképezi a lehetőségeket, így az ott lévő adatokat is fel tudja használni a generáláshoz. Ezen felül a sémákban leírt oszlopnév és oszloptípus alapján intelligensen felismeri, hogy milyen típusú adatot célszerű az adott táblába generálni. Ezen felül képes a generált adatokat az adatbázisban meglévő referenciális integritásnak megfelelően előállítani.

A Red Gate SQL Data Generator professzionális eszköz, támogat minden olyan funkciót, amely natív SQL nyelven megvalósítható.

2.4.6 GS Data Generator

A GS Data Generator[6] szintén egy nagyvállalati professzionális tesztadat-generátor.

A Red Gate megoldásához hasonlóan nagyon széles a támogatott típusok és az állítható paraméterek skálája. A GS Data Generator is rendelkezik intelligens

---

[4] http://sourceforge.net/projects/spawner/

[5] http://www.red-gate.com/products/sql-development/sql-data-generator/

[6] http://www.gsapps.com/products/datagenerator/

típusfelismeréssel, amik az adatbázis sémában leírt oszlopnevek és típusok alapján kiválasztja a megfelelő generálandó típust és a hozzá tartozó beállításokat. A referenciális integritás megtartására szintén lehetőség van.

2.4.7 Upscene Productions Advanced Data Generator

Az Upscene Productions Advanced Data Generator[7] is egy professzionális adatgenerátor, így rengeteg beállítással, beépített típussal és testre szabható paraméterrel rendelkezik.

Hátránya a Red Gate és a GS Data megoldásához képest, hogy nincs benne intelligens típusazonosítás. További eltérés, hogy a generált sorok között jobban paraméterezhető egyedi összefüggéseket lehet definiálni.

2.5 Összegzés

A megvizsgált eszközök képességeiről készítettünk egy összefoglaló táblázatot (1. táblázat).

Eszköz	Típuskészlet	Függőségek	Minta valós adatokról
Generate Data	alap	nem támogatja	támogatja
Databasetestdata	alap	nem támogatja	támogatja
Mockaroo	alap	nem támogatja	támogatja
Spawner	alap	nem támogatja	nem támogatja
Red Gate SQL Data Generator	átlagos	támogatja	támogatja
GS Data Generator	komplex	támogatja	támogatja

[7] http://www.upscene.com/products.adg.index.php

Eszköz	Típuskészlet	Függőségek	Minta valós adatokról
Upscene Productions Advanced Data Generator	komplex	támogatja	támogatja

1. táblázat Megvizsgált eszközök

Számos, jelenleg elérhető tesztadat generáló szoftver megvizsgálása után arra a következtetésre jutottunk, hogy a funkciók nagy része három kategóriába sorolható be: alapszintű adatgenerálás, entitások közötti függőségek kezelése és mintaadatok felhasználása.

2.5.1 Alapszintű adatgenerálás

Az alapszintű adatgeneráló funkció típushelyes adatokat képes létrehozni. A különböző rendszerekben elérhető típusok nagy száma miatt sokféle beállítható paraméter létezik, mint például az értékkészlet egész számokra, véletlen szám generálás esetén annak eloszlása vagy reguláris kifejezések karaktersorozatokra.

A piacon elérhető összes adatgeneráló szoftver tartalmazza ezt az alapfunkciót, de fontos hangsúlyozni, hogy a generált adatok minősége nagymértékben változó.

Minőségi tesztadatnak tekintjük az olyan adatsort, amely tulajdonságaiban a lehető legjobban hasonlít az éles rendszerben alkalmazott adatokhoz és képes lefedni az olyan speciális helyzeteket, melyeket az alkalmazott megszorítások nem zárnak ki.

Teljes körű tesztelés során fontos, hogy figyelembe vegyük az olyan értékeket is, amelyek kívül eshetnek a program értelmes bemeneti tartományán, mégis valamilyen használati eset során tárolásra kerülhet (robosztusság tesztelése). Speciális robosztusság teszt az egyes kitöltetlen (null) értékek beállítása az olyan mezőkre, amelyekre ez lehetséges.

Az előforduló adattípusok értékhatáraira is külön érdemes figyelni, mivel az egyes algoritmusok érzékenyek lehetnek a speciális bemenetekre (határérték tesztelés).

A fejlettebb eszközök képesek lehetnek ekvivalencia osztály alapú tesztelésre is, melynek segítségével kevés számú tesztesettel is lefedhetőek a program főbb végrehajtási útvonalai. Ehhez azonban összetett paraméterezés is szükséges, tehát hangsúlyos az emberi tényező.

A generálás során érdemes lehet külön figyelmet fordítani a speciális karakterek jelenlétére, mert ezek is nagyban módosíthatják a program lefutását. Különböző biztonsági hiányosságok (11) például éppen a speciális karakterek helytelen kezelése miatt lépnek fel (pl. SQL injection).

2.5.2 Entitások közötti függőségek kezelése

Az entitások közötti függőségek kezelése már összetettebb funkció. Relációs adatbázisoknál alapesetben ez a referenciális integritás megtartását jelenti, azaz olyan értékek generálása, amely megfelel az idegen kulcsokkal szemben támasztott követelményeknek.

Ezek az eszközök csak arra törekednek, hogy helyes kimenetet generáljanak, de nem dolgozzák fel szemantikailag az egyes osztályok közötti kapcsolatokat. Általában még összetett konfigurációval sem elérhetőek olyan megvalósítások, amelynek során figyelembe lehetne venni az objektumszintű függőségeket (pl. kör vagy fastruktúra generálása).

A piacon elérhető adatgeneráló szoftverek jelentős része tartalmaz alapszintű függőségkezelést. Általában paraméterként adhatóak meg a függőségek és fontos szerepet kapnak itt is a manuális beállítások.

2.5.3 Minta valós adatokról és mezőazonosítás

A minta valós adatokról funkció egy beépített adatbázis meglétét jelenti, ami tartalmaz mintaadatokat gyakran előforduló fogalmakról, mint például helységnevek, ország jelzések, férfi és női nevek. Tipikusan ez a felhasználói adatok tárolásánál hasznos, mivel az emberek számára könnyen értelmezhető adatok könnyíthetik a fejlesztést és a tesztelést. Ezek mellett a beépített mintaadatok minőségileg jobban hasonlítanak a valós környezetben alkalmazottakhoz, így hatékonyabb lehet a tesztelés.

A mezőazonosítás egy olyan kiegészítő funkció, amely egy adott típusú és elnevezésű attribútumhoz ajánl egyet a beépített mintaadatokból, amennyiben rendelkezésre áll ilyen.

Ezt a funkciót kielégítően általában csak a magasabb árkategóriájú szoftverek tartalmazzák.

3 Tervezés során alkalmazott célok és követelmények

A szakirodalom áttekintése és a jelenleg elérhető megoldások vizsgálata alapján megfogalmaztuk azokat a célokat és követelményeket, amelyek nagyban elősegítenék a megtervezett eszköz felhasználhatóságát a fejlesztés és a tesztelés során.

A különböző programozási platformok közül a Java nyelvet céloztuk meg, mivel úgy találtuk, hogy ezen a nyelven még nem érhető el olyan eszköz, amely hatékonyan tudná elősegíteni a tesztadatok megalkotását.

Az általunk megfogalmazott legfontosabb cél, hogy a Java Persistence API (JPA) annotációkkal megfogalmazott adatmodelleket minél hatékonyabban tudjuk feldolgozni és értelmezni. A JPA platform- és adatbázis-független jellege miatt az elkészült eszköz bármilyen Java nyelvből kezelhető adatbázissal képes együttműködni.

További lényeges célkitűzés, hogy a rendszert bővíteni tudjuk a későbbiekben is különböző működési stratégiákkal, melyeknek minél magasabb szintű támogatást szeretnénk nyújtani az osztályok és objektumok elemzéséhez.

A fejlesztés kutatási jellege miatt nem fordítottunk kiemelten hangsúlyt a könnyű használhatóságra, így első körben csak az automatikus működést és a könnyű paraméterezhetőséget céloztuk meg és nem tervezünk grafikus felületet készíteni az alkalmazás számára.

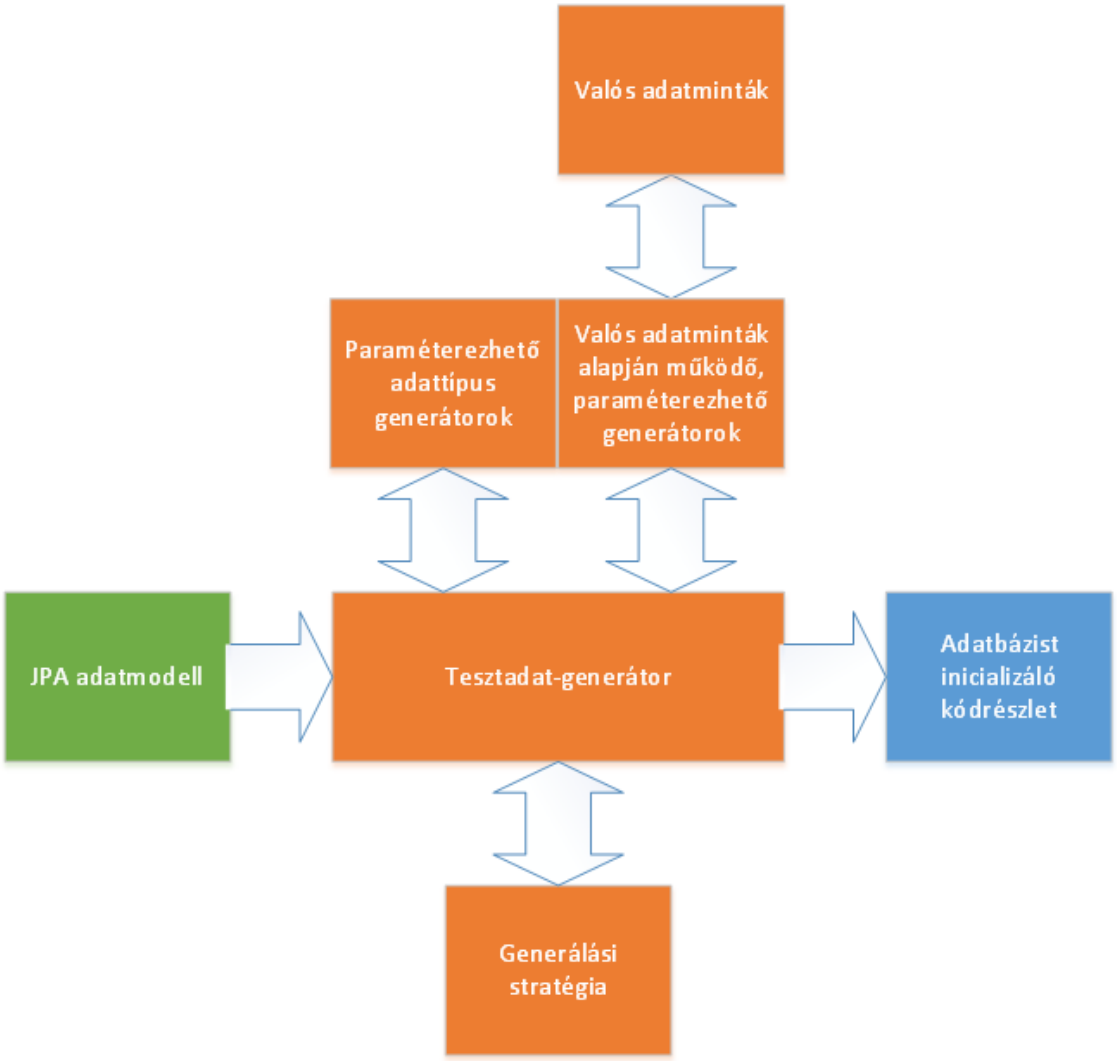
Ugyanígy másodlagos célnak tekintettük a ritkán használt adattípusok támogatását és a kiterjedt mintaadatok integrációját.

3.1 Az eszköz működése

Az általunk elképzelt eszköz tehát egy adatbázismodellt dolgoz fel bemenetként. Az egyes adattípusoknak megfelelő generátorokat a program beépítetten tartalmazza és lehetőség van valós adatminták alapján is a mezők feltöltésére.

A jelenleg elérhető eszközökhöz képest egy kiegészítő funkciót építünk a programba, melynek segítségével lehetővé válik az egyes entitások közötti kapcsolatok értelmezése és elemzése. Az elemzést két szinten is támogatni fogjuk: az entitás osztályok szintjén és a létrehozott példányok szintjén. Ezáltal lehetővé válik speciális feltételek teljesítése is, mint például a stratégiában meghatározott objektumstruktúrák létrehozása.

A mellékelt illusztráción (4. ábra) összefüggéseiben láthatóak a tesztadat-generátor komponensei (narancssárga színnel), annak bemenete (zöld színnel) és kimenete (kék színnel).



4. ábra Az eszköz működése

4 Adatbázismodell elemzése

Az automatizált tesztgenerálás egyik kritikus pontja a bemenetnek tekintett modell feldolgozása és elemzése. A működés szempontjából nem csak a modell osztályainak a felderítése szükséges, hanem az egyes adattagok és kapcsolatok megismerése is lényeges.

Java platformon három szinten lehetséges a feldolgozás: forráskód, byte kód vagy futtatás közben. A forráskód értelmezéséhez összetett szövegfeldolgozás szükséges és nagyon mélyen kell ismerni a Java nyelv lehetséges szintaktikai elemeit. A byte kód esetében már standardizált formában érhetőek el a szükséges információk, de az értelmezés összetett és a specifikáció eltér az egyes verziók esetében. A legkézenfekvőbb megoldás futás közben elemezni az osztályokat. Ehhez szükséges az osztályok betöltése, majd a Java Reflection API használata.

4.1 A Java reflexió használata

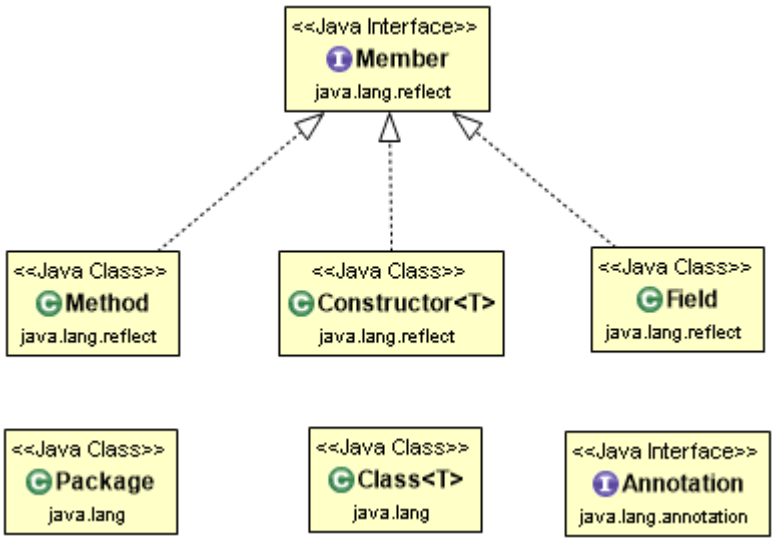
A reflexió egy program azon képességére utal, hogy futási időben lehetséges egy objektumnak, vagy magának a program belső struktúrájának és működésének vizsgálata. A reflexiót elsősorban a magas szintű, menedzselt környezetet alkalmazó programozási nyelvek használják.

A szakirodalomban elsősorban három motivációját szokták kiemelni a Java reflexió használatának (12):

- Szükség van fordítás után csatolható komponensekre (plug-in), amelyek támogatása nem lehetséges egy egyszerű interfészmegvalósítás segítségével. Erre a legjobb példa a Java Enterprise Edition szabvány által bemutatott alkalmazáscsomagok (war, ear, jar).
- Távoli eljáráshívás során megjelenő (fordítás időben nem ismert) osztályok felderítése és példányosítása.
- Kiegészítő biztonsági korlátozások kikényszerítése, amelyeket a Java nyelv beépítetten nem támogat. A leggyakrabban megjelenő gyakorlati alkalmazása ennek a területnek a futtatásra nem engedélyezett csomagok kizárása.

Mindhárom esetben a reflexiót alkalmazó program olyan helyzetre készül fel, amelynek bizonyos aspektusait a program írásakor még nem ismerhetjük. Ehhez az

osztályokról készült belső reprezentációt kell megismernünk, amely metaadatok szintjén lekérdezhető a Java nyelv alkalmazott metamodellje szerint (13). Az 5. ábrán láthatóak a Java reflexióban alkalmazott legfontosabb osztályok.



5. ábra A Java nyelv metamodellje (az egyszerűség kedvéért a kapcsolatok közül csak a leszármaztatás van feltüntetve)

Fontos kiemelni, hogy reflexió segítségével az osztályok olyan tagjai is elérhetőek, amelyek amúgy a láthatóság jelzés alapján rejtettek lennének. Ez a viselkedés a biztonsági beállítások segítségével (SecurityManager) letiltható.

4.2 Annotációk használata

A Java nyelv 1.5 verziótól támogatja az annotációk használatát, amelyek segítségével a program írójának lehetősége van olyan metainformációkkal kiegészíteni a forráskódot, amelyek szorosan kapcsolódnak a környező részletekhez, de mégsem foglalhatók bele a működő kódba. Az alap Java csomagok hét beépített ilyen annotációt tartalmaznak. Ezek közül négyet új annotációk írása során lehet felhasználni, míg három a kód helyességét segíti elő biztonsági ellenőrzések bevezetésével (Deprecated, Override, SuppressWarnings).

Az annotációk írása során alapesetben négy aspektust lehet állítani:

- Az adott annotáció milyen fázisokban legyen elérhető (Retention). Lehetséges értékei szerint fordítási időben, lefordított byte kódban vagy futtatás során olvasható az annotáció.
- Az annotáció megjelenjen-e a dokumentációban (Documented).

- Az annotáció milyen elemekre használható (Target). Lehetséges értékei: annotáció, konstruktor, attribútum, lokális változó, metódus, paraméter vagy típus.
- Öröklés esetén a leszármazottakra való érvényesség (Inherited).

Ahhoz, hogy egy annotációt futási időben fel lehessen dolgozni szükséges az elérhetőség helyes beállítása (RetentionPolicy.Runtime).

Az egyes annotációk paraméterekkel is rendelkezhetnek, melyek értéke az annotáció feldolgozásakor lekérdezhető.

4.3 A Java Persistence API annotációi

Az alábbiakban az egyik elsődleges JPA annotáció látható. A @Entity típusokra használható és mivel az entitás osztályok futási időben lesznek a JPA implementáció által feldolgozva, ezért az összes annotáció futási időben elérhető.

```
@Documented
@Target(TYPE)
@Retention(RUNTIME)
public @interface  Entity {
     String  name() default "";
}
```

A modell elemzése szempontjából leglényegesebb annotációkat a következő táblázat foglalja össze:

Annotáció	Alkalmazható	Magyarázat
@Entity	Típusra	Segítségével az egyes entitás osztályokat jelöljük meg.
@Id	Metódus, Attribútum	Az adott osztály kulcs attribútumát jelöli.
@GeneratedValue	Metódus, Attribútum	Az adott attribútum adatbázisba mentéskor automatikusan kitöltésre kerül.
@OneToOne	Metódus, Attribútum	Egy-egy kapcsolat jelölése.
@ManyToOne	Metódus, Attribútum	Több-egy kapcsolat jelölése.

Annotáció	Alkalmazható	Magyarázat
@OneToMany	Metódus, Attribútum	Egy-több kapcsolat jelölése.

2. táblázat Adatbázismodell elemzése során vizsgált annotációk

4.4 Összegzés

A Java Persistence API annotációi segítségével definiált adatmodell feldolgozására a Java reflexiót találtuk a legalkalmasabbnak. Ebből következően a bemenetnek tekintett entitásmodellt a Java nyelv metamodellje szerint tudjuk kinyerni.

Ez a metamodell azonban nem az ilyen jellegű adatmodellek leírására lett megtervezve, így célszerű lehet ezt olyan formába hozni, amely jobban megfelel az általunk fejlesztett eszköz igényeinek.

5 Automatizált tesztadatbázis-generálás JPA alapokon

Az eddigiekben felvetett problémára megoldásként egy olyan eszközt készítettünk, amely bemenetként egy Java Persistence API szabványnak megfelelő lefordított és becsomagolt entitásmodellt kap, majd a beállított generálási stratégia mentén forráskód formájában előállít egy olyan Java osztályt, amely képes felölteni egy adatbázist a megfelelő objektumokkal, attribútumokkal és kapcsolatokkal.

A megtervezett eszköz könnyen és gyorsan bővíthető saját, egyedi stratégiával, amely képes kiszolgálni az adott környezetből érkező igényeket. Egy stratégia megírásához az eszköz rendelkezésre bocsát minden olyan információt, amely lényeges lehet a bemenetként megadott modellről.

5.1 A generálási folyamat magas szintű áttekintése

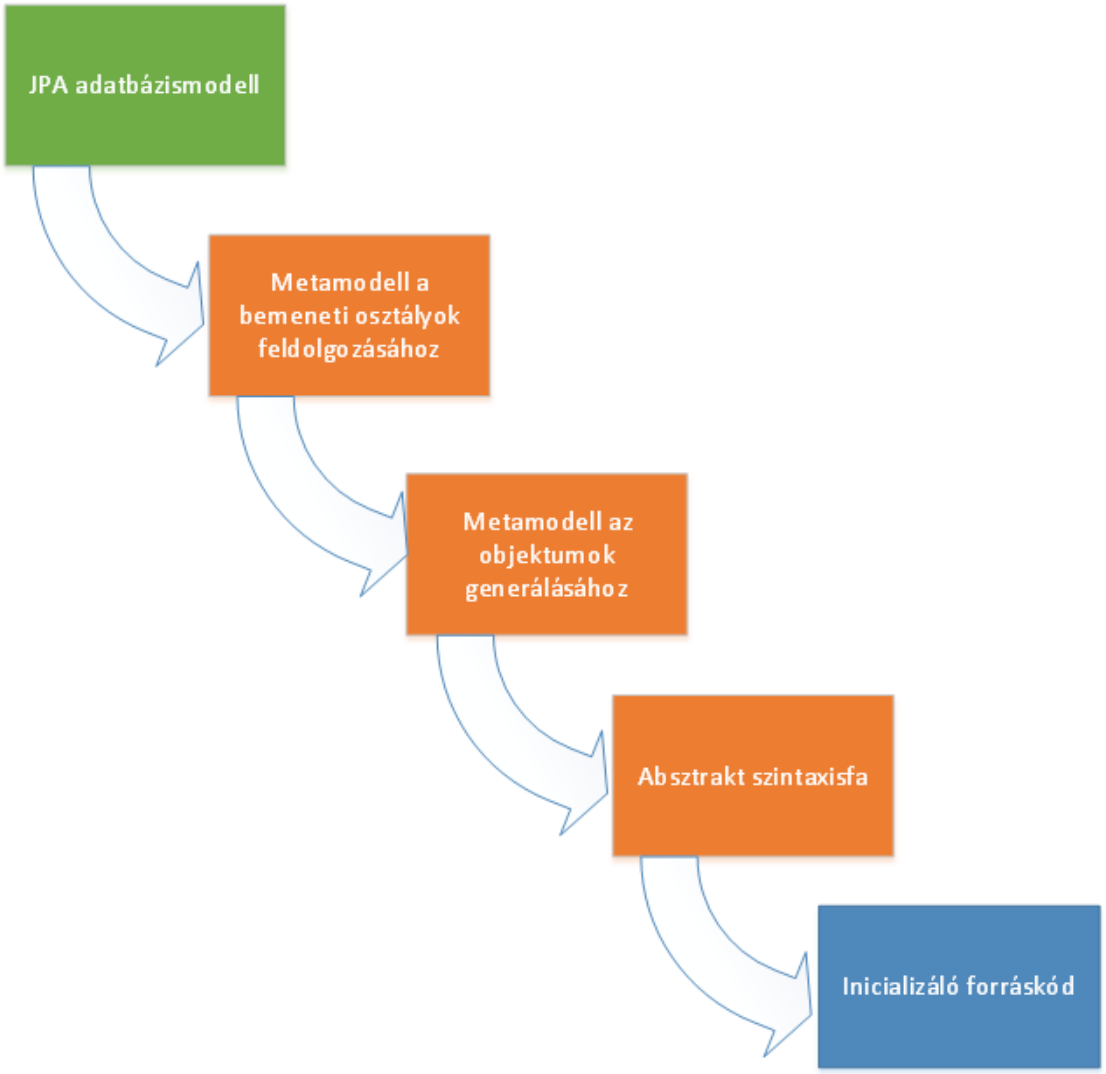
A bemenet egy sor transzformációs folyamat után (6. ábra) jut el arra a szintre, hogy futtatható Java kód álljon elő. Ennek első lépéseként szükséges az entitásmodell Java nyelven történő implementálása. Az implementáció alapján könnyen készíthető osztálykönyvtár, amelyet már képes feldolgozni az általunk javasolt eszköz.

Az osztálykönyvtár feldolgozása során az eszköz a célra kialakított metamodell alapján a Java Reflection API segítségével előállítja a modell egy olyan formáját, amelyet már sokkal kényelmesebben tudunk feldolgozni és átalakítani. A metamodell legfőbb célja, hogy képes legyen hatékonyan megragadni a különböző JPA osztályokat, azok attribútumait és a közöttük fennálló lehetséges kapcsolatokat.

Ezen a ponton lép be először a generálási stratégia, melynek feladata, hogy egy másik metamodell szerint modellezze a végső termékben előálló példányokat. Ezek attribútumai és metódusai még nincsenek kitöltve, csak egy vázat képeznek a végső objektumoknak.

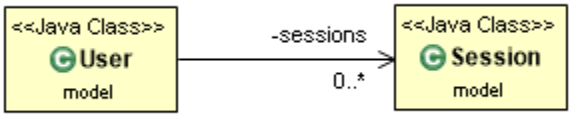
Végül a generálási stratégia alapján előállíthatóak a kívánt attribútumok és kapcsolatok, mivel ezen a ponton mind osztályszinten, mind példányszinten lehetőség van vizsgálatokat végezni, így teljes körű információval rendelkezünk a bemenetként megkapott modellről és a generált objektumokról.

A felépült modell alapján absztrakt szintaxisfa (AST) épül, amely alapján generálható a fordítható és később futtatható Java kód. A futtatás eredményeként inicializálásra kerül a relációs adatbázis a generált mezőértékekkel.

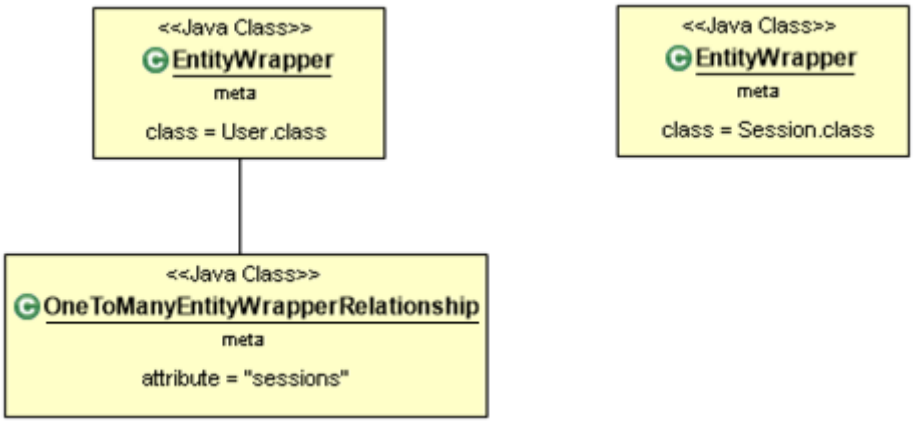


6. ábra Generálási folyamat áttekintése

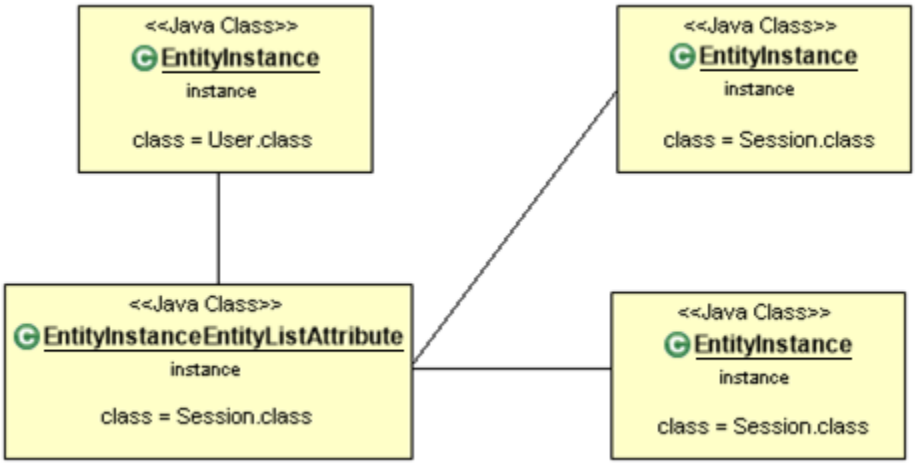
A generálási folyamat első lépéseként a bemeneti modellt (példa: 7. ábra) dolgozzuk fel, melynek eredménye a feldolgozott osztályok egy belső reprezentációja (példa: 8. ábra). A belső reprezentáció alapján jönnek létre logikai szinten a példányok (példa: 9. ábra), amely alapján felépíthető az absztrakt szintaxisfa.



7. ábra Bemeneti modell



8. ábra Bementi osztályok feldolgozása



9. ábra Objektumok létrehozása

Erre az egyszerű példára a futtatás eredménye a következő Java osztály, melynek a *generate()* metódusa létrehoz két felhasználót és három munkamenetet, majd egy bizonyos logika mentén ezeket összeköti.

```
package generated;

import model.Session;
import model.User;
import javax.persistence.EntityManager;

public class Generated {
	EntityManager em = null;

	public void generate() {
		User user0 = new User();
		User user1 = new User();
		Session session2 = new Session();
		Session session3 = new Session();
		Session session4 = new Session();
		user0.setId(01);
```

```
            user0.setName("Endre");
            user0.getSessions().add(session4);
            user0.getSessions().add(session2);
            user1.setId(1l);
            user1.setName("Szilárd");
            user1.getSessions().add(session2);
            user1.getSessions().add(session3);
            session2.setId(2l);
            session3.setId(3l);
            session4.setId(4l);
            em.persist(user0);
            em.persist(user1);
            em.persist(session2);
            em.persist(session3);
            em.persist(session4);
        }
}
```

A generált forráskódhoz megjegyeznénk, hogy egy munkafolyamatot (*session2*) két felhasználóhoz is hozzárendelt az adott generálási logika. Ez annak köszönhető, hogy ez a megszorítás nincs modellszinten kikényszerítve (ld. 7. ábra). Erre a problémára az lenne a megoldás, hogy a *Session* osztályból kiindulva fogalmazzuk meg a kapcsolatot.

5.2 JPA adatmodell tervezése

Az üzleti folyamatok elemzése során előállnak azok a főbb entitásosztályok (entity classes), amelyek lefedik azokat az üzleti igényeket, amelyeket valamilyen számítógépes eszközzel szeretnénk támogatni (14). Általában ezek alapján könnyen készíthető egy olyan modell, amely objektum relációs leképzéssel (Object Relationship Mapping) relációs adatbázisban perzisztálható.

5.3 A Java Peristence API szabvány

A Java nyelven leggyakrabban alkalmazott JPA osztályok a kapcsolódó szabvány szerint (15) a következő feltételeknek kell megfeleljenek:

- annotációval (@Entity) vagy telepítésleíróban jelölni kell, hogy az osztály egy JPA entitás,
- léteznie kell az alapértelmezett konstruktornak public vagy protected láthatósággal,
- nem lehet beágyazott osztály, enum vagy interfész,
- nem lehet letiltani az entitás osztályokból való öröklést final kulcsszóval,

- érték szerinti paraméterátadás esetén szükséges a Serializable interfész megvalósítása,
- szükséges az elsődleges kulcs definiálása.

A JPA egyik alternatívája a Java Data Objects (JDO) specifikáció, amely rugalmasabb az entitásosztályokat tekintve, de kevés nyílt forráskódú implementáció érhető el hozzá.

Az entitások közötti kapcsolatokat tekintve lehetséges az egy-egy (OneToOne), egy-több (OneToMany), több-egy (ManyToOne) és a több-több (ManyToMany) kapcsolat is. Az egyes kapcsolatok lehetnek egy- illetve kétirányúak attól függően, hogy definiáljuk-e mindkét oldalon a kapcsolat fennállását. Az egyirányú kapcsolatok az adatbázis sémában ugyanolyan módon kerülnek elmentésre, mint a kétirányúak, ebből adódóan mindig lehetőség van olyan lekérdezés megfogalmazására, amely segítségével az egyirányú kapcsolat visszafelé is felderíthető.

A specifikáció lehetőséget teremt korlátozások megfogalmazására egy adott attribútummal kapcsolatosan. A kulcsmezőre nyilvánvalóan igaz, hogy nem lehet NULL értékű és egyedinek kell lennie az adott entitásra vonatkozóan. Az egyes attribútumokra is megfogalmazhatóak korlátozások az egyediségre és a kötelező kitöltésre vonatkozóan.

A kapcsolódó Bean Validation szabvány (16) segítségével további korlátozásokat adhatunk meg. Ezek közül példaként említjük meg a maximum/minimum (Max/Min) értéket és a szöveges attribútumokra vonatkozó reguláris kifejezéseket (Pattern).

Az eddig bemutatott tényezőkön kívül a szabvány lehetőséget teremt az objektum relációs leképzés további finomhangolására, ezek azonban nem befolyásolják döntően az általunk bemutatott megoldást.

A konkrét implementációt tekintve a fejlesztés során az EclipseLink nyílt forráskódú osztálykönyvtárat választottuk.

5.4 JPA adatmodell feldolgozása

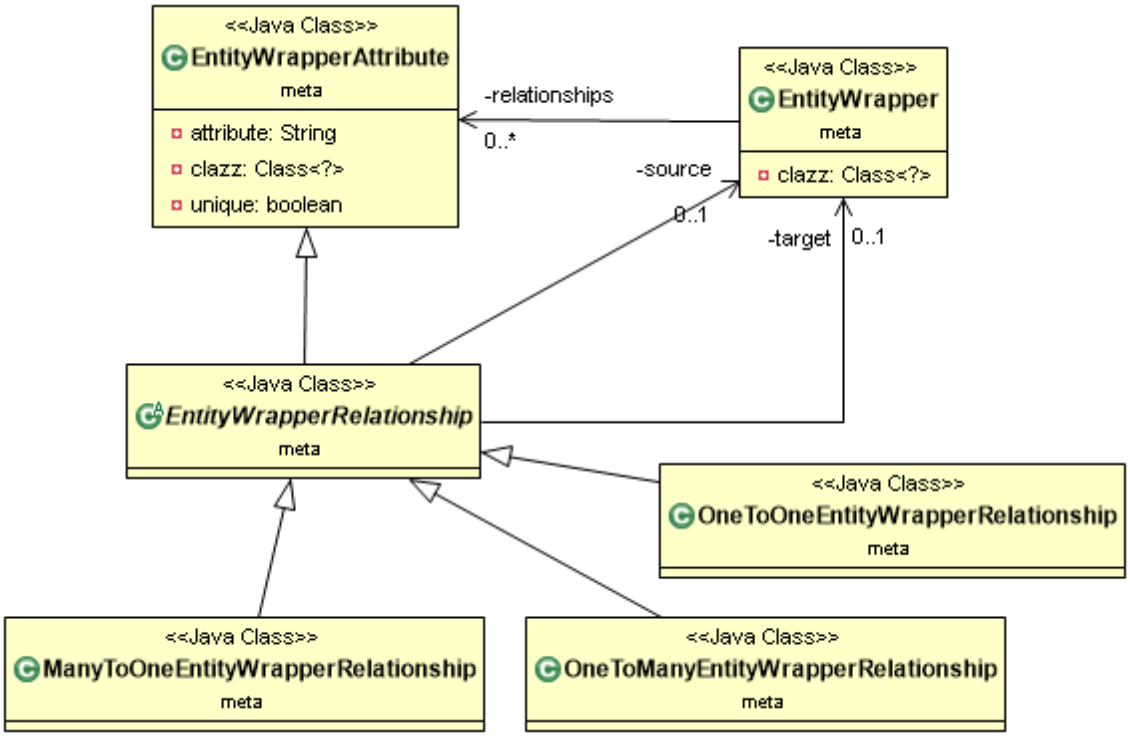
Az általunk fejlesztett eszköz elsősorban a Java osztályokon elhelyezett és futási időben elérhető (RetentionPolicy.RUNTIME típusú) annotációkat vizsgálja. A JPA

annotációkat is futási időben éri el a keretrendszer, tehát nem veszítünk el információt azáltal, hogy nem a forráskódot vizsgáljuk, hanem egy lefordított könyvtárat.

A forráskód szöveges feldolgozásánál jóval kifinomultabb és hatékonyabb eszköz a Java Reflection API. A Java Archive (jar) típusú könyvtárból egyenként feldolgozva a lefordított osztályokat előáll egy olyan lista a bemenetnek tekintett osztályokból, amelyek alapján már elő lehet állítani a belső modellt az entitásokról.

5.5 Metamodell a bemeneti osztályok feldolgozásához

A Java nyelv és a Reflection API által szolgáltatott metamodell nem elég rugalmas ahhoz, hogy dinamikusan tudjuk lekérdezni az éppen szükséges adatokat. Ezért döntöttünk úgy, hogy egy olyan struktúrát alkotunk, amely csak a számunkra lényeges adatokat tartalmazza könnyen feldolgozható formában.



10. ábra Metamodell a bemeneti osztályok feldolgozásához

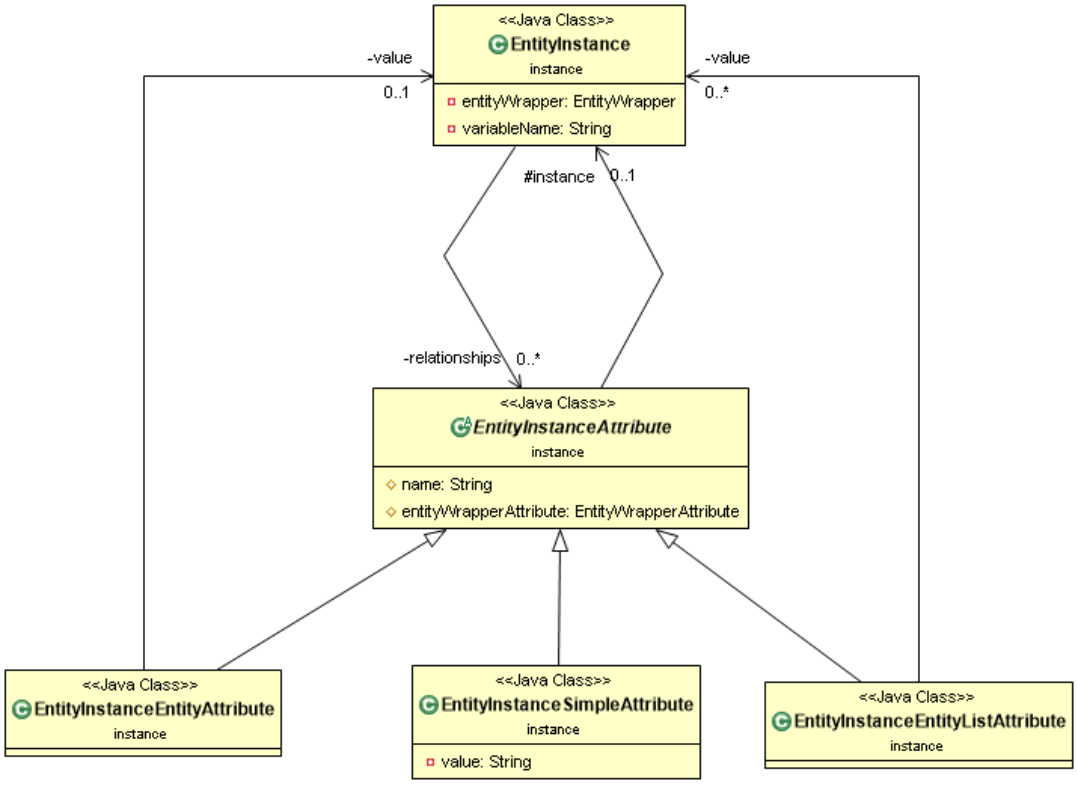
A modellben (10. ábra) a központi elem az *EntityWrapper*, amely a beépített Java *Class* osztályt egészíti ki az *Object Adapter* tervezési minta alapján (17). Egy entitásosztály rendelkezhet adattagokkal, amelyeknek az egyik változata az egyszerű attribútum. Az attribútum specializált változatai az egy-egy, több-egy és egy-több kapcsolatok.

A több-több kapcsolatot szándékosan hagytuk ki a rendszerből, mivel ezek hatékonyan helyettesíthetők egy több-egy és egy egy-több kapcsolattal. A helyettesítés már entitásszinten is nagyban növeli a rugalmasságot, ezért a tervezés során nagyon gyakran már ezen a szinten helyettesítik ezeket a kapcsolatokat.

A JPA által használt modellben ezek az adatok annotációk szintjén jelennek meg. Ezek feldolgozása után a metamodellbeli osztályok attribútumaként könnyen feldolgozhatók és értelmezhetők. A metamodell ráadásul könnyen bővíthető a megadható korlátozások támogatására (pl. min/max érték), ehhez csak új attribútumokat kell felvenni a megfelelő osztályokba.

5.6 Metamodell az objektumok generálásához

Az egyes objektumok hatékony generálásához szükséges a modell kiegészítése egy más szemléletmóddal. Ebben az esetben (11. ábra) a központi elem az *EntityInstance* (entitás példány), amely egy generálandó objektumot reprezentál. Ennek az osztálynak a felelőssége, hogy hozzáférést biztosítson a JPA modellbeli osztály minden adatához (*EntityWrapper* osztályon keresztül) és biztosítsa az adatokat, amelyek a generálás során szükségesek.



11. ábra Metamodell az objektumok generálásához

Az *EntityInstanceAttribute* egy konkrét kitöltendő attribútumot reprezentál, ezért itt lehetőség is van beállítani a konkrét értékeket. A forráskód összeállítása szempontjából három típusát különböztethetjük meg ezeknek:

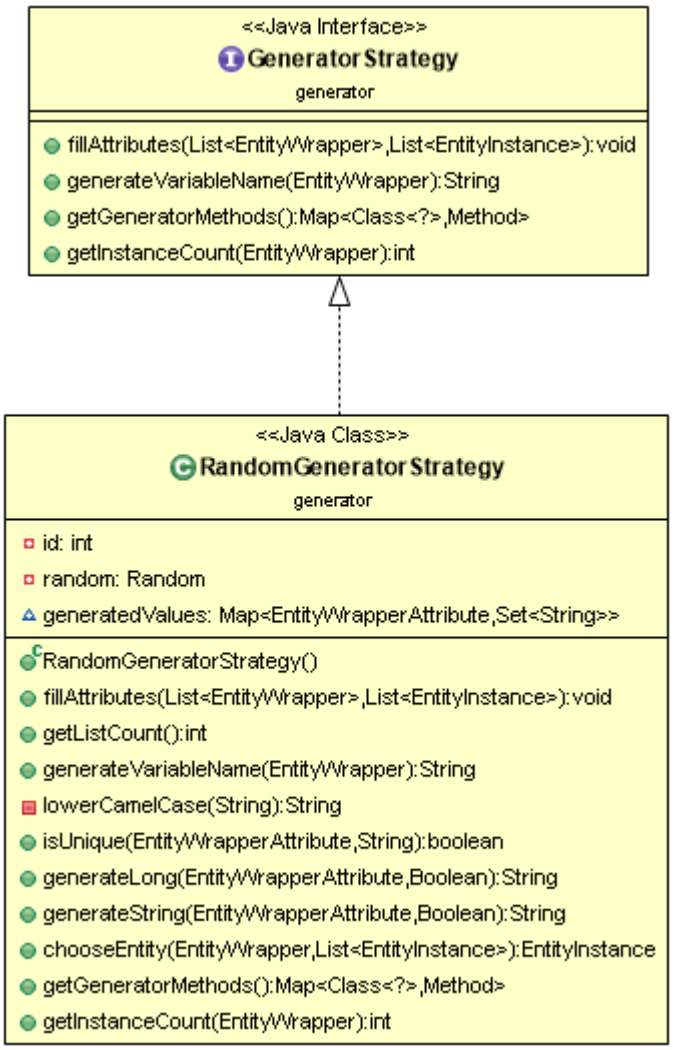
- Egyszerű attribútum (*EntityInstanceSimpleAttribute*),
- Entitás referencia (*EntityInstanceEntityAttribute*) és
- Entitás referenciák kollekciója (*EntityInstanceEntityListAttribute*).

Az egyszerű attribútum esetében jelenleg egy szöveges változóként lehet megadni az értéket, amely egyszerű behelyettesítésként fog megjelenni a forráskódban. Felvetődött külön osztálystruktúra kidolgozása a generálható értékeknek, de az ötletet elvetettük, mivel jelenlegi állapotában könnyebb felhasználni a Java nyelv osztályait, amelyek hangsúlyosan megjelennek az entitás-relációs leképzés során használt modellek esetében.

Az entitásreferencia egy másik generált entitás példányra mutat. Ebben az esetben az entitás azonosítója (a változó neve) kerül behelyettesítésre. Az entitáslista beállítása már többsoros kódot igényel, mivel egy listát tárol az ismert példányokról.

5.7 Generálási stratégia

A *Strategy* tervezési minta (17) alapján megtervezett generálási folyamat részeként meghatároztuk azokat a magas szintű lépéseket, amelyek a generáláshoz szükségesek (12. ábra). Ezeket tartalmazza a *GeneratorStrategy* interfész.



12. ábra A generálási stratégia

A példányok létrehozásának folyamata tehát a következő:

- Példányok létrehozása a modell alapján. Ehhez minden objektumnak egy egyedi azonosítót is elő kell állítani és el kell dönteni, hogy első körben melyik osztályhoz hány példányt generáljunk.
- Az attribútumok feltöltése, amelynek során az entitásosztályok és a példányok is szabadon lekérdezhetők és bejárhatók. Szükség szerint a további példányok létrehozása is lehetséges ebben a fázisban.

A megadott lépéseken kívül még egy metódus található az interfészben, amely visszaadja azokat a függvényeket, amelyek képesek generálni a beépített típusokat és osztályokat. A tervezés során két verziót is felvázoltunk az alaposztályok generálására: külön osztályok létrehozása vagy egyszerű metódusok használata. Végül a második lehetőség mellett döntöttünk, mivel ebben az esetben a generálási stratégia tervezőjére lehet bízni, hogy egy osztályba szervezi a teljes felelősséget vagy összetett működés esetén egy hierarchiába szervezi. Ilyen értelemben a *GeneratorStrategy* interfész megfeleltethető a *Facade* tervezési minta ajánlásainak (17).

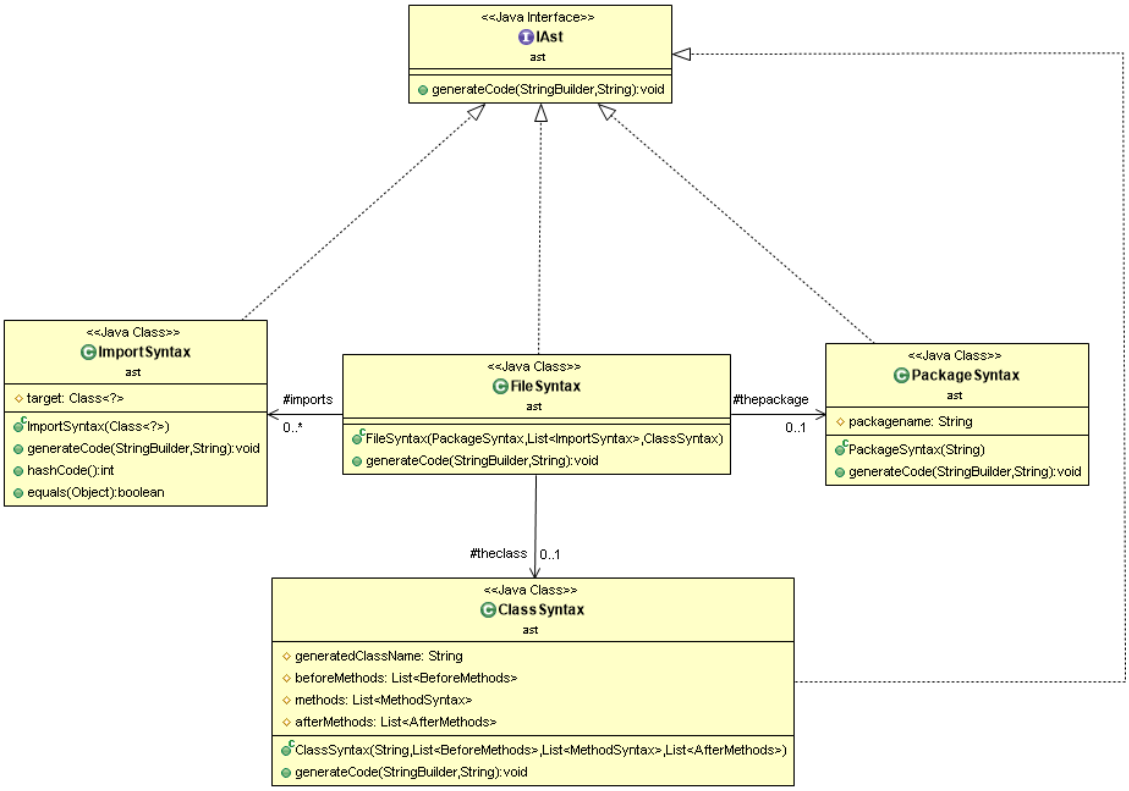
A megadott példastratégián jól látható, hogy a hatékony megvalósításhoz segédmetódusok szükségesek, de ezek létrehozása már a stratégia tervezőjén és megvalósítóján múlik.

5.8 Absztrakt szintaxis fa

Utolsó lépésként a felépített entitáspéldányokból absztrakt szintaxisfa (AST) segítségével fordítható forráskódot generálunk. Ehhez első körben definiáltunk egy olyan fastruktúrát, melynek segítségével könnyen tudjuk transzformálni a bemeneti modellt. A következő részben az öröklési hierarchia elemeit mutatjuk be.

Az absztrakt szintaxisfa magas szintű részei (13. ábra):

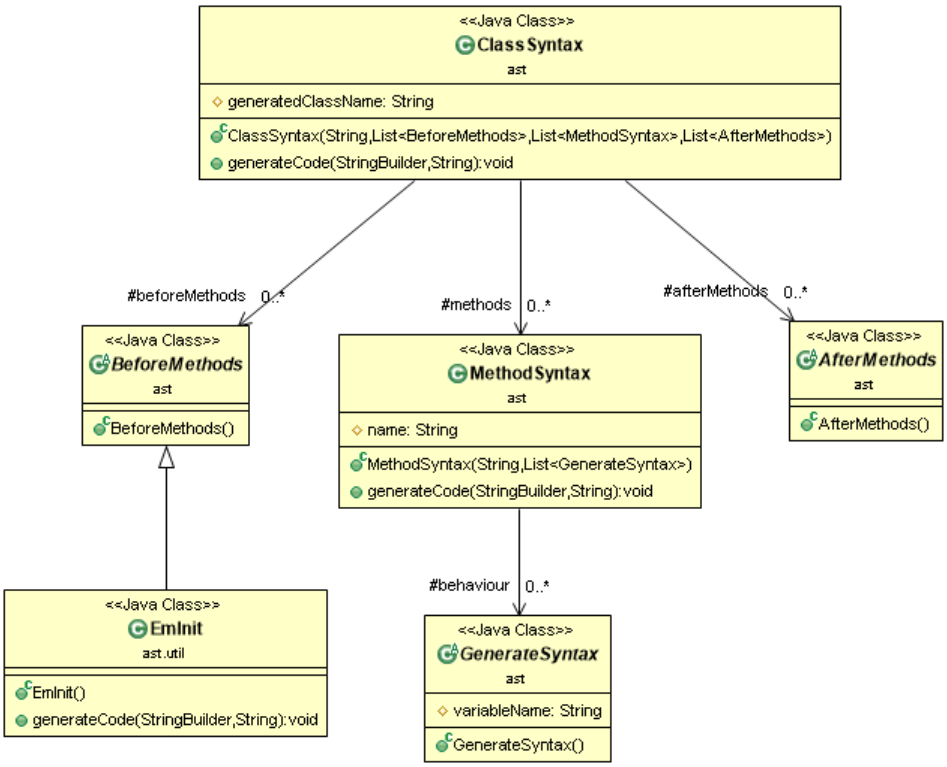
- **Gyökérelem** (IAst): IAst interfész, amely a kódgeneráláshoz szükséges metódust tartalmazza. A szintaxisfa minden eleme futtatható forráskódra alakítható.
- **Java fájl** (FileSyntax): Java nyelv alatt a legkisebb fordítási egység. Tartalmaz egy csomagdeklarációt, tetszőleges számú import deklarációt és egy (top-level) Java osztályt.
- **Csomag deklaráció** (PackageSyntax): A forráskód első sora, meghatározza a tartalmazó csomagot.
- **Import deklaráció** (ImportSyntax): A más csomagokban elhelyezkedő osztályokat külön import deklarációban kell bejelenteni. A mi esetünkben a JPA adatmodell osztályai szerepelnek itt.



13. ábra Absztrakt szintaxisfa összetett elemei

Az absztrakt szintaxisfa szerinti Java osztály felépítése (14. ábra):

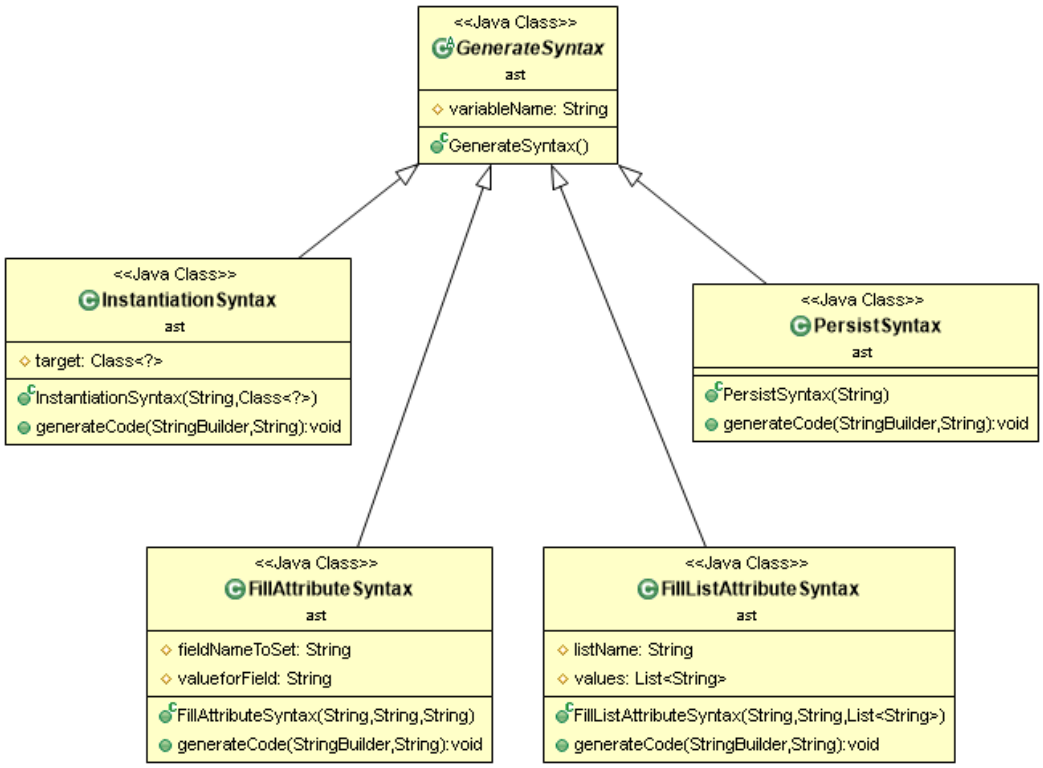
- **Java osztály** (ClassSyntax): Jelen esetben az entitás példányok létrehozási logikájának tartalmazója. Esztétikai szempontból három fontos részét különböztetjük meg: a metódusok előtt megjelenő speciális kódokat, a generátor metódusokat és az ezek után megjelenő részeket.
- **Metódusok előtti részek** (BeforeMethods): Általános konvenció, hogy az attribútumokat a metódusok előtt jelenítjük meg. Ezt egy külön osztály segítségével érjük el, melynek (a program jelen állapotában) egyetlen leszármazottja van, az entitásmenedzser objektum (EntityManager).
- **Metódus** (MethodSyntax): A metódus egy nagyon leegyszerűsített modelljét használjuk, amely a mi szempontunkból releváns generátor logikai egységekből épül fel.
- **Generátor egység** (GenerateSyntax): Közös ősosztály a generálás során alkalmazott műveletekhez.



14. ábra Absztrakt szintaxisfa osztály generálásához

A generálás során alkalmazott atomi műveletek (15. ábra):

- **Példányosítás** (InstantiationSyntax): A megadott osztály és változónév alapján létrehoz egy új példányt.
- **Attribútum kitöltése** (FillAttributeSyntax): A megadott attribútum értékét állítja be a megfelelő értékre.
- **Lista attribútum kitöltése** (FillListAttrbuteSyntax): A lista attribútumok kitöltése többsoros utasítást igényel. Ezt képes előállítani ez az osztály.
- **Adatbázisba írás** (PersistSyntax): A megadott példány adatbázisba perzisztálása.



15. ábra Absztrakt szintaxisfa generálási lépésekhez

A szintaxisfa felépítése után inorder bejárással (18) kapható meg a generált forráskód. Az inorder bejárás során először a baloldali részfát, majd az adott elemet és végül a jobboldali részfát látogatjuk meg.

5.9 Összegzés

Ebben a fejezetben a rendszer magas szintű terveit mutattuk be. Az ötlet lényege, hogy a bemenetként megadott adatbázismodellt különböző transzformációknak alávetve olyan formába hozzuk, hogy egy megadott generálási stratégia hatékonyan tudjon vele dolgozni.

A példányok létrehozása után absztrakt szintaxisfa bejárás segítségével generáljuk le a kimenetként megjelenő forráskódot.

6 Az eszköz implementálása

Az eszköz megvalósítása során több implementációs kihívással is találkoztunk. Ebben a fejezetben ezeket a fő kérdéseket fogjuk körüljárni és bemutatjuk az eszköz működését.

6.1 A szöveges forráskód generálásának implementációs aspektusai

A tervezés során összemértük az egyszerű szövegösszefűzésen és az absztrakt szintaxisfán alapuló megoldást. A rugalmasság és a modellezési hatékonyság miatt az összetett osztályhierarchiát igénylő szintaxisfa mellett döntöttünk.

A Java platform beépített *String* osztálya ún. immutable (módosíthatatlan) tulajdonsággal rendelkezik, melynek lényege, hogy a belső állapotát létrehozás után nem lehet megváltoztatni. Ennek legfőbb motivációja, hogy ezáltal könnyebb átlátni a működést és nagymértékben egyszerűsíti a konkurens programok tervezését.

Objektumorientált környezetben azonban az objektumok létrehozása erőforrás igényes művelet, tehát minden egyes változtatás egy szöveges értéken nagymértékben lassítani tudja a programot.

A megoldás erre a problémára a *StringBuilder* vagy *StringBuffer* osztályok használata, amelyeknek változtatható a belső állapota és erre külön támogatást nyújtanak a dedikált metódusok.

Megoldás	Időigény
String összefűzés	22 746 ms
StringBuffer	12 ms
StringBuilder	11 ms

2. táblázat Megoldások összehasonlítása

A 2. táblázat mutatja az egyes megoldások időigényét. A megoldások sebességeinek összehasonlításához egytől százezerig fűztük össze a számokat egyetlen szöveggé. Látható, hogy a *StringBuilder* osztályra építő módszer a leghatékonyabb, az egyszerű összefűzés nagyságrendekkel lassabb.

Végeredményben tehát a szintaxisfa építése alapján állítjuk össze a forráskódot a *StringBuilder* beépített Java osztály segítségével.

6.2 Generálási stratégiák

Az elkészült keretrendszerhez átlátható módon készíthető új generálási stratégia. A tervek működőképességének bizonyítására két stratégiát készítettünk el.

Az egyes stratégiáknak négy alapvető működést kell megvalósítaniuk:

- generálandó példányok számának megadása,
- változónevek generálása,
- generált típusok és
- attribútumok kitöltése.

Az első stratégia (*RandomGeneratorStrategy*) véletlenszerűen tölti ki az adattagokat és a kapcsolatokat, míg a második stratégia az egyszerű attribútumokat adatlistából tölti fel, a kapcsolatokat pedig egy gráfalgoritmus segítségével alakítja ki.

6.3 Véletlenszerű generálás

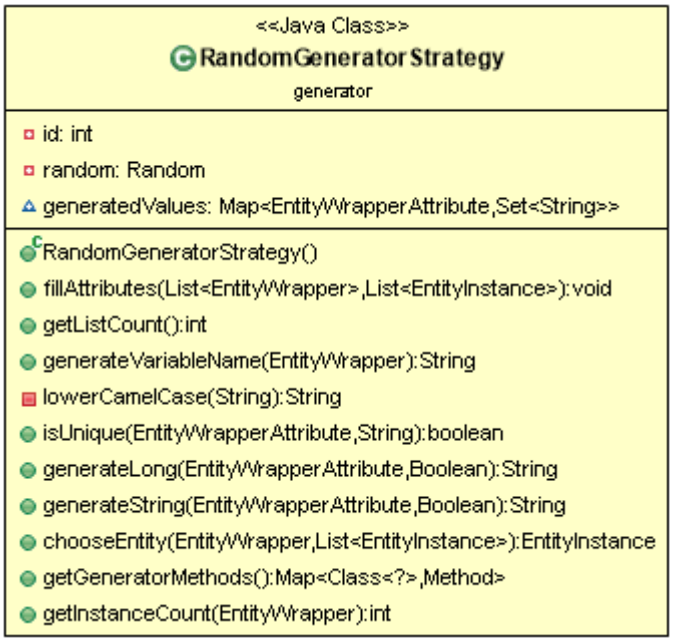
A véletlenszerűen generált tesztadatok esetén fontosnak tartottuk, hogy a megfelelő bemeneti kezdőérték beadásával lehetőség legyen ugyanazt a véletlensorozatot kétszer is determinisztikusan generálni. Ehhez egy véletlenszám generátort attribútumként felvettünk (a megadott értékkel inicializáltuk), amelyet az összes metódus használ (16. ábra).

A generálandó példányszámokra alsó és felső határokat szabtunk meg paraméterként, amelyek között tetszőleges számú objektum generálható.

A változónevek generálásához egy inkrementálisan növekvő azonosítót (id) alkalmaztunk, melyet mindig a generálandó osztály Java konvenciók szerint kiírt (lower camelcase) nevéhez fűzünk.

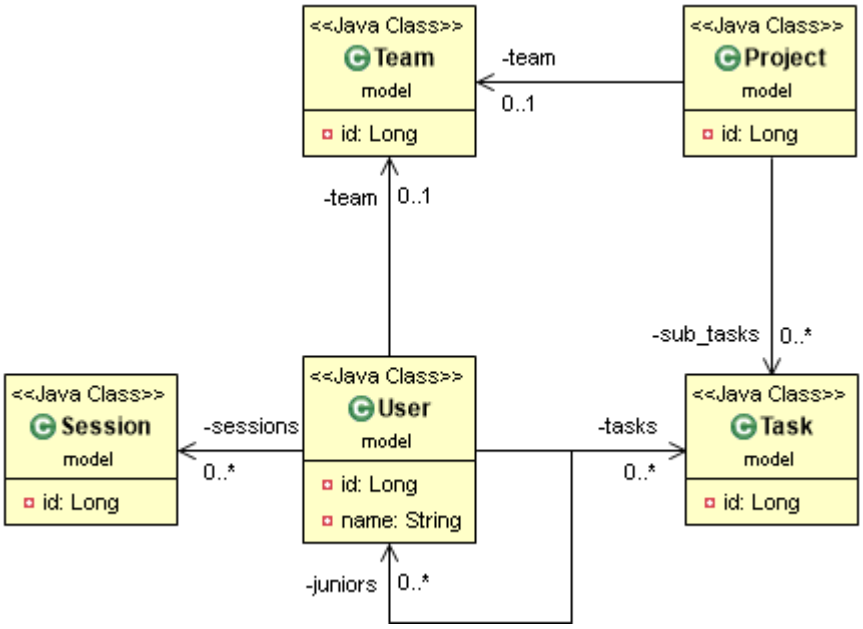
A stratégia szöveges (*String*) és egész szám (*Long*) attribútumokat képes generálni, amelyek véletlenszámokból állnak össze.

A kapcsolatok kitöltése során véletlenszerűen választjuk ki a lehetséges célpontok közül a megfelelőt. Ehhez listaattribútumok esetén egy újabb változó érték adódik, a tartalmazott elemek száma.



16. ábra Véletlenszerű generálási stratégia

6.4 Véletlenszerű generálás a gyakorlatban



17. ábra Bemeneti modell

A tesztelés során használt bemeneti modellre (17. ábra) futtattuk a véletlenszerű generálási stratégiát. A következő kódrészletben jól látható, hogy mind az öt entitásból készül példány, melyeket véletlenszerűen, de típushelyesen köt össze az eszköz.

```java
package generated;

import model.Task;
import model.Team;
import model.User;
import model.Project;
import model.Session;

import javax.persistence.EntityManager;

public class Generated {
	EntityManager em = null;

	public void generate() {
		Session session0 = new Session();
		Session session1 = new Session();
		Team team2 = new Team();
		Team team3 = new Team();
		Team team4 = new Team();
		Project project5 = new Project();
		Project project6 = new Project();
		Task task7 = new Task();
		Task task8 = new Task();
		User user9 = new User();
		User user10 = new User();
		User user11 = new User();
		session0.setId(50436785808272846391);
		session1.setId(-54169025111496012931);
		team2.setId(-46890019710417675781);
		team3.setId(52384931927542392821);
		team4.setId(30419543813668424171);
		project5.setId(507397347020377765581);
		project5.getSubTasks().add(task8);
		project5.getSubTasks().add(task7);
		project5.getSubTasks().add(task8);
		project5.setTeam(team4);
		project6.setId(-202971647850155713871);
		project6.getSubTasks().add(task7);
		project6.getSubTasks().add(task7);
		project6.setTeam(team2);
		task7.setId(85951751699613039301);
		task8.setId(-55516670306053481081);
		user9.setId(251618861683043438 11);
		user9.setName("generatedstring5553183103231869362");
		user9.getJuniors().add(user11);
		user9.getJuniors().add(user10);
		user9.getSessions().add(session1);
		user9.getSessions().add(session1);
		user9.getSessions().add(session1);
		user9.getTasks().add(task8);
		user9.getTasks().add(task7);
		user9.getTasks().add(task7);
		user9.setTeam(team3);
		user10.setId(51756863708851292951);
		user10.setName("generatedstring-71347601201772322641");
		user10.getJuniors().add(user9);
		user10.getJuniors().add(user11);
		user10.getJuniors().add(user10);
		user10.getSessions().add(session0);
```

```
            user10.getSessions().add(session0);
            user10.getSessions().add(session1);
            user10.getTasks().add(task7);
            user10.getTasks().add(task8);
            user10.getTasks().add(task7);
            user10.setTeam(team2);
            user11.setId(91094367803516117691);
            user11.setName("generatedstring18382105495495464 97");
            user11.getJuniors().add(user11);
            user11.getJuniors().add(user11);
            user11.getJuniors().add(user9);
            user11.getSessions().add(session0);
            user11.getSessions().add(session1);
            user11.getSessions().add(session1);
            user11.getTasks().add(task8);
            user11.getTasks().add(task7);
            user11.getTasks().add(task7);
            user11.setTeam(team2);
            em.persist(session0);
            em.persist(session1);
            em.persist(team2);
            em.persist(team3);
            em.persist(team4);
            em.persist(project5);
            em.persist(project6);
            em.persist(task7);
            em.persist(task8);
            em.persist(user9);
            em.persist(user10);
            em.persist(user11);
        }
}
```

6.5 Gráf alapú generálás

Gyakran felmerül az igény, hogy olyan objektumstruktúra álljon össze, amelynek gráfjában nem szerepel kör. Ebben az esetben a gráf csomópontjai az egyes entitáspéldányok, az élek pedig az ezek között beállított kapcsolatok. Fontos, hogy mivel ezek a kapcsolatok adatbázisszinten oda-vissza bejárhatóak, ezért irányítatlan gráfot feltételezünk. Mindezek mellett nincs implementációs akadálya az irányítatlan változat gyors megvalósításának.

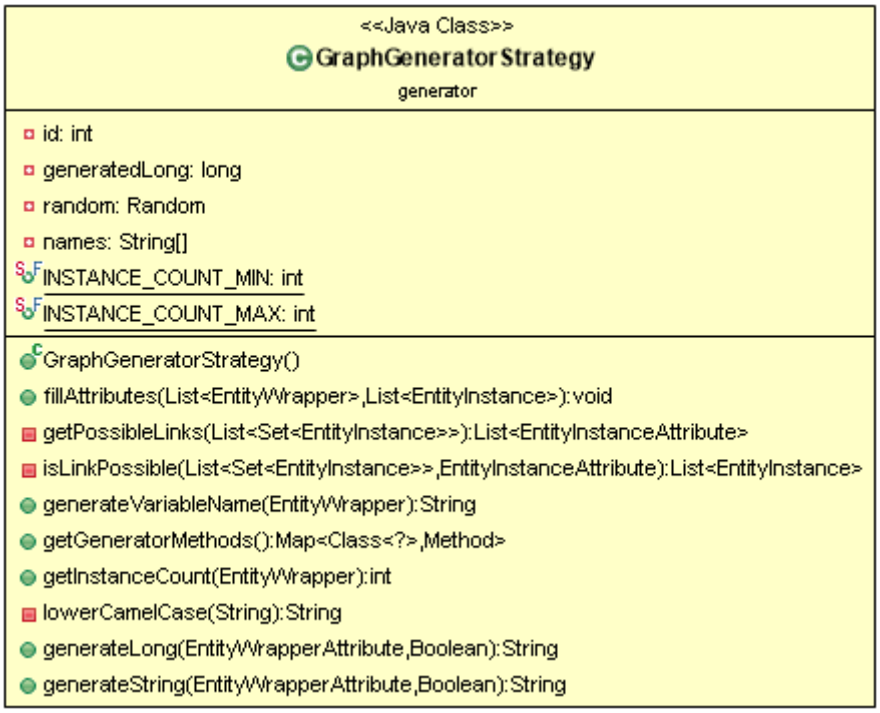
A körmentes struktúra előállításához egy, a Kruskal algoritmushoz hasonló gráfalgoritmust választottunk (19). A kiindulópont egy olyan gráf, amelyben a lehetséges kapcsolatok alapján vannak berajzolva az élek. Ebben a gráfban keresünk egy feszítőfát.

A működés alapja, hogy folyamatosan nyilvántartjuk a gráfban megjelenő elszigetelt részeket és minden lépésben két ilyet próbálunk összekötni. Mivel az egyes

éleknek nincs költségük (hosszuk), ezért minden lépésben véletlenszerűen választjuk ki azt az élet, amellyel összekapcsolható két sziget a gráfban.

Az algoritmus futása két esetben állhat le: már csak egy nagy összefüggő szigetből áll a gráf (összeállt a feszítőfa) vagy ha már az egyes szigetek között nincs él (ebben az esetben feszítőerdőt kaptunk).

Az eredeti feszítőfa építő algoritmustól még egy ponton szükséges volt eltérni, mivel egyszerű entitásra mutató attribútum esetén csak egy kapcsolat lehetséges, így egy ilyen felhasználásakor törölni kell a gráfból az ehhez tartozó éleket. Listaattribútum esetén ezt a törlést nem szükséges elvégezni.



18. ábra Gráf alapú generálási stratégia

A gráf alapú generálási stratégiát (18. ábra) kiegészítettük beépített szöveglistán alapuló létrehozással.

6.5.1 Az algoritmus formális leírása

Adott $G = (V, E)$ gráf, ahol V halmaz az entitás példányok halmaza, E pedig a példányok közötti összes lehetséges kapcsolat alapján kialakított élek halmaza. Egy e élet az $e(v, u, a)$ hármas jellemez, ahol az e él v példányból megy u példányba az a attribútum mentén.

Jelölje $T(e(v,u,a))$ a *v* példányhoz és *a* attribútumhoz tartozó élek halmazát (az *e* élet kivéve), ahol *a* entitás referencia attribútum és legyen $T(e(v,u,a)) = \emptyset$ minden más esetben. Nevezzük a $T(e)$ halmaz elemeit az *e* él társéleinek.

Legyen C_1 és C_2 diszjunkt, összefüggő körmentes részgráf (fa) G gráfból, melyeket egy *e(u, v, a)* él köt össze. Ekkor C_1 és C_2 gráfokat összekötve *e* él mentén olyan *C* gráfot kapunk, mely szintén összefüggő és körmentes (fa) (*).

Egy adott *C* részgráf csak akkor értelmezhető $G = (V, E)$ gráfon, ha $T(e(v,u,a)) \cap E = \emptyset, \forall e \in C$ teljesül, azaz az adott fában nem lehetnek olyan élek, melynek társ élei a gráfban szerepelnek (**).

Ekkor az algoritmus lépései a következők:

1. Legyenek C_1, C_2,…,C_N diszjunkt egy elemű részgráfok, ahol $|V| = N$, azaz minden csúcsra létrehoztunk egy részgráfot.
2. Amíg $|C_1| \neq |V|$ és lehetséges a továbblépés, addig kössük össze C_i és C_j ($i<j$) részgráfokat valamely *e(v,u,a)* él mentén. Az eredmény legyen C_i új értéke, töröljük C_j gráfot és legyen $G = G\big(V, E \backslash T(e)\big)$.
3. Ha a lefutás után $|C_1| = |V|$, akkor megkaptuk a feszítőfát. Ha nem tudtunk továbblépni, akkor az algoritmus eredménye C_i megmaradt feszítőfákból álló erdő lesz, azaz $\bigcup_{\forall i} C_i$.

Az algoritmus eredménye feszítőfa (erdő) lesz, mivel kezdetben minden C_i ($i=1…N$) feszítőfa és a (*) tulajdonság miatt végig feszítőfákkal dolgozunk.

Látható, hogy a (**) feltétel végig teljesül a futás során, mivel mindig eltávolítjuk azokat az éleket, amelyek ezt a tulajdonságot megsértenék.

Ezzel az algoritmus helyességét beláttuk.

6.6 Gráf alapú generálás a gyakorlatban

Bementi modellként ugyanazt az entitásmodellt alkalmaztuk, mint az előző stratégia tesztelésénél (18. ábra). A gráf alapú stratégia alapján azt várjuk, hogy fastruktúrába szerveződjenek az egyes példányok. Ez kiolvasható a közvetkező (generált) forráskódból is, de mellékeltük az ehhez rendelhető gráfot is (19. ábra).

```
package generated;

import model.User;
import model.Team;
import javax.persistence.EntityManager;
import model.Project;
import model.Task;
import model.Session;

public class Generated {
	EntityManager em = null;

	public void generate() {
		Team team0 = new Team();
		Team team1 = new Team();
		Task task2 = new Task();
		Task task3 = new Task();
		User user4 = new User();
		User user5 = new User();
		Session session6 = new Session();
		Session session7 = new Session();
		Session session8 = new Session();
		Project project9 = new Project();
		Project project10 = new Project();
		Project project11 = new Project();
		team0.setId(0l);
		team1.setId(1l);
		task2.setId(2l);
		task3.setId(3l);
		user4.setId(4l);
		user4.setName("Péter");
		user4.getSessions().add(session8);
		user4.getSessions().add(session7);
		user4.getSessions().add(session6);
		user4.getTasks().add(task2);
		user4.setTeam(team0);
		user5.setId(5l);
		user5.setName("Balázs");
		user5.getJuniors().add(user4);
		user5.getTasks().add(task3);
		user5.setTeam(team1);
		session6.setId(6l);
		session7.setId(7l);
		session8.setId(8l);
		project9.setId(9l);
		project9.setTeam(team1);
		project10.setId(10l);
		project10.getSubTasks().add(task2);
		project10.setTeam(null);
		project11.setId(11l);
		project11.setTeam(team1);
		em.persist(team0);
		em.persist(team1);
		em.persist(task2);
		em.persist(task3);
		em.persist(user4);
		em.persist(user5);
		em.persist(session6);
		em.persist(session7);
```

```
            em.persist(session8);
            em.persist(project9);
            em.persist(project10);
            em.persist(project11);
        }
}
```

19. ábra Az entitás példányok gráfja

6.7 Összegzés

A gyakorlati tesztelések eredménye alapján levonhatjuk a következtetést, hogy az eszköz képes elvégezni a tervekben meghatározottakat, ráadásul könnyen bővíthető olyan stratégiával, amely az aktuális környezethez illeszthető.

A létrehozott forráskód könnyen értelmezhető és az eszköz jól használható arra is, hogy az adatbázisséma újragenerálása vagy változása esetén inicializálni lehessen tesztadatokkal.

7 Összegzés

Dolgozatunkban a Java platform szabványosított objektum-relációs leképzését, a Java Persistence API-t használva mutattunk be platformfüggetlen megoldást a tesztadat-generálás kiterjesztett problémájára.

A fejlesztés korai fázisában megtervezett entitás osztályok alapján lehetővé tettük a tesztadatok generálását és az ezek közötti kapcsolatok olyan formában történő kialakítását, hogy az így előálló adatkészlet minél hatékonyabban le tudja fedni a rendszer működését, beleértve mind az üzemszerű, mind a kivételes helyzeteket.

Az általunk javasolt eszköz hasznos lehet nemcsak a tesztelési, hanem a fejlesztési feladatok során is, ezáltal rövidítve a fejlesztési időt és növelve az elkészült termék minőségét.

7.1 Továbbfejlesztési lehetőségek

Az elkészültekhez nagyon sok továbbgondolási lehetőség adja magát. A jövőben új stratégiákat szeretnénk tervezni, implementálni és kiértékelni. Ezen kívül implementációs kérdésekben több további célkitűzést fogalmaztunk meg, mint például a grafikus felület készítése a paraméterek beállításához, az eszköz képességeinek kiterjesztése a JDO specifikációra, illetve a párhuzamosított generálás tranzakció-kezeléssel megvalósítva.

További bővítési lehetőség lehet a stratégiák összetett paraméterezhetőségének támogatása. Ezeket a változókat akár valamilyen szöveges leírókból is be lehetne olvasni (pl. XML, JSON vagy DSL).

Távlati célként megfogalmaztuk az adapterek írását a piacon lévő többi szoftverhez, melynek révén fel tudnánk használni a meglévő jól paraméterezhető típusokat és valós adatmintákat.

A teljesség igénye nélkül felsoroljuk azokat a tényezőket, melyek hozzájárulhatnának ahhoz, hogy a megtervezett eszköz széles körben használható legyen:

- intelligens típusfelismerés,
- JUnit integráció,
- jelentések készítése,

- elosztott futtatás lehetőségének vizsgálata mind az adatgenerálás, mind az inicializáció futtatása során,
- fejlesztőkörnyezetekhez bővítmény megírása.

Irodalomjegyzék

1. **Royce, Dr. Winston W.** *Managing the development of large software systems.* 1970.

2. **Bucanac, Christian.** *The V-Model.* 1999.

3. **Aked, Mark.** Risk reduction with the RUP phase plan. *IBM.* [Online] [Hivatkozva: 2013. október 23.] http://www.ibm.com/developerworks/rational/library/1826.html#N100E4.

4. **Matthias M. Muller, Frank Padberg.** *About the Return on Investment of Test-Driven Development.* 2012.

5. **Binder, Robert V.** *Testing object-oriented systems.* USA : Addison-Wesley, 1999.

6. **Beizer, Boris.** *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* hely nélk. : Wiley, 1995.

7. **Srinivasan Desikan, Gopalaswamy Ramesh.** *Software Testing: Principles and Practices.* India : Dorling Kindersley, 2008.

8. **Elfriede Dustin, Jeff Rashka, John Paul.** *Automated Software Testing.* hely nélk. : Addison-Wesley, 1999.

9. Automated Testing. *IT Glossary.* [Online] Gartner. [Hivatkozva: 2013. október 23.] http://www.gartner.com/it-glossary/automated-testing-and-quality-management-distributed-and-mainframe.

10. Mockaroo. [Online] [Hivatkozva: 2013. október 23.] http://www.mockaroo.com/.

11. **William G.J. Halfond, Jeremy Viegas, and Alessandro Orso.** *A Classification of SQL Injection Attacks and Countermeasures.* Georgia : ismeretlen szerző, 2006.

12. **Ira R. Forman, Nate Forman.** *Java Reflection in Action.* Greenwich : Manning Publications, 2004.

13. **James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley.** *The Java Language Specification.* 2013.

14. **Podeswa, Howard.** *UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering* . hely nélk. : Thomson Course Technology, 2005.

15. **DeMichiel, Linda.** *JSR 317: Java Persistence API, Version 2.0.* hely nélk. : Sun, 2009.

16. **Bean Validation Expert Group.** *Bean Validation.* hely nélk. : Red Hat, 2009.

17. *Design Patterns: Elements of Reusable Object-Oriented Software.* **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** USA : Addison-Wesley, 1994.

18. **Gerdemann, Dale.** *Parsing as tree traversal.* 1994 .

19. **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.** *Introduction to alghorithms.* Massachusetts : Massachusetts Institute of Technology, 2009.

8 Ábrák jegyzéke