Fejlesztés

Dolgozatom keretein belül egy DSL nyelvet kell elkészítenem, mellyel karakterkódolások közötti konverziót lehet definiálni. A DSL nyelvben meg lehet adni, hogy melyik karakterkódolásból kiindulva melyik karakterkódolásba szeretnénk eljutni, és ehhez az egyes kódpontok hogyan változnak meg, vagyis melyik kiinduló kódpont melyik célállapoti kódpontnak felel meg. Az egyszerűség érdekében a kódpontokat nem számmal, hanem hexadecimális kóddal írjuk le.

Itt látható egy kezdeti vázlat a DSL nyelv szintaxisához:

source someEncodingName {

alias anAliasName

// illegalChar 0xffee

// this is a comment

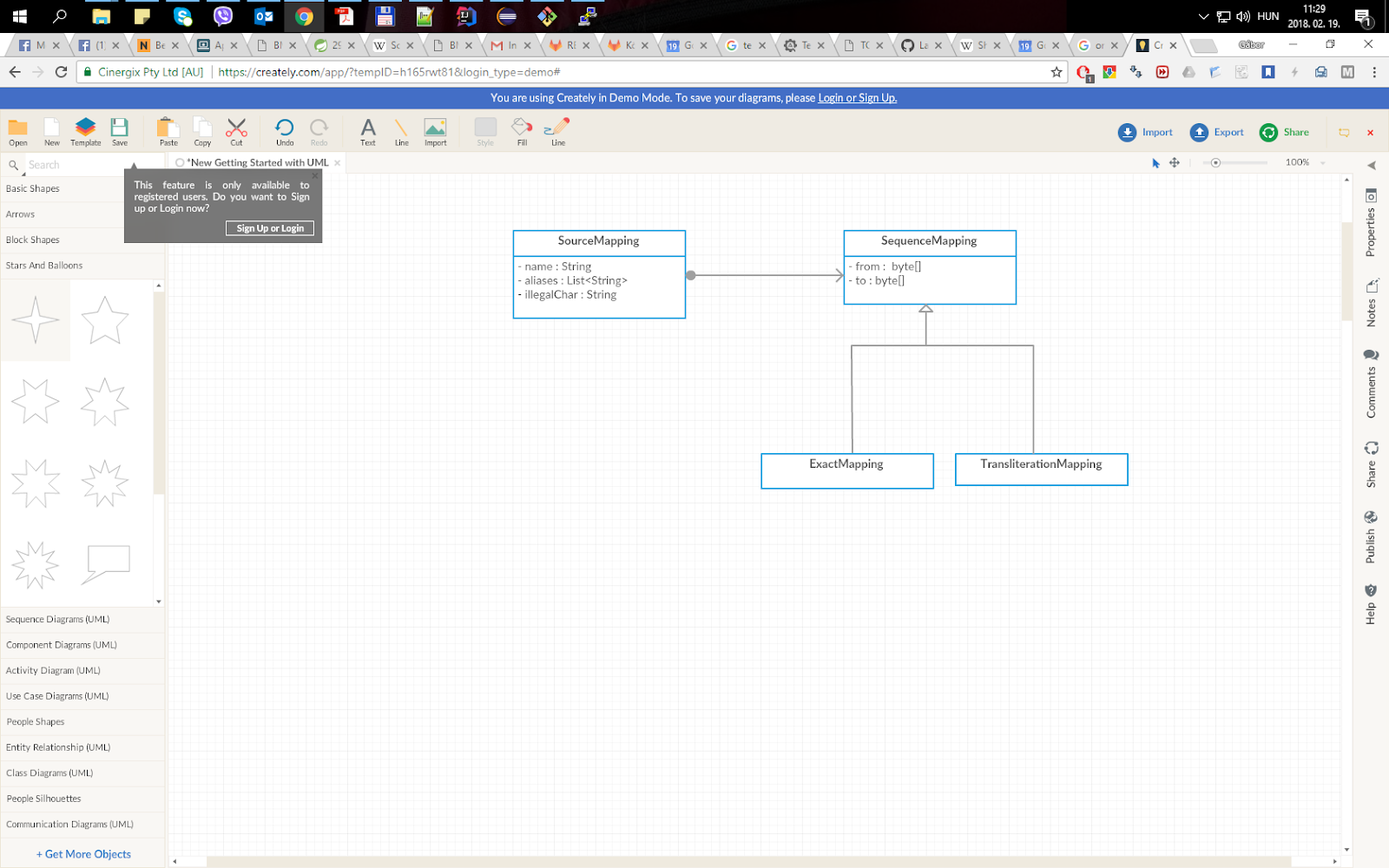
0x0001 = 0xff01 // this will be an exact mapping

0x0002 ~ 0xff02 // this will be a transliteration mapping

// ...

}

A DSL nyelvnek, mint metamodellnek itt látható egy kezdeti diagramja:



Legelőször a saját DSL nyelvemet kellett elkészítenem. Egy új Xtext projekt létrehozása után az EncodingLang.xtext fájlban készítettem el a DSL-em nyelvtanát. Mint minden Xtextes DSL-ben, a sajátomban is Model nevű objektumot kell létrehozni, amely a nyelv többi részét tudja tárolni. Látható, hogy a Model objektum egy elements nevű listában tárolja a SourceMapping nevű objektumokat, a csillag jelzi, hogy legalább egy darabot.

grammar org.xtext.example.EncodingLang with org.eclipse.xtext.common.Terminals

generate encodingLang "http://www.xtext.org/example/EncodingLang"

Model:

(elements+=SourceMapping)\*;

A SourceMapping objektum megadja, hogy egy karakterátkódolásnál, mely kódolásból indulunk ki, annak milyen más elnevezései vannak, és hogyan kell konvertálni egy tetszőleges célkódolásba. A DSL-ben egy SourceMapping írásakor a source kulcsszót kell írni előbb, melyet az idézőjelek jeleznek. Majd egy ID-t ami, a SourceMapping name attribútumába lesz tárolva. Az ID egy ún. terminál (lásd később). Ezután felsorolhatjuk az Aliasokat, és a Conversionöket.

SourceMapping:

"source" name=ID "{"

(aliases += Alias)\*

(conversions += Conversion)\*

"}"

;

Egy Conversionben megadható, hogy mi lesz a kívánt célkódolás, és adott kiinduló kódolás esetén, az egyes kódpontok, hogy változnak meg. A name feltüntetése után írhatjuk le a konkrét konverziókat, melyek objektumainak neve Mappings és a mappings nevű listában tárolódnak el.

Conversion:

"target" name=ID "{"

(mappings += Mapping)\*

"}"

;

Egy Mappings egy konkrét kódpontokat párosít össze, megadva, hogy a kiinduló karakterkódolásban egy kódpont célkódolásban melyik kódpontnak felel meg. Kétféle Mappingot különböztetek meg. Az egyik az ExactMapping, melynek írásakor = karakter, a másik a TransliterationMapping, mely esetben ~ karakter használatos. A két Mapping közötti különbség a jelentésben van: az ExactMapping 1-1 kapcsolatban álló kódpontokat kapcsol össze, míg a TransliteraionMapping a problémásabb párosításoknál használatos. Például ha egy karakternek nincs benne a célkódolásban, így azt egy hozzá hasonló karakterhez kell párosítani. Programozási szempontból nem kezelem másként a két Mappingot, azonban a későbbi bővítés érdekében mégis két objektumot hoztam létre. A kódban lévő INPUTCHAR, az ID-hoz hasonló terminál (lásd később).

Mapping:

ExactMapping | TransliterationMapping;

ExactMapping:

from=INPUTCHAR "=" to=INPUTCHAR;

TransliterationMapping:

from=INPUTCHAR "~" to=INPUTCHAR;

A SourceMapping egyik attribútuma az Aliasok listája. Egy Alias objektum a nyelvben egy alias kulcsszóból és egy ID terminálból áll.

Alias:

"alias" name=ID;

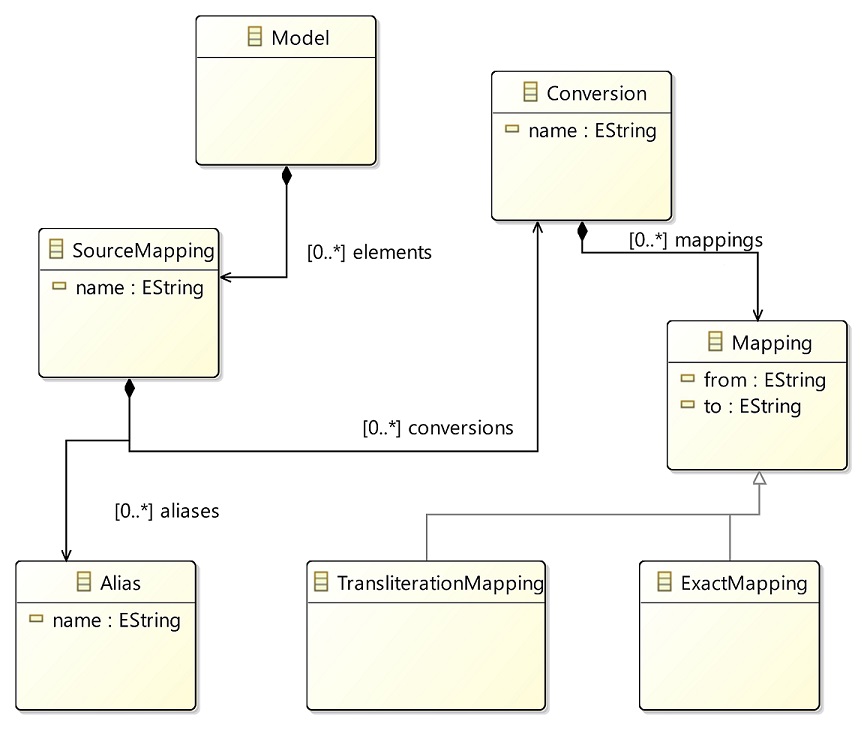
Egy nyelvtan készítésekor terminálok használata legtöbbször elengedhetetlen. A DSL nyelvünkben megírt program szöveges inputot jelent egy parser számára. A parser a kódban szereplő karakterláncoknak és sztringeknek próbál valamilyen logikai jelentést adni. A fenti objektumoknál ez a feladat nem volt nehéz a kulcsszavak miatt. De egyéb sztringeket már nem lehet objektumokhoz rendeli, mert annyira függetlenek tőlük. Ilyen például egy komment a programban. Ezeket a karakterláncokat termináloknak nevezzük, és ezeket a terminálokat lehet hozzárendelni az objektumok attribútumaihoz. Egy terminál implementálásakor meg kell határozni, hogy mely konkrét karakterek tartoznak hozzá. Mivel a legtöbb terminál igen gyakori, az Xtextben ezek már implementálva vannak, például az ID, az INT, a STRING, a DOUBLE, SL\_COMMENT (single line comment), ML\_COMMENT (multi-line comment) vagy a WS (white space). Az én DSL nyelvemben egy új terminált is létre kellett hoznom, melynek neve INPUTCHAR lett. Egy ilyen karakterlánc egy nullával kezdődik, x-szel folytatódik, majd utána páros számú hexadecimális számrendszerbeli karakter következik, legalább két darab, kis és nagybetű között nem teszünk különbséget. A terminál segítségével a karakterkódolásokban használt kódpontokat lehet hexadecimális formában leírni. Például az ASCII hetvenötös kódpontú ’K’ karaktere 0x4B, vagy akár 0X4b formában is leírható.

terminal INPUTCHAR:

'0'('x'|'X')(('0'..'9'|'a'..'f'|'A'..'F')('0'..'9'|'a'..'f'|'A'..'F'))+

;

Ezzel elkészült a DSL nyelv nyelvtana. A nyelvtan elkészítéséhez használt programnyelv az Xtext saját nyelve. Ebből a nyelvtani leírásból az Xtext képes EMF modellt generálni, és sok jelentős Java osztályt készíteni. Az EMF-beli Ecore modellhez képes voltam egy nézetet készíteni, melyben grafikusan látható a modell diagramja.



A diagram tartalmazza az összes implementált nyelvtani objektumot, és a hozzájuk tartozó attribútumokat. Ezek már ténylegesen elérhető és használható Java osztályok. Fontos megjegyezni a Mapping osztály. Itt ugyanis nem volt deklarálva semmilyen attribútum, de mivel egy Mapping objektum vagy ExactMapping vagy TransliterationMapping lehet, és mivel mindkét leszármazott tartalmazta és from és to attribútumokat, ezek az attribútumok ki lettek emelve az ősosztályba. Ezt, és a többi tervezői döntést az Xtext önállóan, emberi beavatkozás nélkül hozta meg.