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

Ferencz Endre

Bozóki Szilárd

JPA ALAPÚ TESZTELÉSTÁMOGATÁS

KONZULENSEK

Budai Péter

Dr. Goldschmidt Balázs

BUDAPEST, 2014

Tartalomjegyzék

Összefoglaló

Napjaink összetett információs rendszerei a felhasználói igények széles skáláját kielégítik, miközben a különböző integrációs lehetőség miatt számos függőséggel rendelkeznek. Ebből kifolyólag mind a hagyományos és az agilis szoftverfejlesztési módszertanok jelentős hangsúlyt fektetnek a tesztelésre. A legtöbb online tartalmat előállító szerver oldali alkalmazás relációs adatbázisokra épül, mégis az objektum relációs leképzés tesztelését megvalósító eszközök száma limitált és nem elég az igények kielégítésére.

A fejlesztés és tesztelés során rendelkezésre állót adat nagyban különbözhet az éles rendszerétől. Főleg biztonsági és adatvédelmi szempontok miatt aggályos az éles rendszer adatainak a használata. Megfelelő tervezéssel azonban előállítható olyan adatstruktúra, ami megfelelően lefedi a felhasználói esetek többségét. Az adatmezők feltöltése a teszt adat generálásnak csak egy kisebb része, ugyanis lényegesen nehezebb kezelni az objektumok közötti bonyolult kapcsolatrendszert, ami döntően befolyásolhatja a működést.

A probléma megoldására mi egy platform független megoldást nyújtunk, ami a Java környezet szabványos objektum relációs leképzésére, a Java Persistence API-ra építkezik. A megoldásunk Java entitás osztályokból kiindulva generál megfelelő kapcsolati rendszerrel rendelkező minta adatokat, amik képesek lefedni a normális és a kivételes esetek nagy részét. A megoldásunkban nagy hangsúlyt fektettünk a generált adatok minőségére, ezért nagyfokú testre-szabhatóságot biztosítottunk.

Az általunk kínált eszköz jelentősen elősegíti a minőségi termékek hatékony előállítását azáltal, hogy a szoftverfejlesztési folyamat legtöbb részében használható, mert már a legkorábbi fejlesztési fázistól kezdve bevethető.

Abstract

While contemporary complex information technology systems satisfy a wide variety of user needs, they also have numerous interdependencies due to the vast number of integration possibilities. Owing to this, both the conventional and agile software development methodologies have a strong emphasis on testing. The majority of the on-line content generating server side applications is built over relational databases. However, the available tools which are capable of testing an object-relationship mapping based applications are insufficient.

The data available during development and testing might greatly differ from real ones. From security and privacy point of view the usage of real data is especially solicitous. By adequate planning it is possible to create a data structure which covers most of the use cases. The filling of data fields is just a smaller part of the data generation, because it is more difficult to handle the complex object relations, which could significantly alter behavior.

We provide a platform independent solution to the problem based on the Java Persistence API, which is the standard object-relationship mapping of the Java environment. Our solution generates sample data from Java entity classes with appropriate relationships, which cover most standard and exceptional cases. In our solution we put great emphasis on the quality of the generated data, so a high level of customization is supported.

Our tool greatly helps the effective creation quality products by being usable in most parts of the software development process, as it is deployable from the earliest development phase.

1 Bevezetés

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.

1.1 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.

Véleményünk szerint az egységteszteléstől a felhasználói elfogadási tesztekig minden fázisban szükség lehet egy jó minőségű adatokkal feltöltött adatbázisra.

Az általunk tervezendő eszközt csak dinamikus tesztek esetén lehet hatékonyan felhasználni. Az automatizált tesztadat-generálás a szürke vagy a fehér dobozos tesztelést képes kiszolgálni.

1.2 Eszközök automatizált teszteléshez

A legtöbb gyakorlati teszteléssel foglalkozó szakirodalom (1) 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.

Az automatizált tesztelés során alkalmazható eszközök közül részletesebben az automatizált tesztadat-generálást támogatóakat vizsgáltuk meg.

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
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.

Nem találtunk olyan eszközt, amely képes lett volna objektumorientált megközelítéssel ORM leképzésen keresztül tesztadatokat generálni.

1.3 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.

1.4 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.

1.5 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.

A Java Persistence API annotációi segítségével definiált adatmodell feldolgozására tehát 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.

2 Automatizált tesztadatbázis-generálás JPA alapokon

A bemenet egy sor transzformációs folyamat során 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.

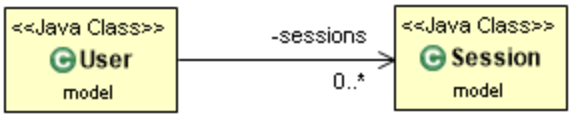
Az osztálykönyvtár feldolgozása során az eszköz a célra kialakított metamodell alapján a Java Reflection API segítségével előállítja a modell egy olyan formáját, amelyet már sokkal kényelmesebben tudunk feldolgozni és átalakítani. A metamodell legfőbb célja, hogy képes legyen hatékonyan megragadni a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

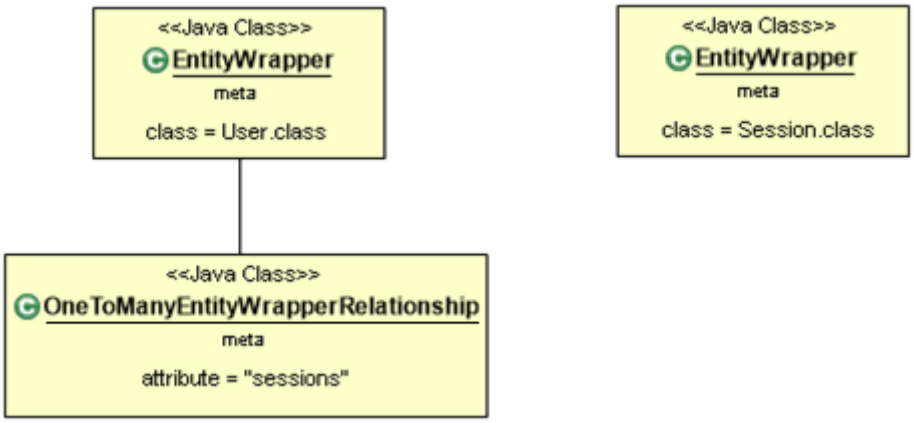
Végül a generálási stratégia alapján előállíthatóak a kívánt attribútumok és kapcsolatok, mivel ezen a ponton mind osztályszinten, mind példányszinten lehetőség van vizsgálatokat végezni, így teljes körű információval rendelkezünk a bemenetként megkapott modellről és a generált objektumokról.

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.

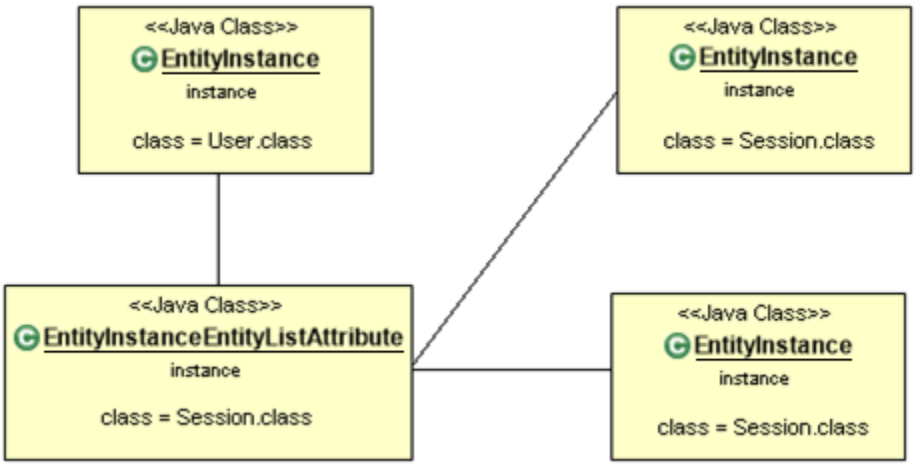
A generálási folyamat első lépéseként a bemeneti modellt (példa: 1. ábra) dolgozzuk fel, melynek eredménye a feldolgozott osztályok egy belső reprezentációja (példa: 2. ábra). A belső reprezentáció alapján jönnek létre logikai szinten a példányok (példa: 3. ábra), amely alapján felépíthető az absztrakt szintaxisfa.



1. ábra Bemeneti modell



2. ábra Bementi osztályok feldolgozása



3. 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.

2.1 JPA adatmodell feldolgozása

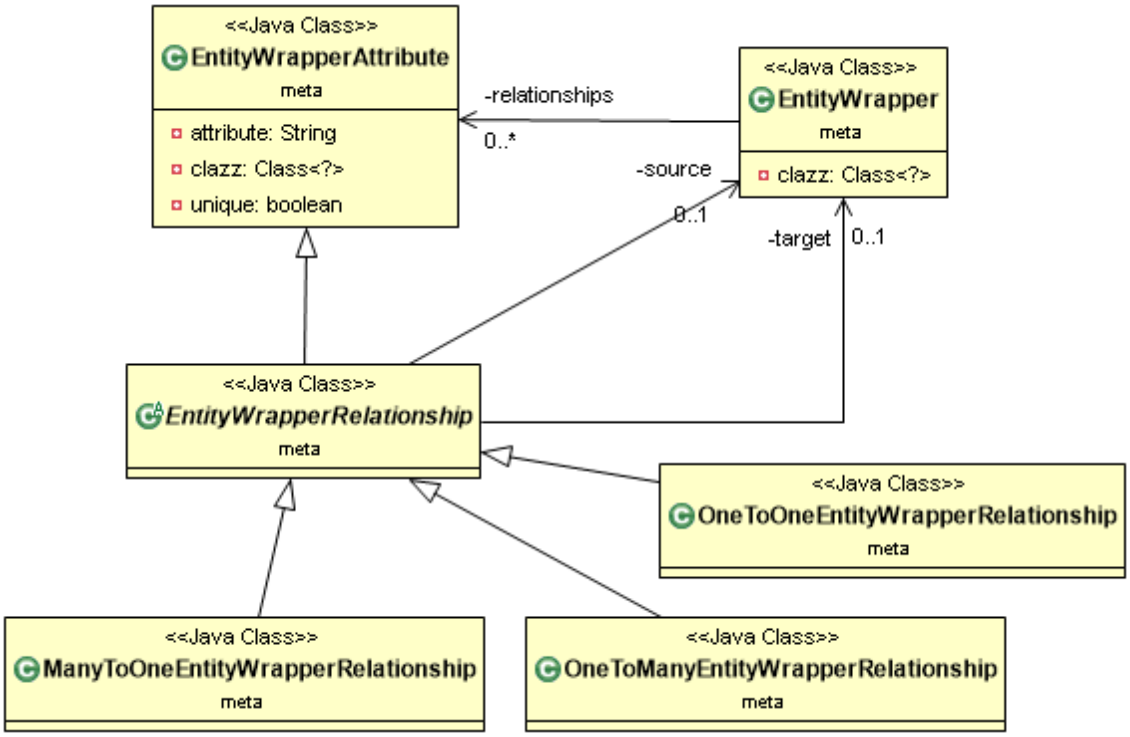
Az általunk fejlesztett eszköz elsősorban a Java osztályokon elhelyezett és futási időben elérhető (RetentionPolicy.RUNTIME típusú) annotációkat vizsgálja. A JPA

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.

2.2 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.



4. ábra Metamodell a bemeneti osztályok feldolgozásához

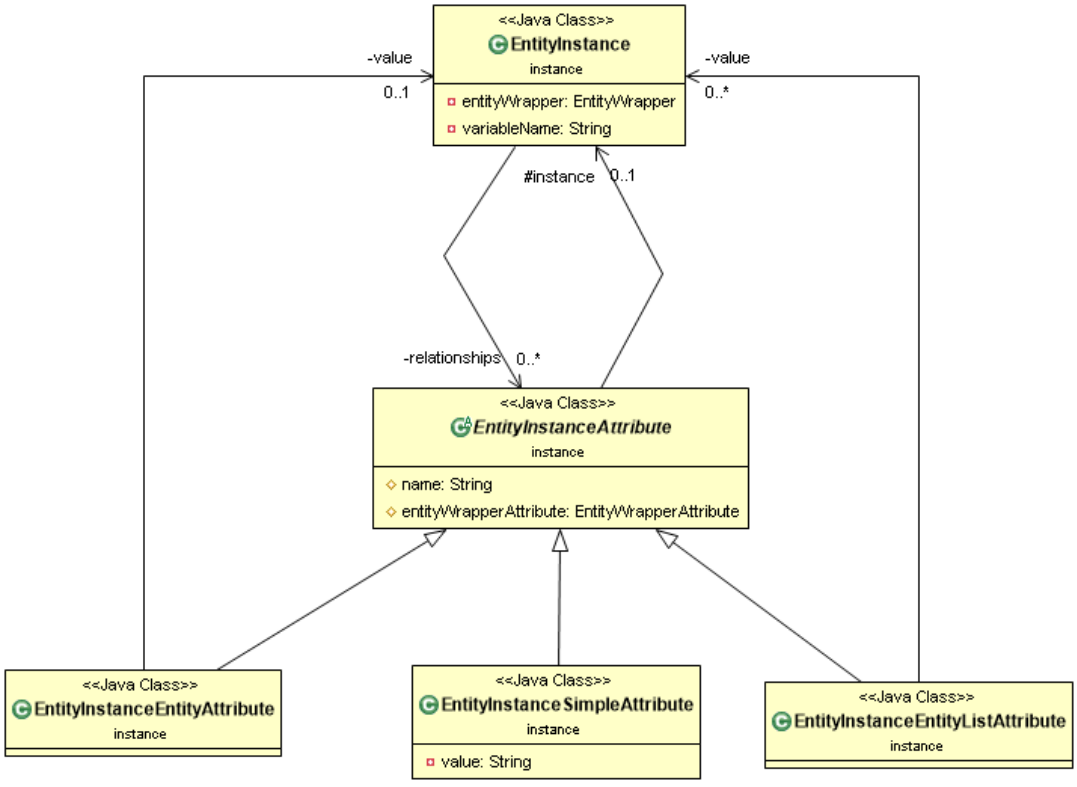
A modellben (4. ábra) a központi elem az *EntityWrapper*, amely a beépített Java *Class* osztályt egészíti ki az *Object Adapter* tervezési minta alapján (2). Egy entitásosztály rendelkezhet adattagokkal, amelyeknek az egyik változata az egyszerű attribútum. Az attribútum specializált változatai az egy-egy, 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.

2.3 Metamodell az objektumok generálásához

Az egyes objektumok hatékony generálásához szükséges a modell kiegészítése egy más szemléletmóddal. Ebben az esetben (5. 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.



5. 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.

2.4 Generálási stratégia

A *Strategy* tervezési minta (2) alapján megtervezett generálási folyamat részeként meghatároztuk azokat a magas szintű lépéseket, amelyek a generáláshoz szükségesek (**Hiba! A hivatkozási forrás nem található.**. ábra).

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 (2).

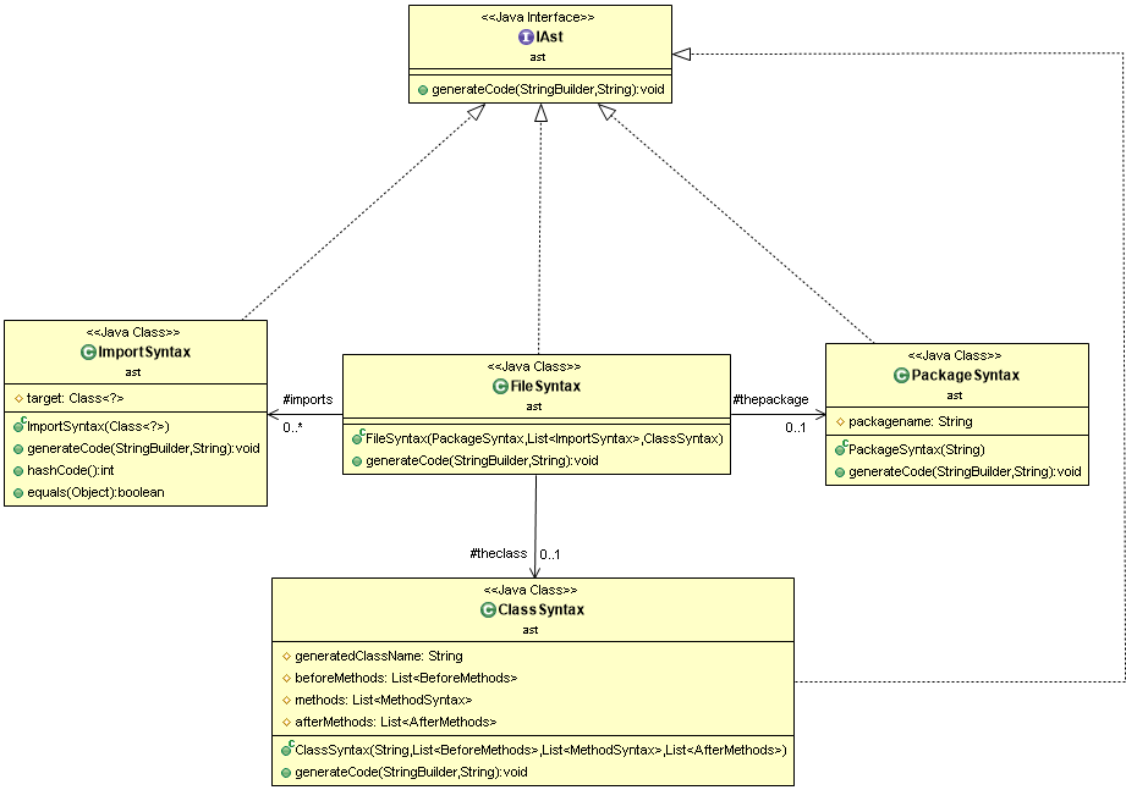
A megadott példastratégián jól látható, hogy a hatékony megvalósításhoz segédmetódusok szükségesek, de ezek létrehozása már a stratégia tervezőjén és megvalósítóján múlik.

2.5 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 (6. ábra):

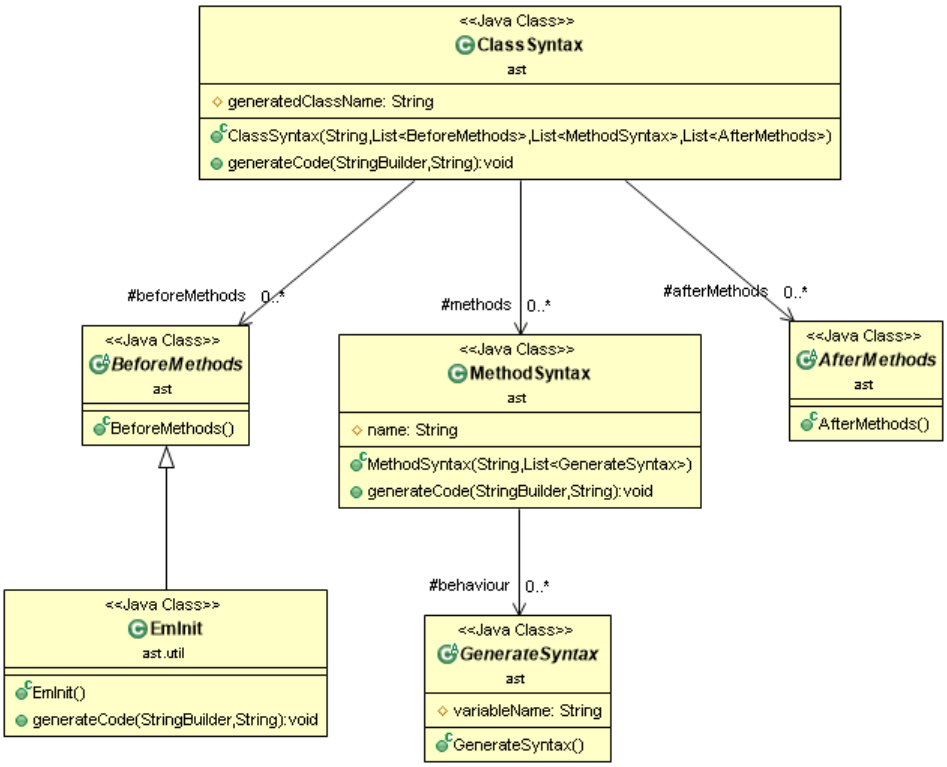
- **Gyökérelem** (IAst): IAst interfész, amely a kódgeneráláshoz szükséges metódust tartalmazza. A szintaxisfa minden eleme futtatható forráskódra alakítható.
- **Java fájl** (FileSyntax): Java nyelv alatt a legkisebb fordítási egység. Tartalmaz egy csomagdeklarációt, tetszőleges számú import deklarációt 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.



6. ábra Absztrakt szintaxisfa összetett elemei

Az absztrakt szintaxisfa szerinti Java osztály felépítése (7. ábra):

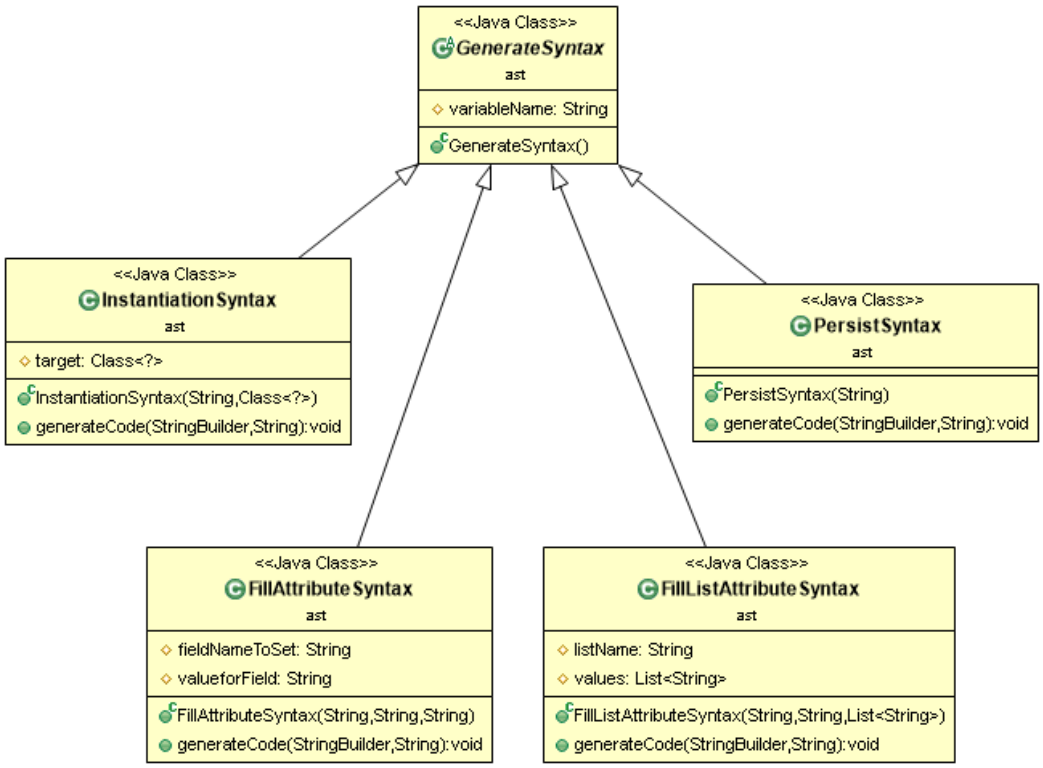
- **Java osztály** (ClassSyntax): Jelen esetben az entitás példányok létrehozási logikájának tartalmazója. Esztétikai szempontból három fontos részét különböztetjük meg: a metódusok előtt megjelenő speciális kódokat, a generátor metódusokat és az ezek után megjelenő részeket.
- **Metódusok előtti részek** (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.



7. ábra Absztrakt szintaxisfa osztály generálásához

A generálás során alkalmazott atomi műveletek (8. 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.



8. ábra Absztrakt szintaxisfa generálási lépésekhez

A szintaxisfa felépítése után inorder bejárással (3) kapható meg a generált forráskód. Az inorder bejárás során először a baloldali részfát, majd az adott elemet és végül a jobboldali részfát látogatjuk meg.

3 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.

Irodalomjegyzék

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

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

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

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

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

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

7. **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.

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

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

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

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

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

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

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

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

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

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

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

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