Debreceni Egyetem
Informatikai Kar

SZAKDOLGOZAT

Programgenerálás XML alapján

Témavezető:
Espák Miklós
egyetemi tanársegéd

Készítette:
Gyenge Csaba
programtervező informatikus
szakos hallgató

Debrecen
2008

TARTALOMJEGYZÉK

1. BEVEZETÉS

Szakdolgozatom témája „Programgenerálás XML alapján”. A témát Espák Miklós tanár Úr biztosította, melyet ketten dolgoztunk fel Laczkó Sándor tankör társammal. A téma bemutatásához szükséges szoftver fejlesztését szintén közösen végeztük Laczkó Sándorral. Ez úton szeretném megköszönni mindkettőjük együttműködő munkáját.

Az informatika a XXI. században rohamosan fejlődik. A felhasználók száma napról napra növekszik, akiknek nagy része egyszerűen és gyorsan szeretné igényeit kielégíteni. Ez szükségessé teszi felhasználóbarát, könnyen kezelhető alkalmazások fejlesztését. Ezek a szoftverek a felhasználó és a számítógép közötti kommunikációt grafikus felület segítségével valósítják meg.

Ez a gyors növekedés olyan intelligens szoftverek meglétét teszi szükségessé, melyek a feladat körültekintő meghatározását és a sarokpontok definiálását követően minimális erőfeszítés és hozzáértés mellett hamar előállítják a programozó igényét kielégítő grafikus felhasználói felületet. A témát egy ilyen feladatot ellátó program fejlesztésén keresztül kívánom bemutatni. Az előállított szoftver a programozók számára testreszabható segítséget nyújt egy olyan grafikus felület kialakítására, melyen keresztül közvetlen kapcsolat létesül a fejlesztendő alkalmazások hátteréül szolgáló adatbázisok és a végfelhasználók között.

A szoftver fejlesztése során szükség volt az XML alapos tanulmányozására valamint a Java nyelv XML feldolgozásához nyújtott lehetőségeinek megismerésére. A Java programozási nyelv lehetőséget biztosít mind az XML dokumentumok értelmezéséhez, mind pedig azok validálásához – szintaktikai ellenőrzéséhez. Természetesen ehhez egyéb technikák megismerésére volt szükség, mint például az XML sémák létrehozásának és azok felhasználásának technikájára. Ezek a sémák biztosítanak alapot az XML dokumentumok ellenőrzéséhez. Mindezek mellett dolgozatomban kitérek még arra, hogy hogyan próbáltam a kifejlesztett programot a célközönség – vagyis a programozók – számára egyszerűen és könnyen alkalmazhatóvá tenni. Erre egyrészt az elterjedt – és a Java programozók által előszeretettel használt – Ant eszköz biztosított lehetőséget. Dolgozatom végén, pedig kitérek a fejlesztett alkalmazás rövid technikai ismertetésére, egyes szakaszainak részletes kifejtésével.

2. Az XML

Az XML, akár csak a HTML, az SGML-ből (Standard Generalized Markup Language) leszármaztatott jelölő nyelv. Az SGML nyelv bonyolultsága és számos hátránya miatt felmerült az igény az SGML egyszerűsített verziójának létrehozására, amely többek számára vonzóvá tenné az általánosított jelölés megközelítését. Az 1996-ban Tim Bray által kifejlesztett Extensible Markup Language (Kiterjesztett Jelölőnyelv), vagy közismertebb nevén XML, a World Wide Web Consortium (W3C) által ajánlott általános jelölő nyelv **[1][4]**. Használatával a követelményeknek megfelelően, egyedi alkalmazásainkhoz speciális célú leíró nyelveket kaphatunk kézhez, melyekkel könnyen megvalósítható az adatok leírása strukturált szöveg formájában és információk megosztása.

2.1. Az XML „tízparancsolata”

A nyelv tervezésének első lépése – mint ahogyan az a tervezéseknek általában – a követelmények felállítása, azok meghatározása volt. A tervezők tíz olyan célt tűztek ki, melyet az új nyelvnek mindenképpen ki kellett elégítenie **[2][3]**:

- Könnyű használhatóság a gyors elterjedés érdekében. Bizonyos SGML-ben lévő szabályokat szándékosan nem vettek át a tervezők, így egyszerűsítve a nyelvet. Egyik ilyen például, hogy az XML DTD nélkül is alkalmazható.
- Az XML-nek sokféle alkalmazást kell támogatnia. Széleskörű információ leírására alkalmas, köszönhető ez egyrészt a dokumentumban tárolt információk hierarchikus szerkezetének. A megjelenését követő három éven belül az XML már elterjedt, alapvető technológiává vált, azóta pedig mindez csak fokozódott.
- Az XML-nek kompatibilitást kell mutatnia elődjével, az SGML-lel. Az XML-t az SGML részhalmazaként definiálták, így biztosítva van, hogy az XML dokumentum mindenképpen feldolgozható legyen SGML eszközök segítségével is. A nyelvben jelentkező felesleges, vagy nem oda tartozó részek ennek a visszafele kompatibilitásnak köszönhetőek.

- Könnyen lehessen XML feldolgozó programokat készíteni. Mivel jóval egyszerűbb elődjénél, így az XML-hez könnyebb feldolgozó alkalmazást fejleszteni.
- Az XML-ben az opcionális lehetőségek számát minimumon, lehetőleg nullán kell tartani. Az XML-ben ugyanazt a hatást ritkán lehet többféle módon elérni, így a cél teljesítettnek tekinthető.
- Emberek által olvasható formátumúak legyenek az XML dokumentumok. Mivel az XML az SGML-en alapszik – mely szöveges formátumú –, így ember által olvasható.
- Az XML tervezésének gyorsan kell végbemennie. Az XML 1.0 elfogadott ajánlása másfél év után jelent meg. A munka semmiképpen nem mondható lassúnak, tehát a cél teljesült.
- Az XML szabványnak formálisnak és tömörnek kell lennie, az egyes félreértések elkerülésének érdekében.
- Könnyen lehessen létrehozni XML dokumentumokat. Az egyszerű ASCII szövegek létrehozására képes programok segítségével már előállíthatunk XML állományokat, illetve számos XML szerkesztő létezik, melyekkel a folyamat gyorsítható.
- A tömörségnek minimális jelentősége van, a lényeg az olvashatóság. Az XML nem rendelkezik az SGML-ben fellelhető minimalizálási lehetőségekkel. Minden jelölőelemnek jelen kell lennie, így segítve elő a könnyű támogathatóság elvét.

2.2. Előnyök

A precíz tervezésnek és a követelmények korai meghatározásának köszönhetően az XML nyelvnek számos előnyös oldala létezik. Ezek az előnyök főként az előzőekben ismertetett tíz kitűzött cél eredményei. Néhány előny például **[4]**:

- Az XML dokumentum egyszerű szöveges fájl, így bárki számára olvasható.
- Számos karakterkódolási eljárást támogat, így lehetőség nyílik bármely információ tetszőleges nyelven való közlésére.
- Az XML nyelv szerkezeti felépítéséből és kötött szintaktikájából adódó egyértelműség eredményeképpen az XML formátum öndokumentáló.
- Az XML dokumentumok struktúrája szinte bármely logikusan csoportosítható adat tárolását lehetővé teszi, így a legtöbb adatstruktúra ábrázolására alkalmas.

- Szigorú szintaktikus és elemzési követelményeket támaszt, így biztosítva, hogy az elemzési algoritmus hatékony, egyszerű és ellentmondásmentes maradjon.
- Egyedi dokumentumtípus definíció létrehozásával saját jelölőnyelv alakítható ki.
- Több mint tíz éve használják, így sok tapasztalat és széles eszközkészlet áll rendelkezésre.
- Az egyszerű szöveges formátum licencektől és egyéb jogi korlátozásoktól mentes.

2.3. Hátrányok

Az előzőekben említett tíz pontban összeszedett követelmények természetesen nem csak és kizárólag előnyöket vontak maguk után. A konkrét célok elérésének érdekében ugyanis egyes területeken kompromisszumokat kellett kötni. E miatt az XML nyelvnek megvannak a maga hátrányai is **[4]**:

- A redundáns szintaxis nehezítheti a fejlesztést, az emberi olvashatóságot és az egyes XML értelmezők hatékonyságát. E mellett nagyobb tárolási költséggel jár, ami megnehezíti az XML dokumentumok átvitelét korlátozott sávszélesség esetén. Bizonyos esetekben az adatátvitel során alkalmazott tömörítés csökkentheti ennek a problémának a súlyosságát.
- Számos homályos és felesleges rész található az XML szintaxisában, mely az SGML-lel való kompatibilitás megőrzésének köszönhető.
- Nincs támogatva az adattípusok túl széles köre. Ez megnehezítheti bizonyos adatok megfelelő formában való kinyerését a dokumentumból, illetve plusz munkával jár a feldolgozó alkalmazás részéről.
- Nincs lehetőség a dokumentum egyes részeinek közvetlen elérésére és frissítésére.
- Nem hierarchikus adatstruktúrák modellezése igen nehéz és bizonyos esetekben komoly erőfeszítéssel járhat.
- Néha szintén nehéz feladat az XML nyelv, relációs illetve objektumorientált paradigmához kötése, habár számos segédeszköz született már az utóbbi években.
- Nincsen egyszerű módja nagy méretű bináris fájlok (képek, hangok) ábrázolásának.
- Az egyszerű szöveges formátum nem teszi lehetővé a tárolt adatok titkosítását. Ehhez külön stratégiát kell kiépíteni az egyes feladatok megvalósítása során.

2.4. Szintaktikai egységek

Mint minden más nyelvnek, az XML-nek is megvannak a saját szerkezeti részei, szintaktikai egységei. Ezekből a jól elhatárolható szerkezeti elemekből, egységekből épül fel egy-egy konkrét XML dokumentum.

2.4.1. Elem

Az XML dokumentumok elemekből (element) épülnek fel. Az elem úgynevezett jelölőkódokkal van körülvéve, melyek arra szolgálnak, hogy a rendszer és a feldolgozó programok érzékelni tudják az adott elem határait, vagyis annak elejét és végét. A jelölőkódok négy fajtája az alábbi listában szerepel:

- *STAGO* (*Start Tag Open*) - `<` : a kezdő jelölőelem nyitó eleme.
- *STAGC* (*Start Tag Close*) - `>` : a kezdő jelölőelem záró eleme.
- *ETAGO* (*End Tag Open*) - `</` : a befejező jelölőelem nyitó eleme.
- *ETAGC* (*End Tag Close*) - `>` : a befejező jelölőelem záró eleme.

A jelölőkód kifejezést, a jelölőelemet határoló karakterek megnevezésére használjuk. Két fajta jelölőelem létezik. A kezdő, mely az elem elejét és a befejező, mely az elem végét jelöli. A kezdő és a befejező jelölőelem között található az elemtartalom. A kezdő és a befejező jelölőelem az elemtartalommal együtt alkotja az úgynevezett hordozóelemet.

Az elem neve, illetve bizonyos, az elemhez tartozó tulajdonságok (attribútumok) a kezdő jelölőelem jelölőkódjai között szerepelnek. Az elemnevek hossza nincs korlátozva, természetesen legalább egy karakterből kell állniuk. Betűvel, aláhúzással, vagy kettősponttal kell kezdődnie, majd tartalmazhat még az előző karaktereken kívül számjegyet, pontot és kötőjelet. A nevek kisbetű-nagybetű érzékenyek.

Az egyes elemek természetesen egymásba ágyazhatóak, vagyis az elemtartalomban szerepelhetnek újabb elemek, melyeket gyermekelemeknek hívunk. Így alakítható ki a kívánt hierarchikus szerkezet.

Előfordulhatnak olyan esetek, hogy egyes elemek üresen maradnak. Ekkor a kezdő és befejező jelölőelem között semmi sem szerepel, vagyis az elemtartalom üres. Üreselemeket ábrázolhatunk a szokásos módon, `<elem></elem>`, vagy egy rövidebb és szemléletesebb módon is: `<elem/>`. A gyakorlatban az utóbbi írásmód terjedt el, mivel egyrészt jelzi a szövegjelölést végző személynek, hogy az elemtartalom üres és minden esetben annak is kell maradnia, ezáltal könnyebb olvashatóságot kölcsönöz az adott dokumentumnak, másrészt nagy dokumentumok esetén kisebb fájlméretet eredményez.

2.4.2. Attribútum

Igazából ez az egység az elemmel – mint szintaktikai egységgel – alkot szoros kapcsolatot, tartalmilag viszont ez a szerkezeti elem is egy külön pontot érdemel. Segítségével olyan egyéb információt írunk le, mely hozzátartozik az elem adott előfordulásához, jellemzi azt. Tulajdonképpen az elem paramétere, amely módosítja, illetve pontosítja annak jelentését.

Ha egy elemet úgy definiálunk, hogy van attribútuma, akkor az attribútum nevét és értékét az elem előfordulásának kezdő jelölőelemében adjuk meg, mégpedig egyenlőségjellel elválasztva őket, az értéket pedig idézőjelek vagy aposztrófok közé téve. Az attribútumokat és az elem nevét whitespace karakternek illetve azok sorozatának kell elválasztania egymástól.

Az adott elemhez tartozó attribútum listában az attribútum nevének egyedinek kell lenni. Az attribútum nevekre érvényes formai megkötések megegyeznek az elemneveknél ismertetettekkel és szintén kis- és nagybetű érzékenyek.

2.4.3. Egyedhivatkozás

Lehetőségünk van az XML dokumentum valamely részét azonosítóval ellátni. Az egyed (entitás, entity) a dokumentumnak ezt az azonosított, névvel ellátott részét jelenti. Tehát az entitások olyan azonosítók, melyek más szövegelemekre mutatnak. Segítségükkel az összetevők hordozhatóvá válnak, megoszthatók és könnyebben kezelhetők lesznek. Az egyedek deklarálása a következő fejezetben kerül ismertetésre, mivel a deklarációk nem tartoznak szorosan az XML dokumentumokhoz, így nem mondhatóak szintaktikai elemnek sem.

Az úgynevezett egyedhivatkozás olyan egyszerű jelölésszerkezet, melynek hatására a hivatkozás helyére behelyettesítődik az éppen hivatkozott egyed által mutatott érték. A hivatkozás formailag egy & jellel kezdődik, pontosvesszővel végződik és e között a két karakter között kell megadnunk a hivatkozott egyed nevét, melyet az előzőekben deklaráltunk.

Néha szükség lehet olyan speciális karakterek elhelyezésére egyes XML dokumentumokon belül, melyek normál esetben nem szerepelhetnek bennük, vagy egyszerűen ezeket a karaktereket nem tartalmazza a billentyűzetünk. Ennek egyik megvalósítási módja az úgynevezett karakterhivatkozás alkalmazása. Segítségével karakterentitásokat használhatunk fel a dokumentumon belül. A karakterentitásokra vonatkozó hivatkozások az egyszerű egyedhivatkozásokhoz hasonlóan & jellel kezdődnek, majd kettős kereszt, vagy kettős kereszt és azt követő x karakter szimbólummal folytatódnak. A végüket szintén pontosvessző jelöli. Az egyednév helyett, pedig egy számot kell megadnunk, mely az adott karakter kódját, vagyis a karakterkészletben elfoglalt helyét jelenti. Az első esetben decimális értékkel hivatkozhatunk az adott karakter kódjára, míg utóbbinál hexadecimális formában tehetjük ezt meg.

Néhány, gyakran használt speciális karakterhez létezik előre definiált általános egyed is. Az ezekre való hivatkozások használata jóval ember közelibb megoldást nyújt a speciális karakterek használatára, mivel azok karakterkódja helyett beszélő neveket használhatunk. Ezen karakterbeillesztési mód sokak számára ismerős lehet a HTML-ből. Ilyen hivatkozás például az `>` (greater than) mely a >, vagy az `<` (less than) ami a < karaktert jelöli.

2.4.4. Megjegyzés

Ez egy olyan karaktersorozat mely nem az értelmezőnek, hanem a dokumentumot olvasó személynek szól **[5]**. Segítségével tetszőleges magyarázó szöveget helyezhetünk el a dokumentum bármely részén. A megjegyzés szövege `<!--` és `-->` között szerepel. A megjegyzés szövegében nem állhat `--` karaktersorozat. Az utánuk található szöveget az értelmező program figyelmen kívül hagyja egészen az adott sor végéig.

2.4.5. Nem elemzett karakteres adat

Előfordulhatnak olyan esetek, amikor az XML dokumentumunkban lévő adatok között jelöléshatároló karaktereket is akarunk használni. Ekkor az egyik mód, hogy a már említett beépített karakterentitás hivatkozásokat használjuk, viszont sok esetben nagy tömegű adatunk miatt nehézkes és problémás lenne az összes lefoglalt karaktert a neki megfelelő entitással helyettesíteni. Így aztán hasznos lenne, ha az XML dokumentumunk adott szövegrészeit megjelölhetnénk a feldolgozó számára, hogy ignorálja a szöveg ezen tartományait. Erre ad megoldást a CDATA-szakasz. A szakaszban elhelyezett adatokat a feldolgozó egyszerű karakteres adatnak tekinti, és nem értelmezi annak tartalmát. A CDATA-szakasz kezdetét a `<![CDATA[` karaktersorozat jelzi, míg végéről a `]]>` karaktersorozat informálja az értelmezőt.

2.4.6. XML deklaráció

Az XML deklaráció a dokumentum elején helyezkedik el, mindent megelőzve. Magáról a dokumentumról szolgál fontos információkkal a feldolgozó alkalmazások számára. A deklaráció `<?` karakterekkel kezdődik és `?>` karakterekkel végződik. Ideális esetben három paraméterből áll, melyek közül kettő használata opcionális.

A kötelező `version` paraméter azt mondja meg a feldolgozó programnak, hogy a dokumentum az XML melyik verziójával van összhangban. Mivel a jelenlegi alkalmazások leginkább az 1.0-ás változatot támogatják, ezért általában ezt a verziószámot adjuk meg a deklaráció ezen paraméterének.

Az opcionális `encoding` a dokumentum készítésénél használt karakterkódolási sémát takarja. Amennyiben nem adjuk meg ezt az attribútumot, abban az esetben az XML feldolgozó feltételezi, hogy UTF-8 vagy UTF-16 karakterkódolást használunk. Amennyiben valamilyen más karakterkódolást használunk, kötelező megadnunk azt.

A szintén opcionális `standalone` paraméter azt jelzi, hogy a dokumentum feldolgozást befolyásolja-e valami külsőleg meghatározott deklarációkészlet. A paraméter `yes`, vagy `no` értéket vehet fel attól függően, hogy a dokumentumunk nem egyedülálló és függhet külső deklarációktól, vagy sem. A paraméter alapértelmezett értéke `no`.

2.4.7. Dokumentumtípus deklaráció

Egy XML dokumentum elején megadható, hogy az adott dokumentum milyen típusú, vagyis milyen elemek és entitáshivatkozások szerepelhetnek benne. Mindezt a dokumentumtípus definíció (DTD, Document Type Definition) segítségével adhatjuk meg. Megadása viszont nem kötelező az XML esetében. Amennyiben mégis meg van adva, abban az esetben az XML deklarációt kell követnie. Az ilyen típusú deklarációk mindig `<!DOCTYPE` karaktersorozattal kezdődnek és `>` karakterrel végződnek.

A `DOCTYPE`-ot egy név követi, mely a dokumentum gyökércsomópontját azonosítja, vagyis annak az elemnek a neve szerepel itt, mely az XML dokumentumban szereplő összes elemet tartalmazza.

A nevet egy kulcsszó követ, mely a `SYSTEM` vagy a `PUBLIC` valamelyike lehet. Az előbbi azt jelzi, hogy a DTD egy meghatározott személyhez vagy szervezethez tartozik, míg az utóbbi azt, hogy a DTD-t egy szabványalkotó testület hozta létre és a nyilvánosság számára elérhető.

Végül a külső azonosító kap helyet, ami a `SYSTEM` kulcsszó esetében egy URI-t takar, míg `PUBLIC` használatakor valamilyen publikus azonosítót és egy azt követő URI-t. Az URI (Uniform Resource Identifier) erőforrás azonosítására használt rövid karaktersorozat. Közismert példái a webcímek, más néven URL-ek.

2.5. XML dokumentumok helyessége

Az XML dokumentumok helyességének két fokmérője van. Az elsőt, ami az úgynevezett jól formázottság (well-formed), akkor teljesíti, ha megfelel az XML által támasztott szabályoknak. A másik, ha a dokumentum érvényes (valid). Ekkor a dokumentumot annak funkciói szerint ellenőrizzük, rendszerint egy XML Schema vagy DTD dokumentum alapján.

2.5.1. Jól formázottság (well-formed)

XML-ről csak abban az esetben beszélhetünk, ha legalább jól formázott **[6]**. Egy jól formázott XML dokumentumnak többek között az alábbi kritériumokat kell teljesítenie:

- Egy dokumentumon belül egyetlen gyökérelem állhat. Az XML deklaráció, a feldolgozó utasítások és megjegyzések természetesen megelőzhetik a gyökér elemet.
- Az elemeket nyitó és záró jelölőelemnek kell határolnia, vagy az üreselemeket alkothatja egyetlen üreselem címke.
- Minden elemnél egy adott nevű tulajdonság legfeljebb egyszer szerepel. Megadásuk sorrendje tetszőleges. Az érték megadása kötelező és idézőjel vagy aposztróf között kell feltüntetni.
- Az elemek egymásba ágyazása korrekt, tehát mindegyik nem gyökér elemet másik elemnek kell magában foglalnia.

2.5.2. Érvényesség (validation)

Az esetek egy részében az érvényesség (validation) egy további kritérium. Az olyan jól formázott dokumentumot, mely megfelel egy adott sémának, érvényes (valid) dokumentumnak nevezzük.

Egy XML séma az XML dokumentum típusának leírása. Az XML megkötésein túli korlátozásokat tartalmazza az adott típus struktúrájára és tartalmára nézve. Nagy mennyiségű szabványos és szabadalmazott XML séma jelent meg, hogy leírja ezeket a megkötéseket, és e sémák egy része maga is XML alapú. A következő fejezetben megismerkedünk az egyik – régebbi és többször alkalmazott – sémával, a DTD-vel.

Alaposan tesztelt eszközök állnak rendelkezésre, hogy az XML dokumentumot a sémával szemben validálni tudjuk. Ezek az eszközök automatikusan ellenőrzik, hogy a dokumentum megfelel-e a sémában kifejtett megkötéseknek. Vannak olyan eszközök, melyek az XML elemzővel egybeépítve érhetők el, és vannak, melyek külön használhatók.

2.6. XML dokumentumok feldolgozása Java-val

Mivel az egyes XML dokumentumok adatokat, fontos információkat tárolnak, ezért természetesen szükséges azok feldolgozása, értelmezése. Ezen cél megvalósításának érdekében majdnem minden program egy úgynevezett XML elemzőt (XML parser) használ fel. Az XML elemzők többek között képesek például beolvasni a dokumentumokat, feloldani az egyes karakter-hivatkozásokat, ellenőrizni a jólformázottságot, validálni az adott dokumentumot. Ezen kívül még sok más feladatot vesznek le a fejlesztő válláról. Alapjában véve két fő módszer – és ezeket megvalósító egy-egy API – áll rendelkezésünkre az XML dokumentumok Java-val történő feldolgozására. Az egyik a SAX (Simple API for XML), míg a másik a DOM (Document Object Model). Természetesen mindkettőnek számtalan verziója született már, sőt rajtuk kívül egyéb sajátos értelmezők is napvilágot láttak már.

2.6.1. DOM – Document Object Model

A DOM modell, a teljes XML dokumentum reprezentációját jelenti tárban elhelyezett objektumok által **[7]**. Amint bekerülnek az objektumok a tárba, és felépül a DOM fa – mely magát az XML dokumentumot ábrázolja (a fa egyes elemei objektumok, melyek egy-egy XML elemet reprezentálnak) – a fejlesztő kedve szerint tud navigálni az elemek között, így dolgozva fel a dokumentum szükséges részeit, így érve el az elemek tartalmát és egyes attribútumainak konkrét értékét. Ez a technika nagyfokú rugalmasságot biztosít a fejlesztők számára, mivel az elemek feldolgozási sorrendje nem szükségszerűen lineáris, vagyis nem kell, hogy egybeessen az XML dokumentumban lévő elemek sorrendjével. Azonban ennek a rugalmasságnak megvan az ára, mégpedig az, hogy sok esetben hatalmas memóriaterületet igényel, és komoly igényeket támaszt a processzor felé az, hogy a dokumentum reprezentációja teljes egészében a tárban van a feldolgozás időtartama alatt. Természetesen kisméretű dokumentumoknál ez a probléma kevésbé jelentkezik, viszont hamar megnőhet a processzor és memória kapacitással szemben támasztott igény nagyméretű dokumentumok esetében.

A következőkben lássunk egy kódrészletet annak bemutatására, hogy hogyan kell feldolgozni egy XML dokumentumot a DOM API segítségével. Először a `doc.xml` nevű fájlt nyitjuk meg, mint adatfolyam **[8]**. A fájlt a `user.dir` rendszertulajdonság által meghatározott

könyvtárban keresi a JVM. Ezt követően három soron keresztül történik meg a feldolgozó létrehozása és magának a DOM modellnek a felépítése. Végezetül pedig a `sampletag` nevű elemek egy listáját érjük el a felépített DOM fán keresztül.

```
InputStream fileInputStream = new FileInputStream(new File("doc.xml"));
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(fileInputStream);
NodeList nodes = document.getElementsByTagName("sampletag");
```

2.6.2. SAX – Simple API for XML

A SAX az XML feldolgozó Java API-k „etalonja” **[1][7]**. Ez a legösszetettebb és legjobban működő API. Használatával egy tökéletes validációra képes értelmezőt kapunk kézhez rengeteg konfigurálási lehetőséggel. A SAX egy eseményvezérelt értelmező. Ez azt jelenti, hogy nem hoz létre egy reprezentációs modellt, mint a DOM, melyet aztán tetszőleges módon bejárhatunk, hanem a dokumentum feldolgozása lineárisan történik. Egy adatfolyamként (stream) kezeli a dokumentumot, és az értelmezés során bizonyos pontokhoz érve különböző események váltódnak ki, melyekre a programozó vagy reagál, vagy pedig figyelmen kívül hagyja azokat. Nincsenek tehát magának az XML dokumentumnak a reprezentálására szolgáló osztályok, hanem az értelmező egy interfészen keresztül szolgáltatja a feldolgozás alatt lévő dokumentumban szereplő adatokat az adott program számára. Ez a módszer a SAX értelmezőnek rendkívüli gyorsaságot és kiváló memóriagazdálkodást kölcsönöz, ugyanis jóval kevesebb memóriát igényel, mint az előzőekben tárgyalt DOM értelmező. A nagy hátránya az, hogy használata bonyolultabb tervezést és implementálást eredményez, mivel nincsenek előre elkészített eszközeink, melyek segítségével a dokumentum szükséges részeit egyszerű módon ábrázolhatnánk és tárolhatnánk. Ezeket a fejlesztőnek kell megterveznie és megvalósítania.

A SAX API használata abban az esetben tanácsos, amikor nagy méretű XML dokumentumok feldolgozása a cél. Ebben az esetben egyrészt kevesebb tárterület szükséges az értelmezés alatt, másrészt lokális fájloknál jelentős sebességnövekedést érhetünk el a DOM API alkalmazásával szemben. Amennyiben nagy szükségünk van az elemek véletlen

elérésére, és a dokumentum jelentős részét használni akarjuk, akkor bonyolultsága miatt tanácsosabb hanyagolni ezt az API-t.

Következzen egy példa az API használatára **[9]**. A dokumentumban lévő adatok egy interfészen keresztül érhetők el. Elsőnek ezt az `org.xml.sax.ContentHandler` interfészt kell implementálnunk. Bevált módszer, hogy inkább az `org.xml.sax.helpers.DefaultHandler` osztályt terjesztjük ki, mivel ez implementálja többek között az említett interfészt is és valójában az egyes implementált metódusok semmilyen műveletet nem hajtanak végre. Így egyszerűbb tehát, ha új osztályunkat a `DefaultHandler` osztályból származtatjuk le, ezáltal a számunkra felesleges metódusokat nem kell implementálnunk.

```
public class MySAXHandler extends DefaultHandler {
      public void startDocument() {
            System.out.println("Dokumentum kezdete.");
      }

      public void endDocument() {
            System.out.println("Dokumentum vége.");
      }

      public void startElement(String uri, String name, String qName,
                  Attributes atts) {
            System.out.println("Elem kezdete.");
      }

      public void endElement(String uri, String name, String qName) {
            System.out.println("Elem vége.");
      }

      public void characters(char ch[], int start, int length) {
            System.out.println(ch.length + " db karakter beolvasva.");
      }
}
```

Ezzel elkészült a saját tartalomkezelő osztályunk, melynek egyes metódusai a feldolgozás alatt bizonyos pontokhoz érve fognak aktivizálódni. A metódusok nevei igen beszédesek. Például a `startDocument()` metódust abban az esetben fogja meghívni az XML feldolgozó, amikor az adott XML dokumentum feldolgozása elkezdődik.

Természetesen még hiányzik az XML dokumentum tényleges feldolgozásáért felelős kódrészlet **[9]**. Az alábbi részlet második sorában hozzuk létre magát az XML feldolgozót, melynek a következő két sorban beállítjuk tartalomkezelőként az előzőleg létrehozott osztályunk egy példányát. Végül az `xr` XML feldolgozó `void parse(InputSource)` metódusának meghívásával kezdetét veszi a `doc.xml` nevű dokumentum tényleges feldolgozása.

```
InputStream fileInputStream = new FileInputStream(new File("doc.xml"));
XMLReader xr = XMLReaderFactory.createXMLReader();
MySAXHandler handler = new MySAXHandler();
xr.setContentHandler(handler);
xr.parse(new InputSource(fileInputStream));
```

3. Document Type Definition

Sok esetben szükség van annak megállapítására, hogy egy adott XML dokumentum érvényes-e. Ekkor kell az úgynevezett XML sémák valamelyikét alkalmazni. Ezek olyan típusdefiníciók, melyek segítségével leírhatjuk, hogy az adott dokumentumtípushoz tartozó dokumentumok milyen elemeket, attribútumokat, értékeket és hivatkozásokat tartalmazhatnak. Ezekkel tudjuk megadni a használatos karakterkészleteket, és azt, hogy az egyes elemek hogyan helyezkedhetnek el a dokumentumban. Az egyik és legősibb ilyen XML séma a dokumentumtípus definíció (Document Type Definition), vagy röviden DTD **[2]**. Ebben a fejezetben a DTD felépítéséről és az egyes szintaktikai egységekről lesz szó.

3.1. Elemdeklaráció

Az elemdeklarációk segítségével új elemet definiálhatunk a DTD-n belül és meghatározhatjuk hozzá a megengedett elemtartalmat. A deklaráció hatására az adott dokumentumtípusú XML dokumentumokban használhatjuk a deklarált elemet, mely aktuális tartalmának illeszkednie kell a megadott elemtartalomra. Például a következő deklarációk egy-egy, a hibergui-mapping XML-ben használható elemet deklarálnak.

```
<!-- A hibergui mapping xml elemeinek deklarációja -->
<!ELEMENT form ((label | edit | code)+, table?)>
<!ELEMENT label EMPTY>
<!ELEMENT edit EMPTY>
<!ELEMENT code (#PCDATA)>
<!ELEMENT table (column+)>
<!ELEMENT column EMPTY>
```

Az elemdeklarációkat – és általánosan a deklarációk összes fajtáját – mindig csúcsos zárójelek fogják közre. A nyitó csúcsos zárójelet követő első karakter, kötelezően egy felkiáltójel, melyet az `ELEMENT` XML szabványban definiált kulcsszó követ kijelölve, hogy elem típusú objektumot deklarálunk.

Maga az elemdeklaráció két fő részből áll. Első helyen egy név szerepel, mely annak az elemnek az általános azonosítóját nevesíti, amelyet éppen deklarálunk. Ezt az úgynevezett tartalmi modell követi. A deklaráció egyes részeit whitespace karakterek – szóköz, tabulátor vagy újsor – választják el egymástól.

3.1.1. Tartalmi modell

A deklaráció utolsó része a tartalmi modell, mely kerek zárójelek között áll. Ez szabja meg, hogy az adott elem a dokumentumokban ténylegesen mit tartalmazhat. A tartalom elsősorban más elemekkel való összefüggéssel, vagy speciális, a szabványban lefoglalt kulcsszavakkal adható meg. Több kulcsszó is létezik.

Az egyik leggyakrabban előforduló kulcsszó a `PCDATA`. Ebben az esetben a deklarálandó elem tetszőleges, az értelmező által feldolgozásra kerülő, elemzendő karakteres adatot tartalmazhat. A szó az angol Parsed Character Data kifejezés rövidítése.

A tartalmi modell egy másik különleges esete az, amikor egy adott elem nem tartalmazhat semmit. Példánkban a `label` egy ilyen elem. Ebben az esetben a tartalmi modelljét az `EMPTY` kulcsszóval adjuk meg.

Amennyiben megengedjük, hogy az elem tetszőleges elemeket tartalmazzon, abban az esetben az `ANY` kulcsszót kell használnunk. Az `ANY`-t tartalmazó elem bármely más, a DTD-ben deklarált elemet tartalmazhat, ám éppen emiatt ritkán alkalmazzák, mivel túl nagy szabadságot ad és egyúttal aláaknázza a dokumentumszerkezet definiálásából adódó előnyöket.

A kulcsszavak után, pedig lássuk a tartalmi modell leggyakoribb esetét. Amennyiben az aktuális elemen belül lehet gyermekelem, akkor a deklaráció rögzítheti a tartalmi modellcsoportot, vagyis megadhatjuk azt, hogy mely elemeket tartalmazhatja, illetve definiálhatjuk azok sorrendjét és előfordulási gyakoriságát is.

Az előző példa `table` elemének deklarációja kimondja, hogy egy táblázat egy vagy több oszlopból áll. Annak jelzésére, hogy a tartalmi modellben megadott elem hányszor fordulhat elő, a deklaráció egy gyakoriságjelzőt – ebben az esetben pluszjelet – használ. Három gyakoriságjelző karakter van. Ezek a plusz, a kérdőjel és a csillag. Jelentésük az ismert reguláris nyelvekben használt karakterek jelentésével nagyjából megegyezik:

- *Pluszjel* (`+`): az adott elem egyszer, vagy többször fordulhat elő.
- *Kérdőjel* (`?`): az adott elem legfeljebb egyszer fordulhat elő, vagyis opcionális.
- *Csillag karakter* (`*`): az adott elem akárhányszor (beleértve, hogy egyszer sem) előfordulhat.

Mivel `form` elemünk tartalmi modellje `((label | edit | code)+, table?)`, ez azt jelenti, hogy több komponenst is tartalmazhat, ezért szükség van arra, hogy megmondjuk, a négy beágyazott elem milyen sorrendben szerepelhet. Ezt a sorrendiséget az összetevők közti csatolójel határozza meg. Két lehetséges csatolójel van, megegyezés szerint a vessző és a függőleges vonal:

- *Vessző* (`,`): az általa összekapcsolt összetevők pontosan olyan sorrendben jelenhetnek meg az XML dokumentumban, ahogyan a deklarációban szerepelnek.
- *Függőleges vonal* (`|`): az általa összekapcsolt összetevők közül csak az egyik jelenhet meg.

3.2. Attribútumlista-deklaráció

Egyes XML elemek rendelkezhetnek tulajdonságokkal, attribútumokkal. Ezek további információval szolgálnak az elemről, vagy annak tartalmáról. Az attribútumokat az elemtől különállóan kell deklarálni. A DTD-kben az attribútumokat az `ATTLIST` deklarációval adhatjuk meg. Egy attribútumlista-deklaráció az alábbi módon épül fel.

```
<!ATTLIST elem-név attribútum-név attribútum-típus alapérték
                  [attribútum-név attribútum-típus alapérték]...>
```

A deklaráció csúcsos zárójelek között szerepel. A nyitó zárójelet felkiáltójel, majd a már említett, szabványban definiált kulcsszó követ, mely jelzi, attribútumlista-deklarációról van szó. Ez után kell megadnunk rendre az elem nevét – melyhez a deklarálandó attribútum tartozik – az attribútum nevét, típusát és végül kezdőértékét. Utóbbi három tetszőleges számban ismétlődhet, melynek hatására újabb és újabb tulajdonságot deklarálunk az adott elemhez. Nézzünk egy példát, mely a `column` elemhez – az elemnév megadása után – soronként egy-egy attribútumot deklarál, összesen négyet.

```
<!-- Attributumok deklarálása a column elemhez -->
<!ATTLIST column
          label CDATA #REQUIRED
          property CDATA #REQUIRED
          align CDATA "left"
          width NMTOKEN "200">
```

A példában jól látszik, hogy az egyes attribútumok nevét az általuk felvehető érték típusa és az alapértelmezett érték követi. A típus egy adott kulcsszó megadásával határozható meg. Ezek a kulcsszavak előre definiált szavak, melyeket a következő listában ismertetek:

- *CDATA*: az attribútum karakteres adat lehet. Az értelmező nem dolgozza fel ezt az adattípust, változatlan formában átengedi az ellenőrzésnél.
- *ENTITY*: az attribútumban egy entitásra vonatkozó hivatkozás található.
- *ENTITIES*: több entitást is meg lehet adni referenciaként. Az egyedeket egy listában, egymástól whitespace karakterekkel elválasztva kell megadni.
- *ID*: az ilyen típusú attribútum egyértelműen azonosítja a dokumentumban azt az elemet, melyhez tartozik. Adott XML dokumentumban szereplő összes ID típusú tulajdonság értékének különbözőnek kell lennie.
- *IDREF*: az attribútum referenciát tartalmaz egy, az adott dokumentumban előforduló ID-ra. (Az ID-ket a dokumentum tartalmának hiperhivatkozásokkal való jelölésére használhatjuk.)
- *IDREFS*: olyan, mint az IDREF, de itt az attribútum, ID-k egy listáját tartalmazza.
- *NMTOKEN*: az attribútum értéke lehet bármilyen szó vagy lexikális elem – szám, betű, írásjel.
- *NMTOKENS*: egymástól szóközzel elválasztott NMTOKEN értékek listája.

- *NOTATION*: az attribútum értéke egy `NOTATION` adattípus, melynek deklarációja a DTD egy másik pontján helyezkedik el.
- *Értéklista*: ez egy olyan eset, amikor nem lefoglalt kulcsszót kell megadni típusként, hanem a lehetséges értékek egy listáját. A listát zárójelek között kell elhelyezni, ahol a lista egyes elemei függőleges vonallal vannak elválasztva egymástól. A konkrét érték mindig a felsorolás valamelyik eleme kell, hogy legyen.

Az egyes attribútumok deklarálásánál az utolsó rész azt jelzi az értelmező program számára, hogy mit kell tennie az adott attribútum hiánya esetén. Ezt úgy jelöljük, hogy megadjuk azt az értéket, amelyet alapértelmezett értékként minden olyan elem visel majd, amelynél nem szerepel attribútumérték. Példánkban, ha a `column` elemnek nem adjuk meg az adott XML dokumentumban a `width` illetve `align` attribútum értékét, akkor ezek automatikus értéke rendre `"200"` és `"left"` lesz. Érték helyett egy speciális kulcsszó is szerepelhet, mely az attribútum alapértelmezett értékének jelölésére szolgál. Ezek a kulcsszavak az alábbiak lehetnek:

- *#REQUIRED* (kötelező): a tulajdonság megadása kötelező a dokumentumban az elem minden előfordulásánál.
- *#IMPLIED* (opcionális): az érték megadása nem kötelező, viszont ebben az esetben nincs alapértelmezett érték sem.
- *#FIXED* (rögzített): jelzi, hogy az attribútum neve után megadott érték rögzített, tehát nem változik, állandó.

3.3. Entitás deklaráció

Az XML nyelvvel foglalkozó fejezetben már megismerkedtünk az entitások (egyedek) jelentőségével, tehát itt az ideje megtudnunk, hogyan kell saját entitásokat deklarálni a DTD sémában, és hogyan használjuk fel őket XML dokumentumainkban. Előbb azonban tisztázzuk, hogy a deklaráció szempontjából milyen típusú entitásokat különböztetünk meg:

- Általános entitások
    - Belső entitás-deklarációval megadott entitások
    - Külső entitás-deklarációval megadott entitások
- Paraméter entitások

3.3.1. Általános entitások - belső és külső entitások

Entitás lehet karaktersorozat, de lehet egy szöveges állomány is. Az egyed mindkét fajtája felhasználható adott XML dokumentumon belül, a már ismertetett entitáshivatkozás által. Deklarációjuk viszont kis mértékben eltérő a DTD-n belül. A következő két deklaráció rendre a belső és külső entitásdeklaráció szintaxisát mutatja.

```
<!-- Belső entitás deklaráció -->
<!ENTITY entitás-név "entitás-érték">
<!-- Külső entitás deklaráció -->
<!ENTITY entitás-név SYSTEM "URI/URL">
```

Amikor az értelmező belső entitáshivatkozást talál, behelyettesíti erre a helyre a deklarációnál megadott értéket. Külső entitásra való hivatkozás esetén ez az érték viszont annak a fájlnak a tartalma, amelyre a rendszerazonosítóval (URI) hivatkoztunk a deklaráció során. Ez a fajta entitás tehát lehetővé teszi, hogy másik fájl tartalmára hivatkozzunk az XML dokumentumból. Az állományok tartalmazhatnak szöveges és bináris adatot is. Amennyiben szöveges adatot tartalmaz az állomány, akkor tartalma az adott XML dokumentumba illesztődik a referencia helyére, és úgy kerül feldolgozásra, mintha a hivatkozó dokumentum része lenne. A bináris adatokat viszont nem elemzi az értelmező.

3.3.2. Paraméter entitások

Ha a DTD készítése során, annak egy összetett részét több helyen is szerepeltetni kívánjuk, akkor az ismétlendő részt egy úgynevezett paraméter egyeddel (paraméter entitással) lehet helyettesíteni. Deklarációja után az egyedre a DTD-n belül bárhonnan lehet hivatkozni. A paraméter entitások a dokumentumtípus definíciók könnyebb olvashatóságát, átláthatóságát, értelmezhetőségét teszik lehetővé. A deklaráció az általános entitások deklarációjához hasonlít, azzal az eltéréssel, hogy itt az `ENTITY` kulcsszó és az entitás neve között egy százalékjelnek kell szerepelnie. A százalékjelet whitespace karaktereknek kell határolnia.

```
<!ENTITY % entitás-név "entitás-érték">
```

A paraméter entitásokra való hivatkozás is eltér az általános entitásoknál ismertetetteknél. Mint azt már említettem, első különbség az, hogy ezt a fajta entitást nem az XML dokumentumainkban használhatjuk fel, hanem a DTD sémánkon belül. E mellett pedig, a `&entitas-nev;` hivatkozási forma a `%entitas-nev;` formára módosul.

3.4. NOTATION deklarációk

Előfordulhat, hogy valamely egyed nem XML adatot tartalmaz. Ebben az esetben a feldolgozó programnak fel kell ismernie az adott objektum típusát, és hogy miként ágyazza be az adott XML dokumentumba illetve, hogy mit kezdjen vele. Ezt tudjuk megadni a `NOTATION` deklarációk segítségével.

4. AZ ANT

Szoftverfejlesztés során gyakran találkozunk olyan műveletsorokkal, melyeket részletes paraméterezés mellett számtalanszor meg kell ismételni. Mivel a fejlesztendő szoftver használatához egy felparaméterezett, bonyolult parancsot kellene futtatni, ezért láttam hasznosnak, hogy a végfelhasználó – vagyis a fejlesztő – számára ne csak a parancssorból való indítás álljon rendelkezésre, hanem lehetősége legyen a Java fejlesztők által jól ismert és kedvelt eszköz használatára is, az Antra **[10]**. Az Ant formájában a fejlesztők egy kellemesebb környezetet kapnak a program használatához, illetve a szoftver későbbi fejlesztésével járó konfigurálási lehetőségek számának növekedése ezzel a megoldással nem jelent problémát. Ezzel ellentétben egy felparaméterezett parancs futtatása a paraméter lista méretének növekedésével egyre kényelmetlenebbé és kezelhetetlenebbé válik. Ez a fejezet az Ant eszközt fogja részletesen bemutatni, illetve kitér arra, hogy miként lehet a szoftverünket az Ant által használható formára hozni.

4.1. A build fájl

Az Ant-ot egy XML alapú szintaxissal rendelkező fájlal, az úgynevezett build fájlal kell konfigurálni. Minden projekthez tartozik egy build fájl, melynek segítségével a projekttel kapcsolatos különböző tevékenységeket lehet végrehajtatni. A következőkben a build fájl szerkezeti felépítése kerül bemutatásra.

4.1.1. Project

Mint az már kiderült minden egyes build fájl egy adott projekthez tartozik, azzal áll szoros kapcsolatban így tehát nem meglepő, hogy minden build fájl gyökéreleme a `project` elem. Ez az elem a következő három attribútummal rendelkezhet:

- Az egyik az opcionális `name` attribútum, mellyel a projekt nevét adhatjuk meg.
- A másik egy szintén opcionális attribútum, a `basedir`, melynek segítségével megadhatjuk a projekt főkönyvtárát.

- Végezetül a harmadik egy kötelező attribútum, a `default`. Itt kell megadnunk annak az alapértelmezett `target` elemnek a nevét, mely abban az esetben fog végrehajtódni, amennyiben nem adjuk meg az Ant-nak a futtatás során, hogy mely `target` hajtódjon végre.

Látható, hogy egy olyan build fájl nem elégíti ki az előzőekben felvázolt igényeket, melyben egyetlen `project` elem van üres elemtartalommal, és maximum három attribútummal. Így tehát, a `project` elemnek kell, hogy legyen néhány gyermekeleme is. Ezek a következők lehetnek:

- A projektnek van legalább egy `target` eleme, mely a következőkben részletes ismertetésre kerül.
- Lehet egy `description` eleme, mellyel rövid, szöveges leírást adhatunk meg a projektről.

4.1.2. Target

Egy projecten belül több, úgynevezett `target` elem szerepelhet. Egy ilyen elem tartalmazza azokat a feladatokat, melyeket egyszerre kívánunk végrehajtatni. Minden target saját, egyedi névvel rendelkezik, mely lehetővé teszi azok egyértelmű beazonosítását. Ennek eredményeként az Ant segítségével a target-ek által meghatározott feladatcsoportok külön-külön végrehajthatóak. A tevékenységcsoportot alkotó tevékenységek egy-egy XML gyermekelemként jelennek meg a target-en belül, melyeket `task`-nak hívunk. Egy target-nak a következő öt attribútuma lehet:

- `name`, melyben a target nevét adjuk meg. Ez a kötelező attribútum a target egyedi azonosítására szolgál. Megjegyezném, hogy amennyiben a target neve kötőjellel kezdődik, nem hívható meg közvetlenül a parancssorból.
- A `depands` opcionális attribútummal a target-ek közötti függőségeket lehet megadni.
- A `description` szintén egy nem kötelező attribútum, mellyel a target feladatának rövid ismertetését adhatjuk meg.
- Az `if` opcionális attribútum segítségével a target futását feltételhez köthetjük. Itt egy property-t lehet megnevezni, melynek értékkomponenssel való rendelkezése jelzi az

Ant számára, hogy a target végrehajtható-e vagy sem. Ha az adott property definiálatlan, akkor nem hajtódik végre a target, míg ellenkező esetben igen.

- Az `unless` attribútum az if attribútum ellentettje. Így abban az esetben hajtódik végre a target, ha a megadott property nincs beállítva.

4.1.3. Property

A build fájl megírása során lehetőségünk van úgynevezett `property`-k használatára. Ezek név-érték párok. A név kis-nagybetű érzékeny. A build fájlon belül `${property-neve}` formában hivatkozhatunk egy-egy property-re. A névhez értéket vagy a build fájlon belül, a property elemmel, vagy egy külső property fájlban rendelhetünk. Az előbbi a következő módon tehető meg.

```
<property name="property-neve" value="property értéke"/>
```

A saját magunk által definiált property-ken kívül az Ant egyrészt lehetőséget biztosít az összes rendszer tulajdonság property-ként való lekérdezéséhez. Ilyen például az `os.name` vagy a `user.home`. Ezek a rendszer tulajdonságok a Java `System.getProperties()` metódusa által szolgáltatott értékeknek felelnek meg. Másrészt létezik még néhány beépített property is:

- A `basedir` a project elemnél beállított basedir attribútum értéke alapján a projekt abszolút elérési útvonalát tartalmazza.
- Az `ant.file` a build fájl abszolút elérési útvonalát adja meg.
- Az Ant verziószámát az `ant.version` property tartalmazza.
- Az `ant.project.name` az aktuálisan futtatott project nevét tartalmazza, mely szintén a project elemnél lett megadva.
- Az `ant.java.version` segítségével az aktuálisan használt JVM verzió számát tudhatjuk meg.
- Az `ant.home` property az Ant könyvtár elérési útját tartalmazza.

4.1.4. Path típusú struktúrák

Lehetőség van `path`, illetve `classpath` elemek létrehozására. Abban az esetben, ha egy ilyen path típusú struktúrát több helyen is használni akarunk, akkor az `id` attribútummal egyedi nevet rendelhetünk hozzá, és így máshonnan hivatkozhatunk rá. A `path` elem `pathelement`, `dirset`, `fileset` és `filelist` gyermekelemekből épül fel.

A `pathelement` elem `location` vagy `path` attribútumot tartalmazhat. Előbbivel egy fájlnév vagy egy könyvtár neve adható meg, míg az utóbbi attribútummal több könyvár is megadható egyszerre, melyek között a pontosvessző és a kettőspont használható elhatároló karakterként.

A `dirset` könyvtárak, a `fileset` pedig fájlok csoportja. Mindkettő tartalmazhat `include` és `exclude` gyermekelemeket, ahol az előbbi a nem szűrt, az utóbbi a szűrt könyvtárakat vagy csoportokat adja meg. Ezen szűrési lehetőségek mellett lehetőség van úgynevezett szelektorok használatára, melyek különböző tulajdonságok alapján választják ki a könyvtárakat és fájlokat.

A `filelist` abban különbözik a `fileset`-től, hogy a `fileset` olyan fájlokat ad vissza, amik szerepelnek a fájlrendszerben, míg a `filelist` olyanokat is, melyeknek szerepelniük kellene, de mégsem szerepelnek.

4.1.5. Taszk

A `task` elem, a target gyermekeleme. Egy target általában több taszkot tartalmaz. Ezek az egyes elemi tevékenységeket írják le az Ant számára, vagyis meghatározzák, hogy az egyes tevékenységcsoportok mely tevékenységekből, feladatokból, vagyis taszkokból épülnek fel. Attól függően, hogy melyik taszkot használjuk, más és más attribútumokkal tudjuk felparaméterezni őket, illetve más-más gyermekelemeket tartalmazhatnak.

Számtalan előre elkészített, beépített taszk létezik, ilyen például a `copy`, `mkdir`, `java`, `javac`, `jar`. Ezek általában elegendőek, de előfordulhat olyan speciális eset, amikor nem elégítik ki az igényeinket. Ebben az esetben lehetőségünk van új taszk létrehozására. Ezeket a

taszkokat bármelyik, Java nyelvben jártas programozó könnyedén kifejlesztheti, ugyanis mind az általunk készített, mind pedig az előregyártott taszkok mögött Java osztályok állnak, így biztosítva a hordozhatóságot.

4.2. Saját taszk készítése

Felmerülhet az igény, hogy saját taszkot készítsünk, melyet a későbbiek során bármikor fel tudunk használni bármely konfigurációs fájlban. Amennyiben ismerjük a Java nyelvet, abban az esetben erre is megvan a lehetőségünk. A kész taszk teljesen hasonló lesz az Ant által kínált beépített taszkokhoz. Ez is egy XML elemként fog megjelenni a build fájlon belül, ez is tartalmazhat beágyazott elemeket, illetve ugyanúgy módosíthatjuk funkcionalitását bizonyos attribútumok segítségével, mint a beépített taszkoknak. A taszkok készítésének menetét az Apache Ant Manual részletesen ismerteti **[11]**.

4.2.1. Az osztály létrehozása

Egy olyan Java osztály elkészítése a feladatunk, mely az `org.apache.tools.ant.Task` osztályt, vagy ennek valamely leszármazottját terjeszti ki. Kötelezően tartalmaznia kell egy `public void execute()`, argumentumok nélküli metódust, mely magát a taszkot valósítja meg, vagyis ez a metódus fog végrehajtódni a taszk használata során. Hiba esetén pedig a `BuildException` kivételt váltja ki (dobja).

4.2.2. A setter metódusok

A taszkhoz megadható attribútumok mindegyikéhez el kell készíteni egy-egy setter metódust. Ezeknek a metódusoknak `public` elérési szintűnek és `void` visszatérési típusúnak kell lenniük. A metódus nevének a `set` szóval kell kezdődnie, melyet az attribútum neve követ. Aktuális paraméterként az egyes setter metódusok az argumentum aktuális értékét kapják meg. Például a `public void setFile(String)` metóduson keresztül a `file` nevű attribútum aktuális értékét érhetjük el, mégpedig a metódus első paraméterén keresztül. Ezen metódusok első, és egyetlen paraméterének típusa a következő típusok egyike lehet **[11]**:

- `boolean`, ebben az esetben logikai igaz értéket vesz fel a paraméter, ha a build fájlban megadott érték `true`, `yes`, vagy `on`. Egyéb esetben hamis értéket fog kapni a paraméter.
- `char` vagy `java.lang.Character` esetén a metódus a build fájlban megadott érték első karakterét fogja megkapni.
- Más primitív típus (`int`, `short`, stb). Ekkor az Ant az adott típusra konvertálja a build fájlban megadott értéket.
- `java.io.File` esetén az Ant megvizsgálja, hogy a megadott érték egy abszolút elérési útvonal-e. Ha nem, akkor a projekt basedir paraméteréhez viszonyított relatív útvonalként fogja értelmezni azt.
- `org.apache.tools.ant.types.Path`, a megadott értéket az Ant a kettőspontok és pontosvesszők mentén tokenizálja és az egyes részeket az előző pontban meghatározottak szerint értelmezi.
- `java.lang.Class`, az Ant a build fájlban megadott értéket Java osztálynévként értelmezi, és betölti a megadott nevű osztályt.
- Bármely más típus, melynek van olyan konstruktora mely paraméterül `String` típusú értéket vár. Az Ant ezt a konstruktort fogja használni, hogy egy új példányt hozzon létre, a konstruktornak átadva a build fájlban megadott aktuális értéket.
- Az `org.apache.tools.ant.types.EnumeratedAttribute` leszármazottja. Az Ant ezen osztály `setValue` metódusát fogja meghívni.
- Java 5 enumeration.

4.2.3. A create metódusok

Amennyiben a készítendő taszk nem csak attribútumokat, hanem beágyazott elemeket is fog tartalmazni, abban az esetben minden elemhez szükség van egy-egy create metódus megírására. A metódus nevének `create` szóval kell kezdődnie, majd ezt magának a beágyazott elemnek a neve kell, hogy kövesse. A metódusnak üres formális paraméter listával kell rendelkeznie. Visszatérési típusa egy `Object` osztályt kiterjesztő osztály, mely a beágyazott elemet reprezentálja, és szintén mi készítjük el.

A beágyazott elemet reprezentáló osztály hasonló a taszkot megvalósító osztályhoz. Ez esetben is a setter metódusokon keresztül érhetjük el a beágyazott elem attribútumainak aktuális értékét, illetve további beágyazott elemek esetén itt is implementálnunk kell egy-egy create metódust. Természetesen ebben az esetben az `execute` metódus nem része az osztálynak.

4.2.4. Az elkészült taszk felhasználása

Az elkészített Ant taszk fordítás után, használható bármelyik build fájlban. Ehhez nem kell mást tenni, mint definiálni kell a build fájlon belül az új taszkot. Ezt a következőképpen kell megtennünk.

```
<taskdef name="tasknev"
         classname="a.task.osztalya.minositett.NevMegadassal"/>
```

A `taskdef` elem használata után a definiált taszkra a `tasknev` néven fogunk tudni hivatkozni. A `classname` paraméternél kell megadnunk, hogy mely osztályt kívánjuk hivatkozni a megadott néven. Ez az általunk elkészített taszk osztály, mely jelen esetben az `a.task.osztalya.minositett.NevMegadassal` osztály.

5. FREEMARKER

A kifejlesztendő szoftver XML dokumentumokban megadott értékekből állít össze Java forráskódokat. Az előállított forráskódok nagy része megegyezik, sok esetben az ezek közötti eltérés minimális. Kis túlzással mondhatjuk, hogy a feladat nem más, mint egy statikus szövegbe (template, sablon) – amely maga a forráskód állandó, nem változó része – dinamikus tartalom beillesztése – ami pedig az XML dokumentumban megadott értékek összessége (adatmodell).

Az egyik megoldás erre a problémára az lehetne, hogy írjuk meg magunk azt az eszközt (későbbiekben sablonmotor), mely elvégzi ezt a feladatot. Ezen próbálkozások viszont általában átláthatatlan kódot, speciális feladatot megoldó sablonmotort, és az egyre több igény kielégítésére egy saját sablon nyelv kialakítását eredményezik. **[12]**

Eredményesebb tehát, ha egy már elkészített sablonmotor használatához folyamodunk, melynek alkalmazásunkba való beépítésével igazán jó eredményt érhetünk el. Szerencsére mára szinte bármelyik jelentősebb programozási nyelvet használva, számos lehetőség közül választhatunk. A főbb sablonmotorokat az 1. számú összehasonlító táblázat tartalmazza.

Sablonmotor	Nyelv	Licenc	Platform	Változók	Függvények	Include	Elágazások	Ciklusok	Kiértékelés	Értékadás	Kivételkezelés
StringTemplate	Java (native), Python, C#	open-source	Platformfüggetlen	I	N	I	I	I	N	N	N
ASP.net (Microsoft)	C#, VB.net	Proprietary	MS Windows	I	I	I	I	I	I	I	I
ASP.net (Mono)	C#	GNU LGPL	Platformfüggetlen	I	I	I	I	I	I	I	I
Apache Velocity	Java	Apache License	Platformfüggetlen	I	I	I	I	I	I	I	I
Beilpuz	PHP 5	GNU LGPL	Platformfüggetlen	I	I	I	I	I	N	I	I
CheetahTemplate	Python	MIT License	Platformfüggetlen	I	I	I	I	I	I	I	I
Chip Template Engine	PHP, Perl	open-source	Platformfüggetlen	I	I	I	I	I	N	I	N
CTPP	C, C++, Perl, PHP, Python	BSD-like	Platformfüggetlen	I	I	I	I	I	N	N	I
Dylan Server Pages	Dylan language	ismeretlen	ismeretlen	I	I	I	I	N	N	N	N
eRuby	Ruby	open-source	Platformfüggetlen	I	I	I	I	I	I	I	I
Evoque Templating	Python	AFL v3.0	Platformfüggetlen	I	I	I	I	I	I	N	I
FreeMarker	Java	BSD-like	Platformfüggetlen	I	I	I	I	I	I	I	N
Genshi	Python	BSD-like	Platformfüggetlen	I	I	I	I	I	I	I	I
Haml	Ruby, PHP (WIP)	MIT License	Platformfüggetlen	I	I	I	I	I	I	I	I
JSP Weaver	Java	Proprietary	Platformfüggetlen	I	I	I	I	I	I	I	I
Jasper framework	Perl, PHP, C#, Java	open-source	Platformfüggetlen	I	N	N	N	N	N	N	N
JavaServer Pages	Java	Proprietary	Platformfüggetlen	I	I	I	I	I	I	I	I
Jinja	Python	BSD-like	Platformfüggetlen	I	I	I	I	I	I	I	N
Kid	Python	open-source	Platformfüggetlen	I	I	I	I	I	I	I	I
Open Power Template	PHP 5	GNU LGPL	Platformfüggetlen	I	I	I	I	I	I	N	N
Outline	PHP 5	open-source	Platformfüggetlen	I	I	I	I	I	I	I	I
PHAML	PHP	MIT License	Platformfüggetlen	I	I	I	I	I	I	I	I
Smarty	PHP	GNU LGPL	Platformfüggetlen	I	I	I	I	I	I	I	I
TinyButStrong	PHP	open-source	Platformfüggetlen	I	N	N	N	N	N	N	N
Vemplator	PHP	MIT	Platformfüggetlen	I	I	I	I	I	I	N	N
VlibTemplate	PHP	open-source	Platformfüggetlen	I	I	I	I	I	N	N	N
WebMacro	Java	open-source	Platformfüggetlen	I	I	I	I	I	I	I	I

1. táblázat. Sablonmotorok

Programunk megírása során a táblázatban is szereplő nyílt forrású sablonmotort, a FreeMarker-t alkalmaztuk **[13]**. Ez egy csomagban, osztálygyűjteményben kínálja fel a Java programozók számára a szolgáltatásait. Mint azt a következőkben látni fogjuk, használata roppant egyszerű, és csupán néhány Java utasítás elegendő használatához.

5.1. Model View Controller szemlélet

Mielőtt elkezdenénk a FreeMarker részletes elemzését, egy kis kitérőt kell tennünk a sablonmotorok világának hátteréül szolgáló MVC (Model-View-Controller) tervezési mintára. Az MVC egy 1970-es években született, főleg objektumorientált tervezési minta, mely egyike a legjobban kipróbált és használt tervezési mintáknak. Az MVC a felhasználói felülettel rendelkező rendszerek fejlesztése során alkalmazandó, és lényege hogy három fő részre osztjuk a rendszert. Ezek a részek a Modell (model), nézet (view) és vezérlő (controller).

Model. A modell részben valósítjuk meg az üzleti logikát. Feladata definiálni a program által használt adatstruktúrákat, illetve azokat a szabályokat, amely alapján hozzáférünk ezekhez, illetve módosíthatjuk őket. Az adatokhoz csak a modell objektumain keresztül lehet hozzáférni.

View. Feladata a modell tartalmának bemutatása, a program kimenetének generálása. Könnyen előfordulhat, hogy rendszerünket többféle interfészen keresztül (böngésző, mobil telefon, webszolgáltatás, stb.) használják, így többféle kimenetre (HTML, WML, XML) lehet szükség. Itt jelentkezik az MVC használatának előnye. Ha egy új interfész típust kell támogatnunk, akkor csak egy új nézetet kell készítenünk, ami a már meglévő modellt használja. Ehhez a részhez tartozik minden egyéb osztály is, ami a megjelenítéssel kapcsolatos, például egy tömb alapján táblázatot generáló osztály. Általában a nézetek tartalmazzák a felhasználói interakciók lehetőségét megvalósító eszközöket, például link, gomb, beviteli mező.

Controller. Ez az alkalmazás lelke. Ez köti össze a modellt és a nézeteket. A felhasználó által eszközölt akciót (form elküldése, kattintás egy linkre, stb.) lefordítja egy a modell által végrehajtandó akcióra. Ezután szintén a controller dolga, hogy az akció lefutásának eredményétől függően megjelenítse a szükséges view-t.

Mint azt láthatjuk mind a FreeMarker, mind pedig a sablonokkal való dolgozás is az MVC alapjaira épül. A sablonmotorok használatával teljesen különválasztható a nézet és a modell. Például egy webes alkalmazás elkészítése során a grafikus dolgozhat a sablon

fájlokon, vagyis a megjelenésre szolgáló részén a rendszernek a nélkül, hogy szüksége legyen a programozó jelenlétére, vagy ő maga módosítaná a rendszermodellt. A programozó pedig az üzleti logikát valósítja meg, vagyis a modellt. Meghatározza mely adatok jelenjenek meg, milyen módon kerüljenek ki a rendszer mögött lévő adatbázisból és a módosuló, illetve új adatok hogyan kerüljenek vissza az adatbázisba.

5.2. A sablon fájlok

A FreeMarker két különálló forrásból építkezik. Ahhoz, hogy a végeredmény megszülessen, szükség van egy sablonra. A sablon egy olyan speciális szöveges fájl, melyben a statikus részek között néhány, a FreeMarker-nek szóló utasítás található. Ez teszi dinamikussá a folyamatot. A sablonban használható utasítások, illetve elkészítésének ismertetése nem tartozik ezen dolgozat tematikájához. A fejlesztőtársam feladata volt egy sablon létrehozása, mely kielégíti a fejlesztendő szoftverrel szemben támasztott követelményeket.

5.3. Az adatmodell

A másik építőelem, amely a FreeMarker-nek szükséges, az úgynevezett adatmodell. Az adatmodell tartalmazza az összes olyan értéket, mely a sablonon belül hivatkozható és behelyettesíthető. Az adatmodell egy hierarchikus adatszerkezet, egy fa, melyben az egyes elemek Java objektumok. Szemléletesen ábrázolva a következőképpen nézhet ki egy adatmodell.

```
(root)
  |
  +- user = "Big Joe"
  |
  +- latestProduct
      |
      +- url = "products/greenmouse.html"
      |
      +- name = "green mouse"
```

5.3.1. Adatmodell felépítése általános módon

Az adatmodellt többféleképpen fel lehet építeni, amit aztán a FreeMarker-nek egy paraméter formájában továbbíthatunk. Egy ilyen adatmodell felépítésének egyik módját az alábbi példa szemlélteti.

```
// Egy HashMap példány létrehozása
Map root = new HashMap();
// "user" nevű elem elhelyezése melynek értéke "Big Joe"
root.put("user", "Big Joe");
// Egy újabb HashMap példány létrehozása
Map latest = new HashMap();
// Az elem elhelyezése latestProduct néven
root.put("latestProduct", latest);
// "url" és "name" nevű elemek elhelyezése a latestProduct gyermekeiként
latest.put("url", "products/greenmouse.html");
latest.put("name", "green mouse");
```

5.3.2. Adatmodell felépítése XML-ből a FreeMarker használatával

Felmerülhet az igény arra – ahogyan az esetünkben is történt –, hogy az adatmodell felépítéséhez szükséges elemek egy adott XML fájlból származzanak. Mint azt láttuk az adatmodell – akárcsak az XML – hierarchikus szerkezetű, így semmilyen elvi akadálya nincs annak, hogy ezt az igényt kielégítsük. Sőt mi több, a FreeMarker beépített eszközöket biztosít ahhoz, hogy a modellünkben lévő adatok XML-ből való kiolvasása minél egyszerűbb legyen. A következő kódrészlet egy XML dokumentumból való adatmodell felépítését mutatja be.

```
Map root = new HashMap();
root.put("doc", freemarker.ext.dom.NodeModel.parse(new File(
        "the/path/of/the.xml")));
```

A példában a `parse(File)` metódus egy `freemarker.ext.dom.NodeModel` típusú objektum referenciával tér vissza, mely a megadott XML fájl teljes tartalmát reprezentáló DOM modell gyökér elemét reprezentálja. Ezen keresztül bármely más, a hierarchiában lévő elem elérhető. Így ezt az elemet kell elhelyeznünk a `root` nevű `HashMap` típusú változóban, mely a sablonban használandó adatmodellt fogja szolgáltatni a FreeMarker számára.

Ezzel a megoldással nem kell az XML egyes elemeit kiolvasnunk, majd az azoknak megfelelő `HashMap` típusú objektumokat létrehozni, vagyis nem kell nekünk bajlódni a FreeMarker-nek szánt adatmodell felépítésével. Nem is beszélve arról, hogy így az adatok dinamikusak, mivel a külső XML dokumentum tartalmát bármikor megváltoztathatjuk.

5.3.3. Adatmodell felépítése XML-ből a DOM API használatával

Az előző esetben egy speciális, a FreeMarker API-ban található `NodeModel` típusú objektumot helyeztünk el a `root` nevű `HashMap` típusú változóban, mely a későbbiekben az adatmodellt szolgáltatja. Ezt a típust természetesen a FreeMarker hibátlanul tudja kezelni, mint egy adatmodell elemet. A sablonon belül megadott utasítások segítségével egyszerűen érhetjük el az egyes `NodeModel` típusú objektumokon keresztül az általuk reprezentált XML elemeket, illetve azok attribútumait és gyermekelemeit.

Lehetőségünk van az előzőekben ismertetett DOM API használatára, illetve az elkészített DOM fa FreeMarker-rel történő feldolgozására is. Tehát a DOM API által megvalósított XML feldolgozó segítségével felépített DOM modell, és annak `org.w3c.dom.Node` típusú elemei szintén értelmezhetőek a FreeMarker számára.

A fejlesztendő projekt esetében ez utóbbi módszert választottam az XML dokumentumok feldolgozására és értelmezésére, mivel a FreeMarker által kínált lehetőség nem elégítette ki teljes mértékben a meghatározott követelményeket és a kitűzött célokat. E mellett szükség volt az adatmodellt szolgáló XML dokumentumokon kívül egyéb dokumentumok feldolgozására is.

5.4. A FreeMarker használata

Az adatmodell felépítése és a sablon elkészítése után következik a FreeMarker alkalmazása. Ez egy Java alapú sablonmotor, felhasználása ezen a programozási nyelven keresztül valósul meg. Alkalmazásával néhány Java utasítás elegendő ahhoz, hogy a sablonból és az adatmodellből előálljon a kívánt végeredmény.

5.4.1. Configuration példányosítása

Első lépésként egy `freemarker.template.Configuration` példány létrehozása szükséges ahhoz, hogy a FreeMarker-t működésre bírjuk. Ezen objektum segítségével történik meg a FreeMarker konfigurálása, valamint ez a felelős a sablon fájlok betöltéséért, elő-feldolgozásáért és gyorsító-tárazásáért.

```
Configuration cfg = new Configuration();                    // (1)
cfg.setDirectoryForTemplateLoading(
      new File("/template/fájlok/elérési/útvonala"));       // (2)
cfg.setObjectWrapper(new DefaultObjectWrapper());           // (3)
```

A fenti kódrészlet hatására, elsőnek (1) létrejön egy új `cfg` nevű objektum, mely `freemarker.template.Configuration` típusú. Majd (2) megadjuk, hogy a FreeMarker honnan töltse be a kívánt sablon fájlokat. Végezetül (3) beállítjuk az alapértelmezett adatmodell feldolgozót, melynek hatására az adatmodellt felépítő objektumok - `String`, `HashMap`, `Number`, `Boolean` vagy éppen `org.w3c.dom.Node` példányok - a FreeMarker számára feldolgozható és értelmezhető típusú, az eredetiekkel egyenértékű új objektumokkal helyettesítődnek.

Természetesen lehetőség van arra is, hogy a FreeMarker több helyről töltse be a sablonokat. Megadhatunk több könyvtárat a fájlrendszerben, vagy éppen egy osztályt, melynek `getResource()` metódusa lesz felhasználva a sablon betöltésére, vagy akár webes alkalmazás könyvtárából is betölthetjük a kívánt sablon fájlokat. Mindezeket a lehetőségeket korlátlan számban használhatjuk fel, és tetszőlegesen kombinálhatjuk kódunkban, így elégítve ki saját igényeinket.

5.4.2. Az adatmodell létrehozása

Második lépés az adatmodellünk felépítése. Az előzőekben ismertetett módszerek valamelyikével hozzuk létre a kívánt adatmodellünket. Ennek eredményeképpen előáll egy `HashMap` példány, melyet a későbbiek során a FreeMarker a sablonokban megadott dinamikus részek kitöltésére fog használni.

5.4.3. Sablon fájlok betöltése

Szükségünk van még a sablonok betöltésére, megnyitására. Egy-egy sablont a `freemarker.template.Template` példányok reprezentálják, melyeket az előzőleg létrehozott és megfelelően beállított `Configuration` példányon keresztül tudjuk létrehozni.

```
Template temp = cfg.getTemplate("test.ftl");
```

Az utasítás hatására a `cfg` névvel hivatkozott `Configuration` típusú objektum segítségével létrehozzuk a `test.ftl` nevű sablon fájlt reprezentáló objektumot. A fájlnak az előzőleg beállított helyek közül valahol szerepelnie kell.

5.4.4. A végeredmény előállítása

Végezetül a FreeMarker használatával össze kell fésülnünk az elkészült adatmodellünket és a sablon fájlunkat. A sablonban szereplő speciális utasításokat értelmezve, a FreeMarker az adatmodellt felhasználva előállítja a kívánt kimenetet.

```
Writer out = new OutputStreamWriter(System.out);
temp.process(root, out);
out.flush();
```

A fenti utasítások eredményeképpen kiíródik a sablon és az adatmodell alapján elkészült végeredmény. Természetesen az eredmény kerülhet egy tetszőleges fájlba, vagy akár egy adatfolyamba, melyet az igényeinknek megfelelően további feldolgozás alá vethetünk.

6. A HIBERGUI FEJLESZTÉSE

A dolgozat témáját egy olyan szoftver fejlesztésén keresztül kívánom bemutatni, melynek fő feladata, hogy XML dokumentumokban definiált paraméterek segítségével Java forráskódot generáljon. A fejlesztés kezdetén már adott volt egy olyan Java osztálygyűjtemény, mely egyedi készítésű, JFace és SWT elemeket kiterjesztő, speciális célú komponenseket tartalmaz. Ezeknek a komponenseknek a felhasználásával olyan grafikus felület készíthető el, mellyel a Hibernate eszközt használó adatbázis alapú alkalmazásaink irányítása, vezérlése történhet meg. Az így létrejött grafikus felületet megvalósító osztályok csak kis mértékben térnek el, szerkezetileg majdnem azonos felépítést mutatnak. Ez adta az ötletet, hogy a komponensgyűjteményt bővítsük ki egy olyan automatizmust nyújtó résszel, mely XML dokumentumokban megadott információk alapján képes létrehozni a grafikus felületet megvalósító Java forráskódokat. Az így létrejövő szoftver – mely tartalmazza a speciális komponenseket és az említett új részt – a HiberGUI fantázianevet kapta.

A Standard Widget Toolkit (SWT) a Java programozási nyelv ablakkezelésre és grafikus felhasználói felületek létrehozására szolgáló komponensgyűjteménye **[14]**. Ellentétben az AWT-vel – mely szintén egy Java API a grafikus felhasználói felületek létrehozására – az SWT nem része a szabvány Java API-nak. Az SWT egy alternatíva az AWT helyett, melyet az IBM fejlesztett ki, és jelenleg az Eclipse Foundation gondozása alatt van. Nagy előnye az SWT-nek, hogy natív kéréseken keresztül, közvetlenül az ablakkezelő rendszerrel kommunikál, így jóval gyorsabb más Java komponenskészleteknél – például az AWT-nél.

A JFace egy Java segédosztály gyűjtemény, amely a GUI programozását hivatott megkönnyíteni **[14]**. Az SWT felett helyezkedik el, arra épül, de nem rejti el azt teljes egészében. Alkalmazásával újabb, magasabb szintű komponenseket kapunk kézhez, de mellettük az SWT komponensek is felhasználhatóak a grafikus felhasználói felület programozása és elkészítése során.

A Hibernate egy olyan réteget biztosít az adatbázis és az alkalmazás között, mely segítségével lehetőség van a relációs adatbázis és az abban lévő adatok objektumorientált módon való kezelésére. Az esetek többségében mondhatjuk, hogy alkalmazásával az adatbázisunk tábláinak egyes rekordjait kezelhetjük egy-egy objektumként.

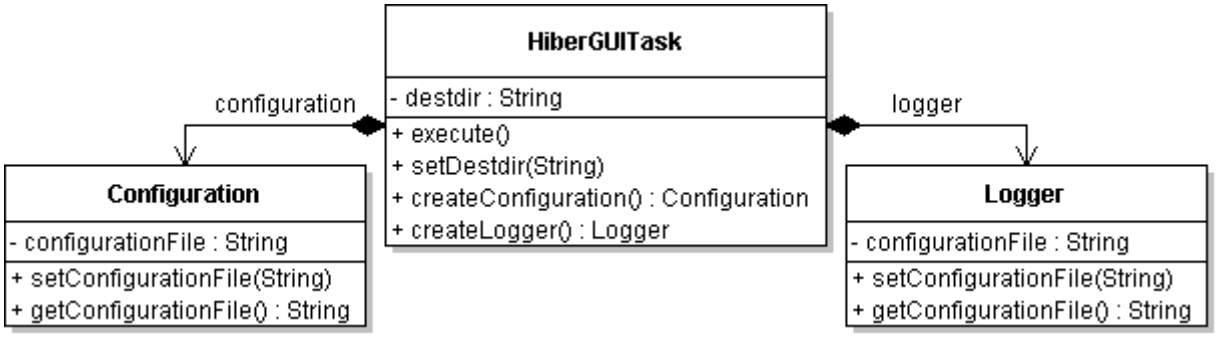
6.1. Az Ant és a HiberGUI

A HiberGUI fejlesztése során az egyik cél az volt, hogy alkalmazása lehetséges legyen a Java fejlesztők által előszeretettel használt Ant eszközzel együtt, így könnyítve meg a fejlesztések menetét. Ahhoz, hogy ez megvalósuljon, szükség volt egy speciális Ant taszk elkészítésére.

A taszkkal szemben támasztott követelmények a következőképpen alakultak. A taszknak lehetőséget kell biztosítania a HiberGUI konfigurációs XML fájl elérési útjának megadására, melyből a generáláshoz szükséges információkat fogjuk kinyerni. A későbbi bővíthetőség miatt, a konfigurációs állomány elérési útját egy gyermekelem attribútumaként lehessen megadni. Szintén egy gyermekelem attribútumaként lehessen megadni a HiberGUI által használt naplózó egység konfigurálására szolgáló külső szöveges fájlt is. Legyen lehetőség e mellett egy célkönyvtár megadására, melybe a generált forrásfájlok fognak kerülni a HiberGUI használata során. Ezt az elérési utat a taszk egyik attribútumaként kelljen megadni. Mindezek után a taszk felhasználása a következőképpen kell, hogy alakuljon.

```
<hiberguitool destdir="${celkonyvtar}">
      <configuration configurationfile="${hibergui-konfiguracios-fajl}"/>
      <logger configurationfile="${logger-konfiguracios-fajl}"/>
</hiberguitool>
```

A fentiek értelmében a taszknak tehát tartalmaznia kell egy setter metódust a `destdir` paraméter számára, egy-egy create metódust a `configuration` és a `logger` gyermekelem létrehozásához, illetve ezt a két elemet reprezentáló egy-egy belső (vagy akár külső) osztályt. E mellett természetesen implementálni kell még az `execute()` metódust, mely magát a taszk feladatát fogja megvalósítani. Az osztálydiagram az 1. ábrán látható módon alakul.



1. ábra. Az Ant taszkot megvalósító osztályok UML diagramja

Az elkészült osztályok beállító és lekérdező metódusai (getter, setter, create metódusok) egyszerűségük miatt különösebb részletezésre nem szorulnak. A taszk feladatát megvalósító `execute()` metódus az, mely további magyarázatot igényel.

```
if (logger != null && logger.getConfigurationFile() != null) {
      System.setProperty("java.util.logging.config.file",
                         logger.getConfigurationFile());
      try {
            LogManager.getLogManager().readConfiguration();
            Generator.getLogger().config("Logger configurations readed" +
" successfully from file \"" + logger.getConfigurationFile() + "\"");
      } catch (IOException e1) {
            Generator.getLogger().severe("Exception while reading logger" +
" configurations from file \"" + logger.getConfigurationFile() + "\"");
            e1.printStackTrace();
            System.exit(2);
      }
}
Generator.getLogger().config("HiberGUI started by ant task.");
if (configuration == null || configuration.getConfigurationFile() ==null) {
      Generator.getLogger().severe("Unspecified configuration XML. Set" +
" it as a nested element of this task.");
      System.exit(1);
}
try {
      if (destdir != null) {
            new Generator(configuration.getConfigurationFile(), destdir);
      } else {
            new Generator(configuration.getConfigurationFile());
      }
} catch (SAXException e) {
      ...
```

A taszk első feladata a naplózó rendszer konfigurálása, a taszk `logger` nevű gyermekelemének az attribútumaként megadott fájl alapján. Ez után, a kód további részében a HiberGUI naplózó egységét, vagyis a `Generator` osztályon keresztül elérhető `java.util.logging.Logger` típusú statikus példányt használjuk bizonyos fontosabb események rögzítésére, naplózására. A `config(String)` metódussal például rögzítjük a taszk elindításának pillanatát. Ezek után ellenőrizzük, hogy meg van-e adva a `configuration` elem és annak `configurationfile` attribútuma. Amennyiben igen, akkor elindítjuk a generálást a `Generator` osztály példányosítása által, ellenkező esetben pedig naplózzuk a hibát és megszakítjuk a program további működését.

Az elkészített Ant taszk fordítás után használható bármelyik build fájlon belül. Ehhez nem kell mást tenni, mint definiálni kell a build fájlon belül az új taszkot. Ezt a következőképpen kell megtennünk.

```
<taskdef name="hiberguitool"
         classname="hibergui.generator.tool.ant.HiberGUITask"
         classpathref="project.classpath.hibergui"/>
```

A `taskdef` elem után, a definiált taszkra a `hiberguitool` néven fogunk tudni hivatkozni. A `classname` paraméternél adjuk meg, hogy mely taszk osztályra kívánunk hivatkozni a megadott néven. Ez jelenleg a `hibergui.generator.tool.ant.HiberGUITask` osztály. A `classpathref` attribútum segítségével a `classpath` környezeti változót állítjuk be egy path típusú struktúrára, a Java futtató környezet (JVM) számára, mely többek között magát a taszkot fogja futtatni. A struktúra a következőképpen néz ki.

```
<path id="project.classpath.hibergui">
      <fileset dir="../HiberGUI/">
            <include name="dist/hibergui.jar"/>
      </fileset>
      <fileset dir="lib">
            <include name="*.jar"/>
            <include name="schema/*.jar"/>
      </fileset>
</path>
```

Végezetül lássuk a teljes target-et, mely a HiberGUI általi generálást fogja elindítani. A target-en belül megadott tevékenységcsoport végrehajtását az `ant gui` paranccsal tudjuk elindítani.

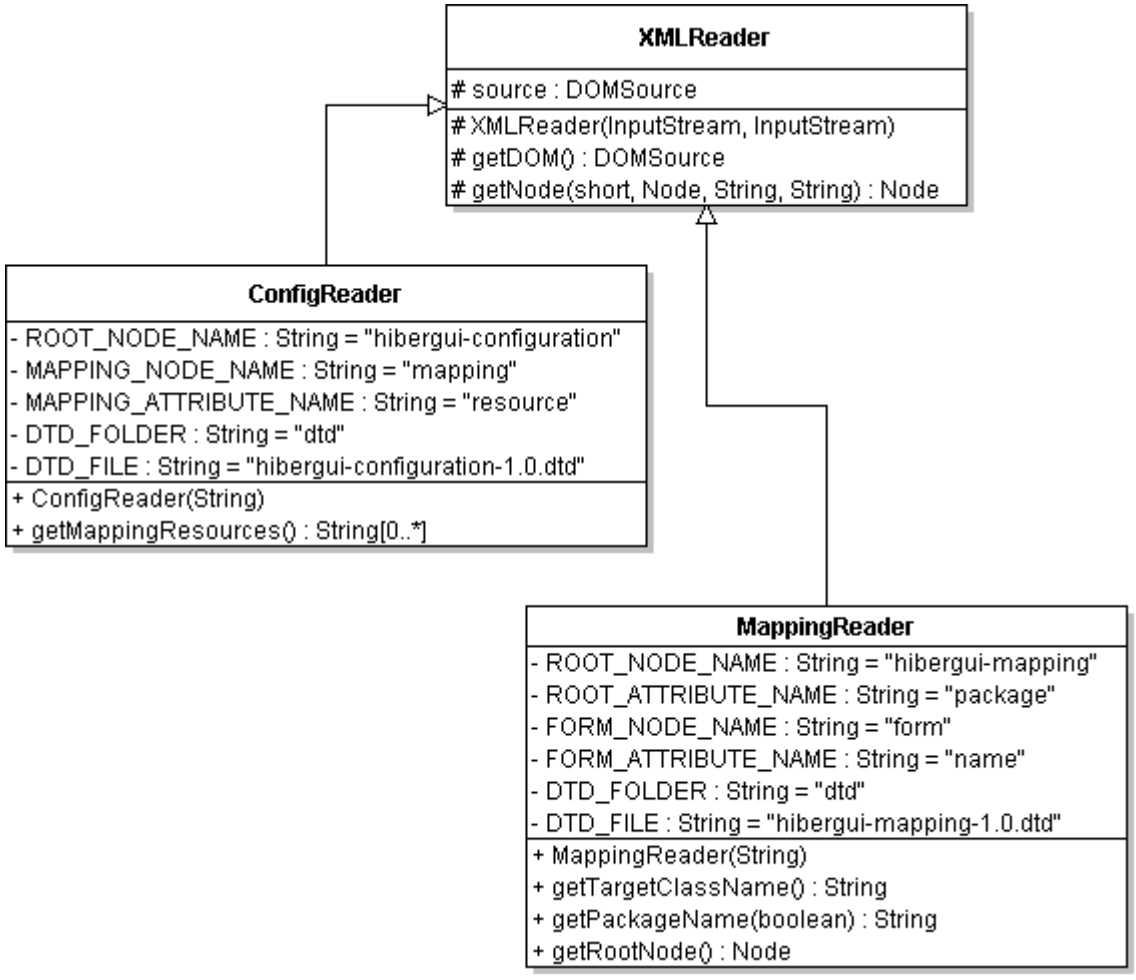
```
<target name="gui" description="Generates GUI files.">
      <taskdef name="hiberguitool"
               classname="hibergui.generator.tool.ant.HiberGUITask"
               classpathref="project.classpath.hibergui"/>
      <hiberguitool destdir="${build.src}">
            <configuration configurationfile="${hibergui.cfg}"/>
            <logger configurationfile="${hibergui.logfile}"/>
      </hiberguitool>
</target>
```

6.2. XML dokumentumok és azok feldolgozása

A HiberGUI a működése során kétféle XML dokumentummal kerül kapcsolatba. Egyik a konfigurációs XML, melyben az esetleges beállítások, illetve további XML állományok elérési útja szerepel. Ezek a további állományok alkotják a második csoportot, melyek a FreeMarker számára szolgáltatják magát az adatmodellt. Általuk írhatják le a HiberGUI-t alkalmazó fejlesztők az egyes generálandó Java forráskódok szerkezetét, illetve változó részeit. Egy-egy ilyen dokumentum alapján épül fel egy-egy Java forráskód.

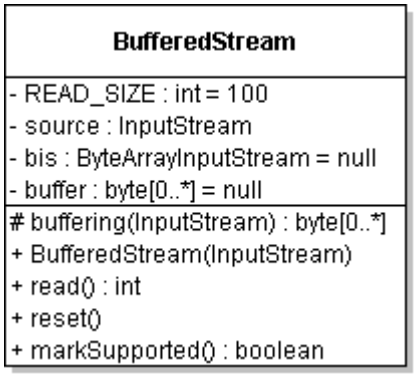
Már a tervezés során biztosra vehető volt, hogy a FreeMarker által kínált XML feldolgozási módszer nem lesz elegendő. Egyrészt szükség van a HiberGUI-nak megadott konfigurációs XML feldolgozására és értelmezésére is. Másrészt az előzőekben ismertetett technika, miszerint a FreeMarker-nek adatmodellként egy konkrét XML állományt adunk meg, nem célravezető. Ennek oka az, hogy a megadott XML dokumentum validálása minden esetben a dokumentumhoz megadott dokumentumtípus definíció alapján történik. Esetünkben ez nem elegendő, mivel a validálás alapját egy külső fájl is szolgáltathatja, abban az esetben, ha az XML dokumentumhoz megadott DTD nem elérhető. Például az egy külső fájlban található egy távoli gépen, és a gép melyen a validálás történik, nem rendelkezik internet kapcsolattal.

Látható, hogy több típusú XML dokumentumot kell értelmeznünk, feldolgoznunk. Ezekhez egy-egy DTD elkészítése szükséges, és az egyes XML dokumentumok validálása ezek alapján kell, hogy történjék. A további fejlesztések pedig újabb és újabb dokumentumtípusokat hozhatnak magukkal. Éppen ezért egy általános XML feldolgozó megalkotása volt az első lépés, mely validálja a megadott dokumentumot a megadott DTD séma alapján, majd felépíti a dokumentumot reprezentáló DOM fát. E mellett a feldolgozónak biztosítania kell egy egyszerű lehetőséget, adott elem értékének vagy attribútumának eléréséhez. Ehhez egy olyan metódust készítettem, amelynek ha megadjuk a gyökérelemtől az adott elemhez vezető utat, elérhetjük magát az elemet vagy annak egyes attribútumait. Ezek után az általános célú XML feldolgozó osztály leszármazottait úgy készítettem el, hogy egy-egy speciális XML dokumentum kezelésére legyenek alkalmasak. A konfigurációs és az adatmodellt szolgáltató XML állományoknak megfelelő két osztály rendre a `ConfigReader` és a `MappingReader` nevet kapta. Az osztályok UML diagramját a 2. ábrán láthatjuk.



2. ábra. Az XML feldolgozó osztályok UML diagramja

Az `XMLReader` – általános XML feldolgozó osztály – konstruktora első paraméterként egy adatfolyamot kap, melyen keresztül maga az XML dokumentum érhető el. Második paraméterként a külső DTD-t tartalmazó adatfolyamot várja a konstruktor, mely akkor kerül használatra, ha az XML dokumentumhoz megadott dokumentumtípus definíció nem létezik, vagy nem érhető el. A fejlesztés során ragaszkodtam ahhoz, hogy az XML dokumentum ne fájlnévként vagy egy `java.io.File` példányként kerüljön az osztály konstruktorához, hanem adatfolyamként, mely az XML dokumentum tartalmát szolgáltatja, így a későbbi fejlesztések során egy jóval kényelmesebben használható osztályt kapunk kézhez. A probléma az volt, hogy az adatfolyamot többször is fel kell használnunk a speciális validálási igények miatt. Mivel hosszas információgyűjtés után sem találtam megfelelő megoldást arra, hogy a `javax.xml.parsers.DocumentBuilder` osztály `parse(InputStream)` metódusa a paraméterül kapott adatfolyamot a feldolgozás végén ne zárja le, illetve, hogy valamilyen egyszerű módon az adatfolyamot újratöltve ismét átadjam az említett metódusnak, ezért készítenem kellett egy saját osztályt mely megfelelően ellátja ezt a feladatot. A létrejött osztály egy adott adatfolyamot teljes egészében beolvas a központi tárba, és onnan tetszőleges alkalommal tudjuk újra beolvasni azt segítségével. A 3. ábra ennek az osztálynak az UML diagramját mutatja.

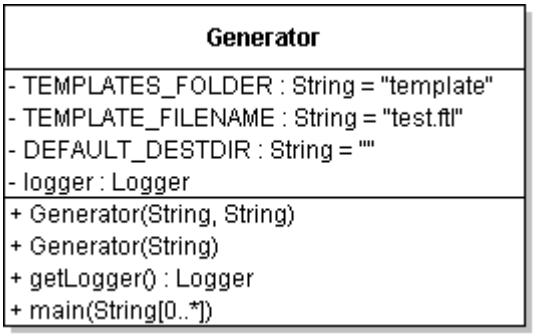


3. ábra. A BufferedStream osztály UML diagramja

A két speciális célú XML feldolgozó konstruktorként az XML dokumentum nevét kapja, és konstans értékekként tartalmazzák a validáláshoz szükséges DTD dokumentumok nevét, illetve elérési útját. E mellett olyan speciális célú metódusokat tartalmaznak, melyek az adott XML dokumentumban lévő adatok eléréséhez szükségesek. Ezeket az osztályokat és metódusaikat a következőkben ismertetésre kerülő `Generator` osztály használja fel.

6.3. A HiberGUI motorja – a generátor

Az elkészült XML feldolgozó osztályainkat valahol használnunk kell. Valahol el kell indítani magát a generálást, mely nem más, mint az egyes XML-ek beolvasása, validálása és értelmezése, majd ezt követően a FreeMarker segítségével a Java forráskód előállítása. Ezt a feladatot valósítja meg a `Generator` osztály, melynek UML osztálydiagramja a 4. ábrán látható.



4. ábra. A Generator osztály UML diagramja

Az osztály `Generator(String, String)` konstruktora tartalmazza magát a generálást megvalósító kódot. A konstruktor két paramétert kap. Az egyik a konfigurációs XML dokumentum relatív elérési útvonala, míg a másik az elkészítendő Java forráskódok célkönyvtára. A generálás első lépéseként – a `ConfigReader`-t a konfigurációs XML-re alkalmazva – megállapítjuk az adatmodelleket leíró XML dokumentumok listáját. Ez után ezeket egyenként véve a FreeMarker használatával előállítjuk az egyes Java forráskódokat a megadott célkönyvtárba.

Kiemelném még a `main` metódust, mely egy lehetséges belépési pont a programba. Az előzőekben ismertetett Ant eszköz használata mellett, lehetőségünk van tehát a program parancssorból való indítására is. A HiberGUI konfigurálása ebben az esetben paraméterek megadásával történik. Természetesen a későbbiek során, mikor az egyes beállítási lehetőségek száma jelentősen megnövekszik, ez az út kevésbé járható, és inkább az Ant taszk további bővítése a kézenfekvőbb, egyszerűbb és jobb megoldás.

7. ÖSSZEFOGLALÁS

A cél az volt, hogy egy olyan programot készítsünk, melyen keresztül bemutatható a Java-val való XML feldolgozás, és az, hogy az alkalmazás az értelmezett XML dokumentum alapján Java forráskódot állítson elő. Megállapíthatjuk, hogy a HiberGUI fantázianévre hallgató szoftver a kitűzött követelményeket teljes mértékben teljesíti. Az elkészült szoftver segítségével ugyanis XML dokumentumok által specifikált grafikus felhasználói felület állítható elő.

További szempont volt még, hogy a szoftver kényelmes és könnyen használható segédeszközt nyújtson a grafikus felhasználói felületet fejlesztő programozók számára. A közkedvelt Ant eszköz bevonásával ez mindenképpen teljesítettnek mondható. A HiberGUI-t alkalmazó fejlesztők számára egy felparaméterezhető, konfigurálható Ant taszk áll rendelkezésre. Az Ant bevonásának köszönhetően a létrejött program nem csak egy jól testreszabható segédeszköz, hanem egy bővíthető alapot nyújt minden jövőbeli fejlesztés során. Ennek és a gondos tervezésnek az eredménye, hogy a jelenlegi hiányosságokat a későbbiek során könnyen, problémamentesen és újratervezés nélkül lehet pótolni.

Egyik ilyen nagy hiányosság a viszonylag kevés grafikus komponens ismerete. Sajnos a jelenlegi fejlesztési idő nem volt elegendő ahhoz, hogy a grafikus felhasználói felület teljes mértékben testreszabható legyen az XML dokumentumok által. Jelenleg igen kevés összetevőt lehet a forráskódok alapjául szolgáló XML dokumentumokban specifikálni, leírni. Ezen hiányosságok megszüntetéséhez csupán a grafikus felhasználói felületet leíró állományok DTD sémájának, valamint a FreeMarker által használt sablon fájlnak a tüzetesebb felülvizsgálata és bővítése szükséges.

8. IRODALOMJEGYZÉK

[1] *Brett McLaughlin: Java és XML* Kossuth Kiadó ZRt., 2001.

[2] *Bíró Szabolcs: Szövegfeldolgozás XML alapokon* Budapest, Neumann Kht. (Neumann-ház), 2005.

[3] *Balázsy Attila: Bemutatkozik az XML* http://www.prog.hu/cikkek/884/Bemutatkozik+az+XML/oldal/1.html

[4] *World Wide Web Consortium: Extensible Markup Language (XML) 1.0 (Fourth Edition)*, 2006. http://www.w3.org/TR/2006/REC-xml-20060816/

[5] *Juhász István: Programozás 1* Debrecen, elektronikus közlés mobiDIÁK könyvtár, 2003. http://infotech.inf.unideb.hu/prog1/docs/programozas120040519.pdf

[6] *Nagy Gusztáv: Web programozás 0.6* Kecskemét, 2008. http://nagygusztav.hu/files/Web_programozas_jegyzet_0.6_jav.pdf

[7] *Elliotte Rusty Harold: Processing XML with Java - A Guide to SAX, DOM, JDOM, JAXP, and TrAX*, Addison-Wesley Professional, 2002.

[8] *Nagy Gusztáv: Java programozás 1.3* Kecskemét, 2006. http://nagygusztav.hu/files/Java_programozas_1.3.pdf

[9] *SAX: Quickstart* http://www.saxproject.org/quickstart.html

[10] *Viczián István: Apache Ant – Java-based build tool*, 2004. http://delfin.klte.hu/~vicziani/pdf/ant.pdf

[11] *Apache Ant 1.7.0 Manual* http://ant.apache.org/manual/index.html

[12] *Viczián István: Velocity Template Engine – Avagy miért ne írjunk saját template engine-t Java nyelven?* Budapest, 2006. http://delfin.klte.hu/~vicziani/pdf/velocity.pdf

[13] *FreeMarker Manual – For FreeMarker 2.3.12* http://freemarker.org/docs/index.html

[14] *Robert Harris: The Definitive Guide to SWT and JFace*, Apress, 2004.