Debreceni Egyetem
Informatikai Kar

Grafikus felhasználói felület generálása XML alapján

Témavezető:
Espák Miklós
egyetemi tanársegéd

Készítette:
Laczkó Sándor
PTI szakos hallgató

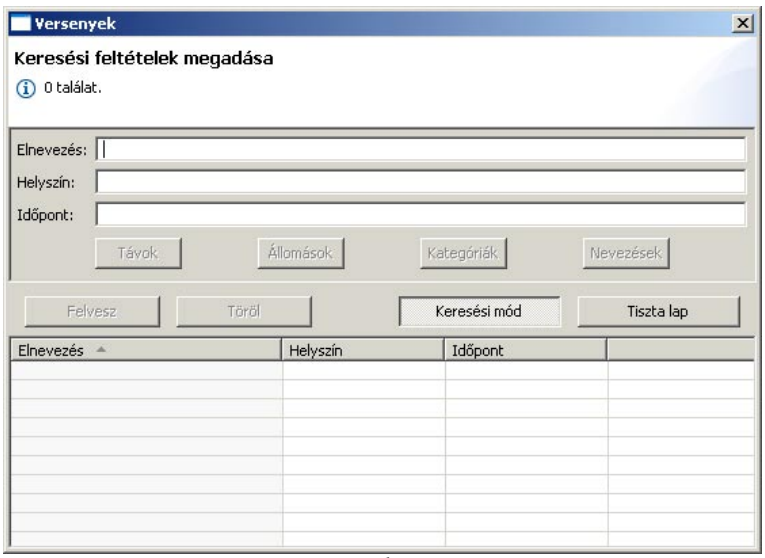
Debrecen
2008

Tartalomjegyzék

1. A felhasználói felületek

A felhasználói felület (User Interface) egy szoftverrendszer azon része, melynek segítségével a felhasználó interakciókat végezhet a programmal. Ez az interakció lehet egy adat lekérdezése, bevitele, megjelenítése stb. A felhasználói felületeket három nagy csoportba lehet osztani: parancssoros, szöveges és grafikus felhasználói felület.

A grafikus felhasználói felület (Graphical User Interface – GUI) olyan felület, amely a számítógép grafikus kártyájának grafikus üzemmódját használva jeleníti meg az elemeket. Ezeknél alapvető szerepet játszik az egér, vagy valamely más mutatóeszköz. Gyakori elemei az ablak, szövegbeviteli mező, gomb, jelölőnégyzet, ikon. A modern operációs rendszerek alapvető jellemzője, hogy grafikus felülettel rendelkeznek. Az ilyen fajta felület használata a felhasználó számára sokkal könnyebben elsajátítható szemben a parancssoros felülettel, ahol utasításokat kell kiadni. A grafikus felület célja tehát a felhasználó munkájának segítése. Ezen oknál fogva a grafikus GUI tervezésekor figyelembe kell venni, a felhasználók mentális és fizikai képességeit, hogy mennyi időt fognak a gép előtt tölteni, a megjelenítendő információk könnyen átláthatóak legyenek, könnyen megtanulható legyen, stb. A felületek ennek megfelelően ideális esetben kevés színből állnak össze, tömörek, egyértelműek.



1. ábra

Az 1. ábrán látható egy tipikus GUI elem, az ablak. Ennek a következő elemei vannak:

– Címsor: Az ablakokhoz nevek rendelhetők, és ezt a nevet találjuk itt.
– Ablakkeret: Az ablak széleit határoló keret.
– Vezérlő gomb(ok): Az ablak méretét állíthatjuk be, illetve be is zárhatjuk azt.
– Vezérlő menü: A vezérlő gombhoz hasonló funkciót ellátó menü
– Nyomógomb: Általános vezérlőelem, funkcióját a rajta elhelyezett felirat árulja el
– Szövegbeviteli mező: Szöveges adatok bevitelére alkalmas vezérlőelem.
– Címke: Valamely elemet megnevező felirat az adott elem környékén.
– Táblázat: Az információk tömör, könnyen feldolgozható formában, táblázatban is megjeleníthetőek.

1.1. A grafikus felület megvalósítása

Egy ilyen felület operációs rendszerenként eltérő kinézetű lehet, de az alapvető elemek általában mindegyiken megtalálhatóak. Ebből adódik a probléma, hogy ha egy GUI-val rendelkező alkalmazást szeretnénk fejleszteni, akkor annak az implementációja nagyban eltérhet más-más operációs rendszer esetén. Ez azt eredményezheti, hogy esetlegesen a felhasználói felületet teljesen újra kell írnunk ha a program felhasználója úgy dönt, hogy különböző platformon kívánja használni. Erre a problémára egy lehetséges megoldás a platform független programnyelv használata. Ilyen nyelv pl. a Java. A Javában megírt program a számítógép operációs rendszerére telepített virtuális számítógépen fut, így áthidalva a különböző rendszerek közötti eltéréseket. A Java nyelv alapcsomagjában kétféle megoldást is találunk a GUI megvalósításra: az AWT-t és Swinget.

Az AWT (Abstract Window Toolkit) egy hordozható programkönyvtár szoftverrendszerek GUI-jának létrehozásához. Lényeges, hogy ez egy magas szintű absztrakciós csomag, mivel elrejti a GUI megvalósításának részleteit. Az AWT a platform grafikus eszközkészletének elemeit jeleníti meg, emiatt szoros kapcsolatban áll velük, ezért komponenseit nehézsúlyú (heavyweight) komponenseknek is nevezik. Az operációs rendszeren való kommunikációt a JNI (Java Native Interface) segítségével valósítja meg Az ilyen módon megjelenített ablak kinézete jobban hasonlít az operációs rendszer által rajzolt ablakéhoz.

A Swing az AWT-vel szemben teljesen Java alapú és több komponenst kínál, mint az AWT. Hátránya viszont, hogy mivel a virtuális gépnek kell elvégezni a grafikus rajzolást, emiatt lassabb futást eredményez, mint a másik megoldás. A Swingben íródott felület kinézete eltér a platform grafikus rendszere által rajzolt felületekétől, sajátos Java kinézetet ad. Ezen kinézet testre szabható a look-and-feel architektúrának köszönhetően, így közelebb vihető a kinézete a platform grafikus rendszeréhez. A Swing segítségével készített felületeket szokás könnyűsúlyúnak (lightweight) felületnek is nevezni.

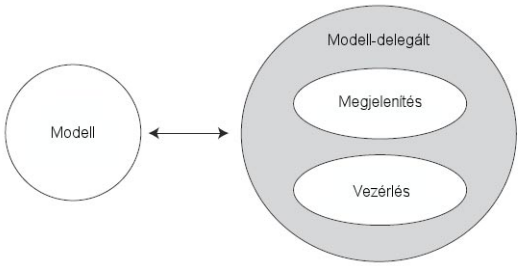
1.2. Az Eclipse SWT eszközrendszere

A fenti két lehetőség kínálkozik a Java keretrendszerében, de léteznek alternatív megoldások is. Egy ilyen alternatív megoldás az Eclipse Foundation gondozásában lévő SWT (Standard Widget Toolkit) [5] is. Az SWT egy nyílt forráskódú eszközrendszer, ami az AWT-hez hasonlóan a platform grafikus eszközrendszerének natív megjelenítési komponenseit használja, így ha egy grafikus elemet helyezünk el az alkalmazásunkban, az az adott platformnak megfelelően néz ki, attól függetlenül, hogy milyen rendszer alatt fejlesztettük, hiszen Java alkalmazásról beszélünk. Az SWT is Javában íródott, de használ egy grafikus eszközkészletet, ami viszont minden platform esetén különböző. A jelenleg támogatott operációs rendszerek a Windows, Linux, Mac OS X, Solaris, QNX, AIX és HP-UX. Az SWT megvalósításában valahol a Swing és az AWT keveredéséből jön létre. Alapvetően natív megjelenítéseket használ, de ahol ez nem megoldható, mert nem áll rendelkezésre a megfelelő rendszerkönyvtár, ott a saját kódját használja a Swinghez hasonlóan. Az SWT megjelenésében az AWT-re, eszközrendszerében viszont a Swingre hasonlít.

1.3. Swing alapok

Hogy jobban megértsük az SWT eszközrendszert, meg kell ismerkednünk a Swing alapjaival. A Swing megjelenése után hamar az egyik legnépszerűbb GUI programozási eszköz lett. Az idő múlásával azonban a fejlesztők elégedetlenné váltak, mivel a Swing által létrehozható GUI elemek leprogramozása bonyolult, és az eredményül kapott program lassan fut. A közismert MVC (Model-View-Controller) tervezési minta helyett a Swing az úgynevezett modell-delegált ar-

chitektúrát alkalmazza, ami a nézet és vezérlő aspektusokat egy UI-delegáltként egyesíti [1]. A 2. ábrán látható a Swing modell-delegált architektúrája.



2. ábra

Ily módon minden GUI elem esetén a Swing egy modellnek foglal memóriaterületet, mely modell tartalmazza a komponens állapotát, és az UI-delegáltat, ami megjelenésért, és az eseményekért felelős. Azzal hogy a szétválasztotta a modellt a megjelenéstől, a Swing programozási eszköz flexibilis, újrahasznosítható kódot eredményez. Mindazonáltal ezzel minden egyes képernyőn megjelenő alkalmazáselem esetén számos objektum áll elő, és ahogyan a GUI egyre komplexebbé válik, úgy egyre nagyobb terhelésnek teszi ki a virtuális gépet.

2. Az SWT

A Swing komplexitására, és futási sebességére reagálva az Eclipse tervezői létrehoztak egy olyan eszközt, amivel olyan Java felhasználói környezet állítható elő, mely ugyanolyan sebességgel fut, mint a natív alkalmazások. Az SWT és az operációs rendszer közötti kommunikáció itt is a JNI segítségével történik. A Swinggel ellentétben az SWT memóriakezelése nem automatikus szemétgyűjtésen alapszik.

Ez két okból volt szükséges:

- A szemétgyűjtés előre nem látható folyamat. Ha valamilyen rendellenesség lép fel felszabadítás közben, lehet, hogy a folyamat félbeszakad, és nem ér véget.
- Az automatikus szemétgyűjtést az operációs rendszer erőforrásaival megoldani nehézkes feladat. A Swing esetén ez könnyen megoldható, mivel az magas szintű komponenseket használ, de az SWT esetében ezt rendszerhívásokon keresztül kellene véghezvinni.

A fenti problémákra válaszként a fejlesztők az automatikus felszabadítás helyett a fejlesztőkre bízták a nem használatos elemek utáni takarítást. A fejlesztő a `dispose()` metódus meghívásával felszabadítja az adott eszköz által lefoglalt erőforrásokat. Ha egy komponenst felszabadítunk, akkor a benne található komponensek is automatikusan fel lesznek szabadítva.

A Swing esetén említett modell-delegált architektúra alkalmazása nehézkes, és egyes egyszerű felületek esetén feleslegesen kifinomult. Az SWT nem szab meg szabályokat a komponensek tervezését illetően. Ez azt jelenti, hogy tetszés szerinti kifinomultsággal, vagy egyszerűséggel lehet kódolni. Számos eszköz létezik az MVC minta SWT alatti megvalósításának támogatására, ilyen például a JFace eszközrendszer.

2.1. Egy form megvalósítása SWT-ben

Ahhoz hogy bármiféle GUI-t létrehozzunk szükséges néhány alapvető lépést megtennünk. A következőkben az `org.eclipse.swt`, és az `org.eclipse.swt.widgets` csomagok lehetőségeit fogjuk megvizsgálni.

A `Display` osztály segítségével hozhatjuk létre a kapcsolatot az operációs rendszer és a felület között. Ez az osztály kezeli az ablakok kirajzolását, mozgatását, újrarajzolását, és gondoskodik arról, hogy a különféle események tovább legyenek küldve az alkalmazásnak, amely kezeli őket. A `Display` végzi az SWT komponenseknek a natív eszközkészletekkel történő kezelését.

Míg a `Display` osztály feladata az ablakok kezelése, addig a `Shell` osztály a GUI-k fő ablakáért felelős. Ez az osztály kezeli az ablak átméretezését, megnyitását stb. A osztály fő funkciója, hogy összekapcsolja az ablakba beépített elemeket. Ebben a felfogásban a Shell az összetevők szülője. Minden SWT-ben írt alkalmazásban léteznie kell egy fő Shell példánynak, de ettől függetlenül egyéb `Shell` objektumok létezhetnek, melyek a fő ablak gyermekeinek tekinthetőek. Ezeket az objektumokat másodlagos shelleknek nevezzük, mivel ők nincsenek közvetlen kapcsolatban a `Display` példánnyal. A `Display` osztállyal közvetlen kapcsolatban levő shelleket legfelső szintű shelleknek nevezzük. A shellek komponenseinek kinézetét, karakterisztikáját úgynevezett stílusok segítségével adhatjuk meg. Egy shell stílusát a konstruktorban paramétereként adhatjuk meg. Ezek a stílusok az SWT osztályban konstansok formájában találhatók meg, és bitenkénti vagy művelettel tetszőlegesen kombinálhatóak. A megjelenésen kívül a stílusok segítségével modalitásokat is rendelhetünk a grafikus elemhez, amely megszorításokat tehet arra vonatkozóan, hogy a felhasználó megváltoztathat e bizonyos állapotokat a shellen (pl. egy ablak átméretezhetősége).

2.2. A Widget és a Control ősosztályok

A `Widget` az absztrakt ősosztálya minden olyan osztálynak, amely információt jelenít meg, illetve felhasználói felület megvalósítását teszi lehetővé. Ezen osztályban találhatók meg az egyes elemek közötti kommunikációhoz, illetve ezen elemek felszabadításához szükséges metódusok.

2.2.1. A Control absztrakt osztály

A `Control` absztrakt osztály a `Widget` osztály alosztálya. Mivel a különböző operációs rendszerekben eltérő GUI elemek lehetnek, ezért csak azokat az elemeket lehet natív módon elérni, amelyeket az adott operációs rendszer ismer. Ezeknek az elemeknek a listáját a `Control` osztály `handle` mezőjében találjuk meg. A legáltalánosabb GUI elemek, mint a `Label`, `Button` és `Composite`

ezen osztály alosztályai. Egy `Control` objektum segítségével megtudhatjuk, hogy mekkora az úgynevezett preferált méret, ami azt mondja meg, hogy mekkora a minimális terület, ahol elférnek a rajta található elemek. A `pack()` metódus segítségével az alkalmazásunkat erre a preferált méretre állíthatjuk be, ami nagyon hasznos lehet, tekintve hogy nem tudjuk előre, hogy milyen operációs rendszeren, illetve milyen felbontáson fogják használni az alkalmazást.

2.2.2. Címkék

A `Label` osztály talán a legegyszerűbb alosztálya a `Control` osztálynak. A címke olyan statikus információt jelenít meg a GUI-n, mint pl. egy szöveg, vagy egy kép. A címkék megjelenését a `Shell`-hez hasonlóan itt is stílusok segítségével szabályozhatjuk. A szöveges címkék elrendezéséhez használhatjuk az `SWT.CENTER`, `SWT.LEFT` és `SWT.RIGHT` értékeket a konstruktornak paraméterként megadva, vagy utólag a `setAlignment(int alignment)` metódust meghívva. Egy címke szöveges tartalmát a `setText(String string)` metódus segítségével állíthatjuk be. Hasonló módon, a `setImage(Image image)` metódus szolgál egy kép címkéhez való hozzárendelésére.

2.2.3. Gombok

A gomb az egyik legegyszerűbb elem, amely a felhasználó interaktivitását lehetővé teszi. A `Button` objektumok szintén egyszerű komponensek, hiszen bináris értékűek: vagy bekapcsolt állapotban vannak, vagy kikapcsoltban. Természetesen ennél többet is megtehetünk eseményalapú vezérlés segítségével, de a gomb mint objektum ennyire hivatott. A különböző gombok létrehozásához öt alapvető stílust nyújt az SWT. A gomb feliratát a címkével azonos módon, a `setText(String string)` metódus segítségével állíthatjuk be, és ráadásként a szövegformázási stílusokat, illetve a `setAlignment(int alignment)` is alkalmazhatjuk rá. Természetesen a szöveggel analóg módon a `setImage(Image image)` metódussal képet is elhelyezhetünk egy gombon. Az egyik tipikus gomb a nyomógomb az `SWT.PUSH` konstans segítségével állítható elő. Ha nem adunk meg stílust a konstruktorban, akkor alapértékként ezt a beállítást kapja. Egy másik gyakran használatos gombtípus a kapcsológomb, amely megtartja bekapcsolt állapotát, amíg újra meg nem nyomják. A hozzá tartozó konstans az `SWT.TOGGLE`. Állapotuk a `setSelection(boolean selection)` segítségével módosítható.

A jelölőgombok funkciójukat tekintve a kapcsológombokhoz hasonlatosak, és általában funkciójukat illetően csoportokat alkotnak. Az ilyen eszközök az `SWT.CHECK` konstans segítségével konstruálhatók. Ezeket tehát érdemes lehet valamilyen kollekcióban kezelni. Állapotuk a `getSelection()` segítségével lekérdezhető, a `setSelection()` segítségével pedig módosítható.

A rádiógombok olyan gombcsoportok, amik közül csoportonként mindig csak egy gomb lehet aktivált. A hozzájuk tartozó stílusérték az `SWT.RADIO`. Mivel ezek nagyon hasonlóak az előző jelölőgombokhoz, ezeket is érdemes valamilyen kollekcióba összegyűjteni.

2.2.4. Komponensek Composite-ba gyűjtése

A tartalmazó eszközök jelenléte szükségszerű az alkalmazásokban, hiszen ezek adják a belső felépítését a felületnek. A `Composite` objektumok segítségével jól szervezetté tehetjük az alkalmazást a kódban és a felhasználói felületen egyaránt. Ahogyan más `Control` objektumok, ez is átméretezhető és pozícionálható az alkalmazásban. A `Composite` metódusai segítségével lekérdezhetjük az ő általa tartalmazott objektumokat, beállíthatjuk és lekérdezhetjük a felületi elrendezését (layout), és tab körbejárási sorrendjét. A tab körbejárási sorrend azt mondja meg, hogy a felhasználó a billentyűzetén található tab billentyű lenyomása segítségével milyen sorrendben halad végig a `Composite` elemein. A `Composite` osztály közvetlen leszármazottja a `Scrollable` osztálynak, így elhelyezhető rajta a `ScrollBar` osztály példánya, melynek hatására görgetősávja is lesz az objektumunknak.

2.2.5. A Group osztály

A `Group` osztály a `Composite` egy közvetlen leszármazottja. Segítségével csoportokba rendezhetjük az elemeinket, és négyszögletes szegélyt rajzolhatunk az elemek köré, amit egy címkével elnevezhetünk. Ennek az eszköznek a segítségével tovább finomíthatjuk a felületünk elrendezését.

2.3. Eseménykezelés

A felhasználói felületünk események kezelése nélkül csupán egy funkció nélküli dekoráció. Az SWT eseménykezelése: Az operációs rendszernek létezik egy úgynevezett eseménysora, amely rögzíti és listázza a felhasználó interakcióit.

Amikor egy SWT alkalmazás fut, a `Display` osztály kiválogatja az operációs rendszer eseményei közül a programra vonatkozókat a `readAndDispatch()` metódusa és a `msg` mezője segítségével. Ha talál ilyen eseményt, akkor azt elküldi a legfelső szintű `Shell` objektumnak, amely eldönti, hogy melyik elemnek kell fogadnia az eseményt. Ezután a `Shell` elküldi az eseményt ennek az elemnek, amely átalakítja ezt az információt egy hozzárendelt interfésznek, amit listenernek nevezünk. A listener metódusainak egyike végrehajtja a szükséges utasításokat, vagy meghív egy másik metódust, hogy kezelje le a felhasználó tevékenységét. Ezt a metódust eseménykezelőnek hívjuk. Amikor egy felhasználói felület készül, a fejlesztőnek el kell döntenie, hogy mely elemeken milyen események hajtódhatnak végre, és ezeknek megfelelően kell az eseménykezelőt, illetve a listenereket megírni. Az SWT listener interfészei többnyire csak a felhasználói események egy részére reagálnak. Ebből kifolyólag a megfelelő események észleléséhez a megfelelő listener használata szükséges. Pl. az egérkattintás egy egér által keltett esemény, `MouseEvent`, amit a `MouseListener` tud fogadni. A listenerek interfészek, melyeket implementálni kell, hogy használhassuk őket. Bizonyos listenerekhez léteznek úgynevezett adapter absztrakt osztályok, amik implementálják az interfészeket. Ezzel sok kódolási időt lehet megspórolni, hiszen ekkor elegendő ahhoz a metódushoz megírni a kódot, ami aktuálisan az alkalmazásunkat érinti.

2.4. Szöveg bevitele

Szöveg kezelésére az SWT alapvetően két lehetőséget kínál. Az egyik az egyszerű, formázatlan szöveg kezelését szolgáló `Text` osztály, a másik a `StyledText`, melynek segítségével különböző formázási beállításokat adhatunk a szöveghez. A `Text` osztályhoz is tartoznak listenerek, amiknek a segítségével a begépelt szöveget validálhatjuk, és ennek megfelelően akár módosíthatjuk is. Ennek segítségével felhasználóbarátabbá tehető az alkalmazás, hiszen a felhasználó nem a teljes űrlap kitöltése után értesül róla, hogy valahol elgépelt valamit, hanem miközben gépel, látja, hogy a szöveg megfelel-e a formai követelményeknek. Ezzel az eszközzel akár a szövegszerkesztőkben megtalálható nyelvhelyesség-ellenőrző is létrehozható. Mint általában minden eszköz, ez is testre szabható stílusok segítségével. Az `SWT.SINGLE` konstans esetén a szövegbeviteli mezőnek egyetlen, míg `SWT.MULTI` esetén több sora van. Természetesen a szövegbeviteli mezőből a szöveg a `getText()` metódussal kiolvasható.

2.5. A Combo eszköz

A `Combo` osztály segítségével hozhatunk létre legördülő menüt. Ez tipikusan egy olyan elem, ami lehetőséget ad a felhasználónak, hogy felsorolt lehetőségek közül válasszon. Három típusát különböztetjük meg:

- Egyszerű: Tartalmaz egy szerkeszthető szöveg mezőt a tetején, és egy lista dobozt az alján a választható lehetőségekkel. Ez az alapértelmezett `Combo` beállítás.
- Lenyíló: Egy szerkeszthető szöveg mező, egy nyíllal a jobb oldalán. A nyilat lenyomva legördül egy opciókat tartalmazó lista.
- Csak olvasható: Egy lenyíló menü, aminek nem szerkeszthető a szöveg mezője. Ezt akkor szokás használni, ha bizonyos előre definiált elemekre szeretnénk korlátozni a felhasználó lehetőségeit.

2.6. Elrendezés (Layout)

Az alkalmazásunkban számos eszközt helyezünk el a felhasználói felületen, amiknek meg kell határoznunk az elrendezését. Az SWT-ben a `Layout` az, ami a felületen való rendezettséget biztosítja. A `Layout` a `Composite`-hoz van hozzárendelve, tehát minden egyes `Composite` együttes esetén különböző lehet az elrendezés, de azon belül egységes marad. A `setLayoutData` segítségével állíthatjuk be az elrendezés paramétereit.

2.6.1. A kitöltés elrendezés

Ebben az esetben a rendelkezésre álló területet annyi részre osztjuk, ahány eleme van a `Composite`-nak, és úgy pakoljuk bele őket. Alapértelmezetten balról jobbra haladva kerülnek bele a felosztásba az elemek. Ezt az elrendezést a `FillLayout` osztály segítségével valósíthatjuk meg, aminek paraméterül két stílus egyikét adhatjuk meg: `SWT.HORIZONTAL` a vízszintes, balról jobbra rendezettséghez, és `SWT.VERTICAL` a függőleges, fentről lefelé való kitöltéshez. Az elrendezést egy `Composite` elemen a `setLayout(Layout layout)` metódus segítségével alkalmazhatjuk.

A sor elrendezés (row layout): A sor elrendezés esetében a fejlesztő bizonyos számú sorokra, illetve oszlopokra oszthatja a rendelkezésre álló területet. Ezt a megjelenítést a `RowLayout` osztály segítségével hozhatjuk létre. Ennél a kiosztásnál lehetőség van margó és sorköz megadására. Egy cella továbbosztására

akkor van lehetőség, ha az `SWT.WRAP` stílust alkalmazzuk. A kiosztás további beállításához rendelkezésre állnak a stíluson kívül az osztály bizonyos mezői:

- `wrap`: Egy logikai érték, ami alaphelyzetben true. Kikapcsolása esetén, ha az elem túlnyúlik a cella méretén, akkor ez a rész levágásra kerül.
- `pack`: Egy logikai érték, alapértéke true. Kikapcsolása esetén minden cellának ugyanakkora mérete lesz, mint amekkora cella ahhoz kell, hogy a legnagyobb elem elférjen benne.
- `justify`: Logikai érték, alaphelyzetben false. Azt határozza meg, hogy ha a szülőobjektum mérete megváltozik, akkor megváltozzon-e a tartalmazottak mérete is.

2.6.2. A rács elrendezés

A rács elrendezés a sor elrendezésen alapszik. Lehetőséget ad, hogy a területet tetszőleges számú oszlopra, illetve sorra osszuk. A `GridLayout` osztály konstruktora két paramétert vár, az első megmondja, hogy hány oszlopa lesz az elrendezésnek, a második pedig egy logikai érték, ami megmondja hogy a cellák szélessége azonos legyen-e.

2.6.3. Az űrlap elrendezés

Az előző elrendezések mind azon alapultak, hogy cellákra osszák fel a rendelkezésre álló területet. Ez az elrendezési mód eltérően az előzőektől az elemek egymáshoz való viszonyán alapszik. A `FormLayout` osztály elég egyszerű. Az egyetlen beállítási lehetőség az elemek egymástól való távolságának megadása. Hogy minden egyes elem a megfelelő helyre kerüljön, egy `FormData` példánnyal le kell írni a rájuk vonatkozó rendezettséget. Ha egy elemhez nincs `FormData` hozzárendelve, akkor automatikusan a tartalmazó elem jobb felső sarkába kerül. A `FormAttachment` osztály segítségével az elemek közötti kapcsolatot írhatjuk le, ami az elem négy oldalához és a tartalmazó objektumhoz, vagy bizonyos egyéb tartalmazott elemekhez viszonyított távolságot definiálja százalékban vagy pixelekben megadva.

2.6.4. Tesztre szabott elrendezés

Az alap elrendezések többnyire biztosítják azt a rendezettséget, amit a fejlesztő kapni szeretne, de bizonyos esetekben szükség lehet saját készítésű rendezettség létrehozására is. Ez viszonylag ritkán fordul elő, de az SWT-ben erre is van lehetőség. Egy ilyen rendezettségi osztály létrehozásához egy, a `Layout-`

ot kiterjesztő osztályt kell létrehozni, és ebben következő két metódust implementálni: `computeSize()` és `layout()`. A `computeSize()` metódust a tartalmazó eszköz hívja meg, hogy kiszámítsa mennyi helyre lesz szüksége az elrendezés kialakításához. Ezután a `layout()` metódus kerül meghívásra, hogy elrendezze az elemeket. Ezen metódusok részletezése most hely szűke miatt nem kerül tárgyalásra.

2.7. Táblázatok

Sok esetben lehet szükség táblázatok alkalmazására. Többnyire egy adatbázissal dolgozó szoftver így jeleníti meg a lekérdezéseket a felhasználó számára, aki a táblázatot egy kétdimenziós, sorokból és oszlopokból álló rácsként látja. Az SWT megfelelő komponense a `Table` osztály. Az osztályra jellemző, hogy amíg az állapot lekérdezésére viszonylag számos lehetőséget nyújt, addig a táblázat testre szabására nem sok eszköz áll rendelkezésre. Valójában az elemek közvetlen behelyezése helyett, a tartalmazottak példányosításukkor rendelhetőek a táblázathoz. A táblaelemek létrehozására a `TableItem` osztály ad lehetőséget. Minden egyes példány egy egész sort reprezentál a táblázatban, amely tartalmazhat szöveget és képet egyaránt. A `TableItem` tartalmát a `setText(int column, String string)`, illetőleg a `setImage(int column, Image image)` metódusok segítségével adhatjuk meg, ahol az első paraméterként megadott szám a sor oszlopának sorszámát jelöli, a második paraméter pedig értelemszerűen az elhelyezni kívánt tartalom. A `TableItem` konstruktora paraméterül vár egy táblázatot, amibe kerül, és egy stílust. Jelen esetben kivételesen ez a stílus kötelezően `SWT.NONE` kell, hogy legyen. A konstruktor által létrehozott elem automatikusan a táblázat utolsó eleme lesz.

Egy táblázat oszlopait a `TableColumn` osztály segítségével hozhatjuk létre, amely minden esetben egyetlen oszlopot fog hozzáadni a hozzárendelt táblához. Az oszlop testre szabására metódusok adnak lehetőséget. Az oszlop fejlécének elnevezésére pl. a `setText()` metódus szolgál. A táblázat oszlopainak tulajdonságának megadása egyszerűbben véghez vihető a `TableLayout` osztály segítségével, az `addColumnData()` metódus meghívásával. A metódus alkalmazásához a `ColumnWeightData` példány a kulcs, amely lehetőséget nyújt a relatív szélesség beállítására, így nem kell pixelszámokkal bajlódni. Ezt a lehetőséget használva tehát a következő lépések elvégzésével juthatunk el a kívánt táblázathoz:

- `TableLayout` példányosítása
- A `TableLayout` példány `addColumnData(ColumnWeightData columnWeightData)` metódusának kívánt paraméterezésével való meghívása
- Táblázat létrehozása a `Table` osztály példányosításával
- A példányosított `Table` objektum `setLayout(Layout layout)` metódusának alkalmazásával a szükséges elrendezés kialakítása
- A tábla oszlopainak létrehozása a `TableColumn` segítségével
- A tábla elemeinek létrehozása a `TableItem` példányosításával,
- `setText()` illetőleg `setImage()` metódusok segítségével rendelhetünk tartalmat az elemekhez.

2.8. Az eszközök együttes alkalmazása

A fentebb ismertetett eszközök segítségével már elég sokféle GUI-t létrehozhatunk. Az egyszerű eszközök együttes használata segítségével tetszőleges bonyolultságú, testre szabott felhasználói felületet kaphatunk, ami a natív eszközök használatának köszönhetően gyors és jól illeszkedik az operációs rendszer egyéb elemeihez.

2.9. JFace

Az SWT eszközrendszerével ugyan tetszőleges összetettségű és működésű felület létrehozható, ám ezek fejlesztése bizonyos helyzetekben nehézkessé válhat. Ennek egyszerűsítésére hozták létre a JFace eszközrendszert, amely az SWT eszközeit használva osztályokba és interfészekbe gyűjt bizonyos szolgáltatásokat. Ezeket modell alapú adaptereknek vagy segédosztályoknak szokás nevezni. Ezek lényegében négy nagyobb kategóriára oszthatóak.

Az első, és talán leggyakrabban használt kategória, a `Viewer` osztályoké. SWT-ben az információ és a megjelenítés egybetartozó dolgok, nem választhatók külön, míg JFace segítségével ugyanaz az információ más-más megjelenési formában is megjeleníthető. Például az SWT-ben egy fa objektum elemei nem jeleníthetőek meg másként, míg JFace-ben ugyanazon információ megjelenhet `TreeViewer` vagy `TableViewer` segítségével is. Ezzel az eszközzel az MVC tervezési minta könnyebb megvalósítására van lehetőség.

A második kategória az eseményekhez kapcsolódik. Elválasztja egymástól az eseményeket és a felhasználó által kiadott parancsokat, amelyek ezeket az eseményeket kiváltják. Például ha SWT-ben négy azonos kontrollon ugyanazt az eseményt kell kiváltani, akkor mindegyik esetében meg kell hívni az eseménykezelőt, míg JFace-ben ezek a kontrollok összevonhatók.

A harmadik kategória a képek és betűtípusok bejegyzésére, nyilvántartására vonatkozik. Mivel SWT-ben minden hasonló jellegű hivatkozás operációs rendszer erőforrást igényel, és saját kezűleg gondoskodni kell a használat végeztével az erőforrás felszabadításáról, ezért szükséges ezek megfontolt használata. JFace segítségével ezek az erőforrás lefoglalások automatizálhatóak, tehát nem szükséges kézzel végrehajtani a szemétgyűjtést.

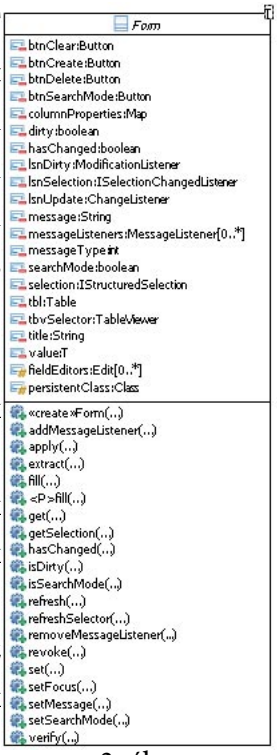
Az utolsó csoport a dialógusok és varázslók csoportja. A JFace kiterjeszti az SWT által nyújtott dialógus lehetőségeket, amik segítségével üzeneteket jeleníthetünk meg, vagy tájékoztathatjuk a felhasználót a folyamat jelenlegi állásáról. A JFace továbbá tartalmaz egy eszközt, amely segítséget nyújt a felhasználónak az olyan lépésről lépésre végrehajtandó folyamatok közben, mint például a telepítés. Ezt az eszközt szokás varázslónak nevezni.

Míg az SWT-ben az ablakokat a Shell osztály adja, addig a JFace-ben az `ApplicationWindow` a GUI alapja, amely nagyobb szabadságot nyújt a testre szabásban, és lehetőséget nyújt az egyéb elemek, mint például menü, eszköztár és hasonlók könnyű integrálására is.

Egy összetett felhasználói felület létrehozását tehát célszerű az SWT és JFace eszközrendszerek ésszerű kombinációja segítségével megvalósítani.

3. A HiberGUI

A HiberGUI egy olyan eszközrendszer, amely SWT és JFace alapokon nyugszik. Az űrlapok létrehozását segítendő jött létre. A JFace-hez hasnlóan ez is modell alapú adaptert tartalmaz. A HiberGUI használata során természetesen minden SWT-beli, vagy egyéb Java-beli elem használható. Az eszközrendszer kiemel bizonyos dolgokat, melyek tipikusan előfordulnak egy űrlap jellegű felhasználói felületen. Mivel az eszközrendszer alapjában véve adatbázis alapú rendszerekhez készült formok létrehozásának segítését célozza, ezért bizonyos metódusokkal (`save, delete, update`...) felüldefiniálhatjuk a form által kezelt objektumok elmentésének, törlésének stb. folyamatát. A HiberGUI-ban két fontos ősosztály található, amelyek kiterjesztésével a form, illetve annak az azon megtalálható összetevők hozhatóak létre. Az egyik ilyen ősosztály az absztrakt Edit osztály, amely segítségével az összetevők, mint pl. szövegbeviteli mező, legördülő menü stb. hozhatók létre. A másik ilyen ősosztály a Form ősosztály. Ezen osztály UML diagrammja a 3. ábrán látható. Ennek egy gyermeke a `HibernateForm`. Egy HiberGUI-ban létrehozott formot megvalósító osztálynak ki kell terjesztenie a `HibernateForm` generikus absztrakt osztályt, és felül kell definiálnia bizonyos metódusokat.



3. ábra

Az egyik metódus a `createFields`, amely segítségével a felület fő Composite eleme hozható létre. Paraméterként a szülő `Composite`-ot várja, amelyre majd a form kerül. Ezen metóduson belül a formon elhelyezhetünk egyéb `Composite`-okat, `Group`-okat, meghatározhatjuk ezek elrendezését, illetve feltölthetjük őket a megjelenítendő elemekkel, mint pl. címke, vagy szövegbeviteli mező.

A másik ilyen felüldefiniálandó metódus a formon található táblázat létrehozásához szükséges `createColumnsForTable`, melynek oszlopait az `addColumn()` metódus segítségével tölthetjük fel. Ez utóbbi paraméterezése a következőképpen néz ki: egy stílus segítségével megadhatjuk az igazítást, egy

sztring tulajdonság segítségével hivatkozhatunk egy adott elemre, egy sztring segítségével elnevezhetjük az oszlopot, és végül egy pixelszámmal megadhatjuk az oszlop szélességét. A `createFields()` metódusban sorban definiált elemek a formon azonos sorrendben, az elrendezésnek megfelelően fognak megjelenni. Ezek az elemek jellemzően címkék, illetve valamilyen felhasználói interakciót megvalósító eszközök, mint például szövegbeviteli mező, legördülő menü, rádiógomb stb. A HiberGUI eszközeit használva az eseménykezelés már nem feladata a programozónak, a kitöltött mezők értékének továbbadása automatikusan megtörténik. Az osztály konstruktorában a `setWindowTitle()` metódus segítségével nevezhetjük el az ablakunkat.

4. Az XML

Az XML az Extensible Markup Language rövidítése, az SGML (Structured Generalized Markup Language) egy leszármazottjaként létrejött jelölőnyelv. Az SGML egy általános jelölőnyelv, amely saját elemkészlettel rendelkező dokumentum leírást tesz lehetővé. Az SGML-ben íródott dokumentumok feldolgozása kicsit nehézkes volna, mivel túlságosan általános. Az XML-t a W3C fejlesztőcsapata hozta létre. Ebben az elemek létrehozására szigorúbb szabályok vonatkoznak, így könnyebben tanulható és feldolgozható. Az XML dokumentumok olyan szöveges fájlok, melyek az XML szabványa szerint pontosan definiáltak. Egy XML dokumentumot jól formáltnak (well-formed) nevezünk, ha teljesíti az alábbi követelményeket:

- Egyetlen elem tartalmazza az egész objektumot (ezt nevezzük gyökérelemnek)
- Minden nyitócímkéhez tartozik egy megfelelő zárócímke
- Az egyes elemek egymásba ágyazottak lehetnek, de nem fedhetik át egymást (minden beágyazott elemet a beágyazó elem előtt le kell zárni)
- Minden a dokumentumban hivatkozott egyed jól formált

Az XML dokumentumban használható érvényes karakterek: a tab, kocsi vissza, új sor, és a Unicode illetve ISO/IEC 10646 érvényes karakterei [3]. A szabvány azt is definiálja, hogy ezen karakterek közül mely karakterek tekintendőek számnak, és melyek betűnek. A nevekben megengedettek a `.`, `-`, `_` és `:` karakterek

A nevek olyan karaktersorozatok, melyek betűvel, `_`, vagy `:` karakterrel kezdődnek, és a nevekben megengedett karakterekkel folytatódhatnak. Kivételt képez ez alól az `XML` karaktersorozat bármely kis- és nagybetűs kombinációja, illetve az ezzel kezdődő karaktersorozatok, mert azok nem használhatóak. A nevekben előforduló `:` karakter az XML feldolgozó számára speciális jelentést hordoz, névtereket definiál, használatát ennek megfelelően illik szem előtt tartani.

Literálnak nevezzük azokat a karaktersorozatokat, melyek `"` vagy `'` jelek által határoltak, és nem található meg bennük az elhatároló jel. Jól formált literál például a `'"Literál"'`, `"Literál"`, vagy a `"'123'"`, de nem jól formált a `'Literál"`.

Az XML nagyon fontos részét képezik a címkék. A nyitó címke szintaxisa:

```
<név [attribútum]…>
```

A záró címke szintaxisa a következő:

```
</ név>
```

Egy címkét üreselem címkének nevezünk, ha nincs benne tartalom. Az üreselem címkének külön szintaxisa van:

```
<név [attribútum]… />
```

A nyitó és üreselem címkékben egy adott nevű attribútum legfeljebb egyszer szerepelhet. A címkékben megadott attribútumok felsorolásának sorrendje lényegtelen. A nyitó és záró címke közötti szöveget az elem tartalmának nevezzük.

Az XML-ben jól definiált szabályokat nem betartó kódrészek leírását teszik lehetővé a `CDATA` szakaszok. A `CDATA` szakasz szintaxisa a következő:

```
<![CDATA[ cdata szekció ]]>
```

A `CDATA` részek bárhol előfordulhatnak, ahol karakteres adat előfordulhat. A `CDATA` részen belüli szövegrész karakteres adatként értelmeződik, így bármilyen tartalmat átadhatunk a feldolgozó egységnek.

Az XML-ben elhelyezhetünk megjegyzéseket is, a következő módon:

```
<!-- megjegyzés -->
```

A megjegyzésen belüli tartalom nem kerül értelmezésre, az a kód olvasójának, illetve szerkesztőjének szóló, megértést segítő szövegrész.

Minden XML dokumentum az XML deklarációval ajánlott, hogy kezdődjék. Ebben adjuk meg, hogy miként kell értelmezni a szövegben leírt kódot. Ez a deklaráció gyakorlatilag egy fejléc, ami a következőképpen néz ki:

```
<?XML version=”1.0” [encoding = ”karakterkódolás”]
[standalone=”yes”|”no”]>
```

Jelen esetben a verziószám 1.0, mert ezt a szabványt tárgyaljuk. Az `encoding` rész mondja meg, hogy milyen karakterkódolással íródott a kód. Használata kötelező, ha UTF-8-tól vagy UTF-16-tól eltérő karakterkódolást használunk. A

`standalone` mező megadása opcionális, lehetséges értékei a `yes` és a `no`. Ha nem adjuk meg ezt a mezőt, akkor alapértelmezetten `no` lesz a beállított érték. Ez az opció közli az XML feldolgozó modullal, hogy a dokumentum tartalmazni fog külső jelölésdeklarációt („no” érték esetén). Ha ezt beállítjuk, de a feldolgozó mégsem talál ilyet, akkor hibajelzést ad. Ellenkező esetben azt közöljük a feldolgozóval, hogy nem tartalmaz jelölésdeklarációt [3].

5. A DTD

A DTD (Document Type Declaration, azaz a dokumentumdeklaráció) segítségével érhetjük el, hogy a dokumentumunk ne csak jól formált, hanem érvényes is legyen. Ez tehát egy validációs eszköz, mely szabályokat ír le, aminek meg kell felelnie az XML kódnak. Ezek a szabályok logikai szerkezetre és tartalomra vonatkoznak. A DTD szabályokat összegyűjthetjük egy fájlban, vagy megadhatjuk az XML-en belül is.

Külső DTD megadása a következőképpen történik:

```
<!DOCTYPE név SYSTEM literál>
```

ahol a literál egy URI hivatkozás. Vagy megadható a következőképpen:

```
<!DOCTYPE név PUBLIC literál1 literál2>
```

Itt a literál1 egy nyilvános azonosító, a literál2 pedig egy URI hivatkozás. Ha a DTD-t helyben definiáljuk, akkor a következőképpen járhatunk el:

```
<!DOCTYPE név [ jelölésdeklarációk ]>
```

Az előző kettő együtt is alkalmazható. Ez a következő szintaxissal érhető el:

```
<!DOCTYPE név SYSTEM literál [ jelölésdeklarációk ]>
```

vagy

```
<!DOCTYPE név PUBLIC literál1 literál2 [
jelölésdeklarációk ]>
```

A DTD segítségével a következőképpen definiálhatjuk a szabályokat. Egy elemtípus deklarációja a következőképpen néz ki:

```
<!ELEMENT név tartalom>
```

ahol a név az éppen deklarálni kívánt elem neve, a tartalom pedig megadja, hogy milyen tartalommal rendelkezhet az adott elem. A tartalom következők egyike lehet:

- `EMPTY` – üres elem, ami sem szöveget, sem gyermeket nem tartalmazhat.

  Pl.: `<!ELEMENT URES EMPTY>`

- `ANY` – bármilyen tartalom megengedett (szöveges, elemes, vegyes), de figyelnünk kell arra, hogy ha gyermekelemeket engedünk meg, akkor azokat deklarálnunk kell.

  Pl.: `<!ELEMENT BARMI ANY>`

- A megengedett elemtartalmakat fel is sorolhatjuk:

  pl.: `<!ELEMENT (ELEM [, ELEM]…)>`

- Vegyes tartalom esetén a deklarált elemeket is tartalmazhatja meghatározatlan számban.

  pl.: `<!ELEMENT (#PCDATA, ELEM) [?|+|*]>`

Ha az elem gyermekelemet tartalmazhat, akkor az összes gyermekelemet fel kell tüntetni, amely előfordulhat, és mindet deklarálni is kell. Az előfordulások számosságát korlátozhatjuk. Ha az elem neve mögött egy `?` áll, akkor az elemből nulla vagy egy előfordulás megengedett. Ha `+` karakter áll az elem neve után, akkor egy, vagy több előfordulás lehetséges. Végül, ha * karaktert helyezünk az elem neve mögé, akkor azzal a nulla vagy több előfordulást engedélyezhetjük. Példa:

```
<!ELEMENT NEV (ELEM1+, ELEM2, ELEM3*, ELEM4?)>
```

Nem minden esetben lehet kötelező, hogy egyes elemek előforduljanak. Ekkor az előfordulások között megengedő feltételt tudunk biztosítani a `|` karakter közbeiktatásával.

A szintaxis tehát a következő:

```
<!ELEMENT NEV (ELEM1 | ELEM2 [| ELEM]…)>
```

Ekkor tehát az elemek közül valamelyiknek elő kell fordulnia, hogy a DTD ellenőrzésen ne akadjon fent a kód. Az előző két deklarációs módot természetesen keverhetjük is, így bonyolultabb kifejezéseket is leírhatunk velük. Pl.:

```
<!ELEMENT NEV ((ELEM1 | ELEM2)+, ELEM3)*>
```

Az elemekben használt attribútumokat is deklarálni kell, ennek szintaxisa a következő:

```
<!ATTLIST NEV attr_név, attr_típus, alapért_dekl>
```

ahol a NEV azt mondja meg, hogy melyik elem attribútumait definiáljuk. Ezután következnek a deklarációk. Az attribútum nevének és típusának megadása után az attribútum alapértelmezés deklarációja következik, amely megmondja, hogy kötelező–e megadni az adott attribútumot, és ha nem, akkor milyen értéket rendeljen hozzá alapértelmezetten. A jellemzőknek háromféle értéke lehet:

– Karakterlánc : `#PCDATA` – az ilyen típusú elem bármit tartalmazhat, értékét karaktersorozatként megkapja az értelmező.
– Token típus : A tartalma bizonyos korlátok közé van szorítva.
  - ID: az ilyen jellemző tartalmának olyan névnek kell lennie, amely egyedi, nem lehet két azonos ID típusú jellemzője két különböző jellemzőnek. Pl.:

```
<!ATTLIST ELEM egyedi_ertek ID #REQUIRED>
<ELEM egyedi_ertek=”22022”/>
<ELEM egyedi_ertek=”18456”/>
<ELEM egyedi_ertek=”22022”/>     <!–helytelen!-->
```

  - IDREF : értéke csak egy már létező ID típusú jellemző értéke lehet. Pl.:

```
<!ATTLIST ELEM egyedi_elem ID #REQUIRED>
<!ATTLIST ELEM2 egyedi_elem ID #REQUIRED egyedi_
elem2 IDREF #REQUIRED>
<ELEM egyedi_elem=”22022”/>
<ELEM2 egyedi_elem=”18456” egyedi_elem2=”22022”/>
```

  - IDREFS : segítségével egyidejűleg több – már létező – azonosítóra hivatkozhatunk úgy, hogy az egyes referenciákat szóközzel elválasztjuk egymástól. Pl.

```
<!ATTLIST ELEM egyedi_elem ID #REQUIRED egyedi_
elem IDREFS #REQUIRED>
<ELEM egyedi_elem=”18456” egyedi_elem2=”22022
25183” />
```

  - NMTOKEN : értéke névtoken lehet. Pl.:

```
<!ATTLIST ELEM elem_azon NMTOKENS #REQUIRED>
```

```
<ELEM elem_azon=”id-123-422-ab” />
```

- NMTOKENS: értékként egyidejűleg több NMTOKEN típusú értéket adhatunk meg, egymástól szóközzel elválasztva.
- ENTITY: értéke olyan egyed lehet, amely már deklarálva lett a DTD-ben, de még nem lett értelmezve.
- ENTITIES : értékként több olyan egyedet adhatunk meg, amelyek már deklarálva lettek, de még nem lettek értelmezve. Egymástól szóközzel választjuk el őket.

– Felsorolás típusú jellemzők esetén megadhatunk egy értékhalmazt. Ha a felvett érték a halmaz egyetlen elemével sem egyezik meg, az XML dokumentum érvénytelenné válik. Az értékhalmaz elemei névtokenek segítségével adhatók meg, melyeket a ’|’ karakter választ el egymástól.

Az alapértelmezés deklaráció lehetséges értékei a következők:

– `#REQUIRED`: ebben az esetben a jellemző megadása kötelező.
– `#IMPLIED`: a jellemző megadásával, illetve elhagyásával is érvényes marad a dokumentum.
– Attribútum értéket is megadhatunk, amit akkor kap az attribútum, ha nem rendelünk hozzá explicit módon értéket.
– `#FIXED`: ekkor a jellemző alapértelmezetten megkapja ezt az értéket ha nem adnak meg explicit módon semmit. Ha explicit módon az alapértelmezettől eltérő értéket állítanak be, akkor érvénytelenné válik a kód.

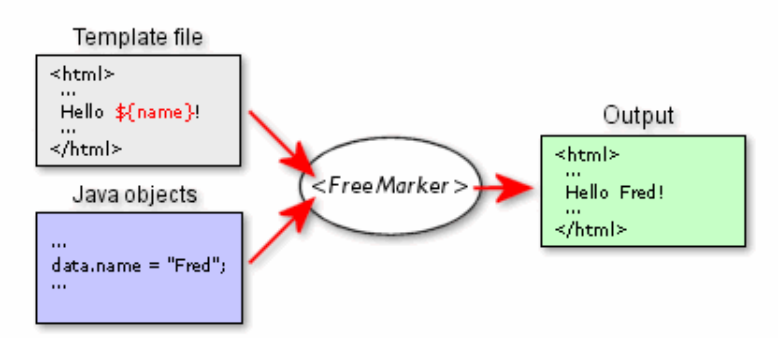
6. Az XML és a Java kapcsolata

Az XML technológia a Java szoftverfejlesztés számos területén megjelenik. A Java kódokban felfedezhetőek olyan szabályszerűségek, melyeket a kódolónak minden azonos funkcionalitású vagy felépítésű program esetén ugyanúgy be kell gépelnie, csak más adatokkal kiegészítve. Az XML alkalmas arra, hogy akár egy komplett programot is generáljunk belőle. Ha az XML-hez megfelelő DTD-t írunk, alkalmas lehet bizonyos problémák automatizált megoldására, mely segítségével egyszerűbben leprogramozhatjuk a problémára megoldást nyújtó programot. Ezt sokféleképpen megtehetjük, de ha a szemünk előtt van néhány jól összeszedett, megfelelően tördelt, hasonló tevékenységet végző Java kód, akkor könnyen észrevehetjük, hogy ha a kódból kivesszük a váltakozó elemeket, akkor egy sablonhoz hasonló kódot kapunk. Ha ezt a szöveget sikerül megfelelő formába önteni, akkor már csak fel kell tölteni a szükséges, változó adatokkal, és eredményül megkapjuk a Java kódot. A sablon megírása sokféleképpen történhet, de a kitöltéséhez szükséges adatok megadásához választhatjuk az XML nyelvet.

7. A FreeMarker

A FreeMarker egy ingyenes, nyílt forráskódú sablon motor. Általános eszköz arra, hogy sablon alapján kimenetként egy szöveget (bármit, a HTML-től kezdve akár a Java forráskódig) generáljunk. Ez egy Java programkönyvtár. Az eszközt úgy tervezték, hogy praktikus legyen vele HTML weblapokat készíteni, különös tekintettek az MVC mintát követő servlet-alapú alkalmazásokra. Annak ellenére, hogy ez az eszköz eredetileg webes fejlesztéshez lett kialakítva, semmit nem tud a webes alkalmazásokról, és tökéletesen alkalmazható más problémák megoldására is.

7.1. A FreeMarker működéséről



4. ábra

A 4. ábrán látható, hogy milyen módon áll elő a célszöveg a sablon és az adatok egyesítéséből. Ha van egy sablon, amely tartalmazza a megfelelő alap kódot, akkor az adatok beillesztését a FreeMarker elvégzi. A sablonban hivatkozásokat kell elhelyezni, amelyek megmondják, hogy mely helyekre mely adatoknak kell kerülnie. Ha például van egy Java kódunk, amiben a felhasználót üdvözölni szeretnénk, akkor az alap kódrészlet a következő:

```
public void udvozles(){
        System.out.println(”Hello World!”);
}
```

Később ezt a függvényt a kódban meghívva üdvözölhetjük a felhasználót (jelenleg a `World`-öt). Ha a `World` szövegrészt ki szeretnénk cserélni, úgy hogy a

felhasználó nevét be szeretnénk helyettesíteni az adat táblánk segítségével, akkor a következőképpen módosul a kódrészlet:

```
public void udvozles(){
          System.out.println(”Hello
${felhasznalo.@nev}!”);
}
```

A `World` kifejezést egy hivatkozással helyettesítettük, mely hivatkozást a FreeMarker kicseréli a hivatkozott adatra, és így előáll a megfelelő kód. Vegyük észre, hogy ezzel az eszközzel még fordítás előtt helyeztük a felhasználó nevét a kódba. Jól látható, hogy a sablon megalkotójának nem kell ismernie, hogy hogyan áll elő a `felhasznalo` néven hivatkozott adat. A sablon szerkesztőjének csak az adott nyelvet kell megfelelően ismernie, amelyen megírt programba később az adatok kerülnek. Ez lehetőséget ad arra, hogy egy hasonló jellegű problémát egyszerre több ember oldjon meg, esetleg kisebb csapatokban. Az egyik félnek csak a sablon segítségével előállítandó célnyelvet kell jól ismernie, míg a másik félnek az adatokat kell megfelelően előállítani, átadni.

Mivel az adatok előállítója és a sablon gyártója akár különböző személyek is lehetnek, ezért az adatok elhelyezkedéséről sem kell tudnia a sablon készítőjének, ellentétben a FreeMarkerrel, amely egy fix elrendezést biztosít a sablon szerkesztőjének. Az adatokat úgynevezett adat-modellbe csomagolja az eszköz. Az adatmodell fa szerkezetű. Az előző példának megfelelően a `felhasznalo.@nev` mellé elképzelhetünk még egyéb adatokat is, mint pl. `felhasznalo.@lakcim` stb.

7.2. A FreeMarker típusai és változói

A FreeMarker a következő skalár típusokat ismeri:

– String: karakterek tetszőleges sorozata, a lenti példában az `egér` az egy ilyen típusú változó. Az ilyen változók értékét mindig ”, vagy ’ jelek között kell megadni.

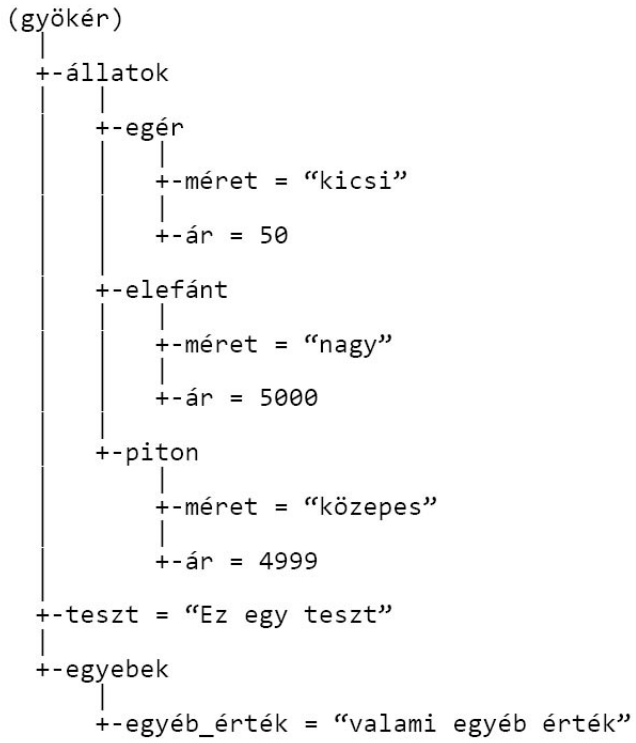
– Number: Numerikus érték, mellyel később számításokat hajthatunk végre. Az ”50” és az 50 értékek FreeMarkerben teljesen különböző jelentést hordoznak. Az első esetben egy sztringet, egy karaktersorozatot adtunk meg, míg a második esetben a változó egy numerikus érték, amivel számolhatunk is.
– Date, Time : Dátum, illetve idő típus. Három változata létezik:
  • Nap pontosságú dátum: Pl.: Április, 4, 2008
  • Idő: Az idő milliszekundumos pontossággal van tárolva. Pl.: 10:19:18 PM
– Dátum, idő : Időbélyegnek is szokás nevezni, az előző kettő együtt van jelen. Pl.: Április, 4, 2008 10:19:18 PM
– Boolean: Logikai típus, két értéke lehet: true vagy false. Segítségével logikai értékeket rendelhetünk a változókhoz.

A FreeMarker ismeri az úgynevezett konténer típusokat is, ezek olyan típusú objektumok, melynek értékei egyéb változók. Gyakran nevezik őket alváltozóknak is.

Ezek a típusok a következők:

– Hashtábla: Minden belső változóhoz egy egyedi hivatkozási nevet rendel. A név egy feltételekhez nem kötött sztring. A hashtábla nem definiál sorrendet a benne foglalt belső változók között. A változókra a nevükkel hivatkozhatunk.
– Szekvencia: Minden belső változóhoz egy egész számot rendel. A változók számukat 0-tól n-1-ig kapják n elemszám esetén. Ezeket a számokat gyakran szokás indexeknek nevezni. A változóknak tehát meghatározott sorrendjük van. A szekvenciák gyakran sűrűk, minden indexhez tartozik elem az elsőtől az utolsóig, de ez nem kötelező. A belső változók típusainak nem kell megegyeznie.
– Kollekció : A sablon alkotójának szemszögéből a kollekció egy megszorításokkal ellátott szekvencia. Nem kérhető le a mérete, nem indexelhetőek az elemei, de a list direktíva (ld. 7.3.2. fejezet) segítségével bejárható.

A könnyebb megértés érdekében vegyük az 5. ábrán látható adatmodellt, melyet fa szerkezetben ábrázolunk.



5. ábra

Az 5. ábrán látható példában található hash változók: gyökér, állatok, egér, elefánt, piton, egyebek. A hashtáblában található belső változók: állat, egér, ár. A skalár értékek: méret, ár, név, teszt, egyéb_érték. Ha egy belső változóra szeretnénk hivatkozni a sablonban, akkor a gyökértől kezdve meg kell adni az elérési utat, `.`-tal elválasztva. Ha tehát hivatkozni szeretnénk a `felhasznalo` elem `nev` elemének `vezeteknev` részére, akkor a `felhasznalo.nev.vezeteknev` módon tudjuk ezt megtenni. Ha még ehhez hozzávesszük a speciális ${…} (ld. 7.3. fejezet) karaktereket, akkor már az értékére hivatkozunk az adatelemnek, így a célszövegben már az érték jelenik meg. A jelenlegi példánkban az állatok egy szekvenciát alkotnak, ahol az elemek sorban az egér, elefánt, piton. Itt tehát indexek segítségével érhetőek el az elemek. Ha pl. az első állat nevére szeretnénk hivatkozni, akkor azt az `állatok[0].név` formában tehetjük meg.

7.3. A sablon

Egy egyszerű sablon bármilyen szöveg lehet. Hogy ezt a bizonyos szöveget dinamikussá tegyük, elhelyezhetünk benne hivatkozásokat, amik helyére a FreeMarker majd behelyettesíti a megfelelő adatelemet. A főbb jelölések a következők:

– ${…} : A { és } jelek között megadhatunk egy változót. A FreeMarker a változó éppen aktuális értékét fogja ebben az esetben behelyettesíteni. Az ilyen jellegű hivatkozásokat interpolációnak nevezzük.
– FTL direktívák: FreeMarker Template Language (FreeMarker Sablon Nyelv) címkék. Ezek kicsit hasonlóak a HTML címkékhez; ezek jelen esetben utasítások az eszköznek, tartalmuk nem fog megjelenni a célszövegben. Ezen direktívák a # karakterrel kezdődnek. Lehetőség van a felhasználónak saját FTL direktívát létrehozni, ezeknek a neve a @ karakterrel kezdődik.
– Megjegyzések: A megjegyzéseket a szövegben bárhol elhelyezhetjük következő formában:

```
<#-- megjegyzés -->
```

Minden, ami nem interpoláció, FTL címke vagy megjegyzés a sablonszövegben, pontosan ugyanolyan módon fog megjelenni kimenetként, mint ahogyan eredetileg áll, beleértve a white space és vezérlő karaktereket is. Az FTL címkék segítségével direktívákra hivatkozunk, ami alatt ugyanazt kell érteni, mint HTML-ben a címkék (pl.: <table> és </table>), és elemek (pl. table) között.

7.3.1 Az if direktíva

Az if direktíva segítségével ki lehet hagyni bizonyos részeket a sablon szövegben. Az általános szintaxis a következő:

```
<#if feltétel>
[Szöveg]
[<#elseif feltétel>
[Szöveg]
[<#elseif feltétel>]…
[Szöveg]
</#if>
```

Az `<#if>` utáni szövegrész akkor lesz figyelembe véve, ha az `<#if>`-ben megadott feltétel kiértékelésekor true értéket kapunk. Ha false értéket kapunk, az `<#if>` és `<#/if>` közötti rész mellőzve lesz, és ha van `<#elseif>` ág, akkor az ebben megadott feltételt értékeli ki. Ha az itt megadott feltétel teljesül, akkor az ehhez az ághoz tartozó szöveg lesz figyelembe véve, majd kilép az elágaztató utasításból az értelmező. Ha nem teljesül a feltétel, és van további `<#elseif>` ág, akkor azokra is ugyanezt hajtja végre. Ha egyetlen feltétel értéke sem true,

és létezik <#else> ág, akkor az annak megfelelő szöveg lesz figyelembe véve. Mindezek után az </#if> után következő részeket értelmezi a FreeMarker.

Például, ha egy szövegben másképp szeretnénk köszönteni a főnököt, mint az egyszerű felhasználót, azt a következőképpen tehetjük meg:

```
A mai dátum: 2008. április 9.
<#if user == ”Nagy Főnök”> Üdvözlöm hőn szeretett
Főnökünket!
<#else> Üdvözlöm Tisztelt látogatónkat!
</#if>
Kellemes napot!
```

A fenti szövegben található „Üdvözlöm hőn szeretett Főnökünket!” szövegrész csak akkor íródik ki, ha a user változó aktuális értéke éppen „Nagy Főnök”. Ettől eltérő esetben mindig az „Üdvözlöm Tisztelt látogatónkat!” felirat íródik ki. A feltétel előtti, és a feltétel utáni szöveg természetesen mindenképpen a kimenet része lesz, hiszen ehhez nem kötöttünk semmiféle feltételt.

A feltételt jelen esetben egy összehasonlító operátorral képezzük, amely a két oldalán található értékek egyezőségét vizsgálja meg. Ha a két érték megegyezik, az operátor true értéket ad vissza, ha különbözik, akkor false-t. Freemarker-ben használhatjuk az == és az = jelölést is az értékek egyezőségének tesztelésére. További összehasonlító operátorok a <, >, <=, >=, !=.

7.3.2 A list direktíva

Szekvenciák, illetve kollekciók bejárására alkalmas direktíva. A szintaxis a következőképpen írható le:

```
<#list szekvencia as elem>
Szöveg
</#list>
```

ahol a szekvencia a bejárni kívánt szekvencia vagy kollekció, az elem pedig egy speciális változó, a ciklusváltozó, amely a szekvencia elemeit kapja értékül. A ciklusváltozó kezdetben értékül kapja a szekvencia első elemét ha van , majd ennek megfelelően kiértékelődik a szöveg tartalma, és ha van további elem a szekvenciában, akkor az elem értékül kapja a következőt. Ha nincs a szekven-

ciának további eleme, akkor a `<#/list>` után folytatódik a kód értelmezése. Ha a szekvenciának nincs eleme, tehát üres, akkor a `<#list>` és `<#/list>` között megadott szöveg egyszer sem kerül értelmezésre. Az iteráció véget ér, ha a szekvencia összes elemét bejártuk, vagy explicit megszakíthatjuk a `<#break>` utasítás segítségével. Ekkor a `<#/list>` után folytatódik a szöveg értelmezése. A ciklusváltozó értékére csak a cikluson belül lehet hivatkozni, a cikluson kívül a változó nem látható. A cikluson belül elérhető két speciális változó:

- `elem_index`: Egy numerikus érték, amely megmutatja, hogy az éppen aktuális elemnek mi az indexe a szekvenciában.
- `elem_has_next`: Egy logikai érték, amely megmondja, hogy az aktuális elem után van e még következő elem a sorozatban.

A direktíva segítségével bejárható számsorozat is, a következőképpen:

```
<#list> a..b as i>
Szöveg
</#list>
```

ahol `a` és `b` egészek. Ebben az esetben az `i` ciklusváltozó kezdetben felveszi az a értéket, majd kiértékelődik a szöveg, és ezután felveszi az aktuális értékénél egyel nagyobbat, ha az nem nagyobb, mint `b`. Minden esetben `a` <= `b` kerül tesztelésre, ha ez nem teljesül, a szövegrész mellőzve lesz.

7.3.3. Az include direktíva

Az include direktíva segítségével egy fájl tartalmát illeszthetjük bele a sablonunkba. Előfordulhat, hogy esetenként bizonyos elemeket különböző sablonokban azonosan használnunk kell. Ekkor ezt a bizonyos ismétlődő részt kiemelhetjük egy fájlba, és ezt a fájlra hivatkozhatunk később a sablonunkban. Természetesen ekkor elég egyetlen helyen megváltoztatnunk a tartalmat, és egyetlen – a fájlt hivatkozó – sablonban sem kell módosítást végrehajtanunk. Ekkor a kimeneti szövegben a fájl teljes tartalma megjelenik.

Az include direktíva szintaxisa:

```
<#include fájlnév>
```

ahol a `fájlnév` annak a fájlnak a neve (elérési úttal együtt), melynek tartalmát a célszövegben látni szeretnénk.

A fenti direktívákat természetesen tetszőleges számban és mélységben egymásba ágyazhatjuk, ismételhetjük.

7.4. Határozatlan értékek esete

A gyakorlatban sok változó értéke lehet opcionális. A FreeMarker nem tolerálja azt az esetet, ha egy változóra úgy hivatkozunk, hogy nincs értéke, hacsak nem mondjuk meg hogy mi ebben az esetben a teendő. Ha egy változót hivatkozunk, és előfordulhat hogy nem kapott még előtte értéket, akkor alapértelmezett értéket rendelhetünk hozzá. Ezt a következő módon tehetjük meg:

```
változónév!alapértelmezett_érték
```

Ha a változónak nem volt megadva értéke, akkor az `alapértelmezett_érték`-et kapja értékül, ha volt megadott érték, akkor a `!alapértelmezett_érték` mellőzve lesz. Egy másik lehetséges megoldás a hiányzó érték problémájára, ha leteszteljük hogy kapott-e értéket a változó. Ennek módja a következő:

```
változónév??
```

Ez egy logikai értéket ad vissza, amely true abban az esetben, ha a változónév nevű változónak van megadott értéke, és false, ha nincs. Ezt egy if direktívában könnyen és jól fel lehet használni.

A fent leírt két direktíva a [2]-ben leírtaknak megfelelően működik. A GUI-t generáló alkalmazás fejlesztése során (ld. 10. fejezet) az említett eszközök nem működtek. A második eszközre talált alternatív megoldás a következő:

```
változónév[0]?exists
```

amely egy true vagy false értékkel tér vissza annak megfelelően, hogy a `változónév` nevű változó létezik e. A változó értékének létezését a

```
@változónév[0]?exists
```

formában tesztelhetjük.

7.5. Függvények és eljárások

A FreeMarker is ismeri az eljárás és a függvény fogalmakat. Ezek alatt az eljárás-orientált nyelvekből ismert hasonló fogalmakat értjük. A függvények használata is hasonló, visszatérési értékét itt is felhasználhatjuk paraméterként, vagy értékül adhatjuk változóknak. Egy függvény deklarálásának szintaxisa:

```
<#function név [paraméter [=kezdőérték]]…>
    Szöveg
    <#return visszatérési_érték>
szöveg
    [<#return visszatérési_érték>
szöveg]…
    </#function>
```

ahol a név az a név, amellyel később a függvényre hivatkozhatunk, a megadható paraméterek formális paraméterek, melyeknek = jel után alapértelmezett értéket adhatunk. Az utolsó paraméter lehet egy szekvencia, amely extra, előre nem meghatározott számú paramétereket tartalmazhat. Ennek jelölése `paraméter...` segítségével történik. A visszatérési_érték az az érték, amelyet a függvény meghívásakor eredményként megkapunk. A return direktíva bárhol kiadható a szövegben. A függvény véget ér, ha eléri a return direktívát, és ekkor a függvény visszatérési értéke az ott megadott érték lesz. Ha a vezérlés a `</#function>` részre kerül, és nem volt előtte return direktíva, a függvény szabályosan befejeződik, visszatérési értéke határozatlan. Példaként egy függvény, ami kiszámolja két szám átlagát:

```
<#function avg x y>
  <#return (x + y) / 2>
</#function>
```

7.6. Az adat rész felépítése

Ahhoz hogy az adatokat a sablonszövegbe be tudjuk illeszteni, el kell készítenünk a megfelelő adatmodellt. Az előzőekben tárgyalt típusok megfeleltetései a következők lehetnek:

- String: java.lang.String

- Number: java.lang.Number
- Boolean: java.lang.Boolean
- Szekvencia: java.util.List vagy tömb típus
- Hash: java.util.Map

Egy egyszerű példaként nézzük meg, hogyan valósítható egy egyszerű adatmodell:

```
// A gyökér elem létrehozása
Map gyoker = new HashMap();
// A ”felhasznalo” sztring gyökérbe helyezése
gyoker.put(”felhasznalo”, ”Nagy Fonok”);
// A hash létrehozása a ”legújabb termék” számára
Map legujabb = new HashMap();
// amit a gyökérbe helyezünk
gyoker.put(”legújabb termék”, legujabb);
//az url és a név hozzáadása a legujabb-hoz
legujabb.put(”url”, ”products/greenmouse.HTML”);
legujabb.put(”name”, ”green mouse”);
```

7.7. A sablonszöveg megnyitása

A sablonszövegek `freeMarker.template.Template` példányként jelennek meg. Valahányszor egy sablonszövegre van szükség, az megnyitható a `getTemplate()` metódus segítségével. Például a test.ftl sablonszöveg beolvasható a következő módon:

```
Template template = cfg.getTemplate(”test.ftl”);
```

Ahol a cfg egy `Configuration` példány, amely a megfelelő beolvasáshoz szükséges beállításokat tartalmazza. A `template` példányban a sablonszöveg nem szövegként, hanem feldolgozott formában tárolódik.

A két fájl egyesítése, azaz a sablonszöveg és az adatok összerendelése a `process()` metódus meghívásával történik. Egy példa ennek használatára:

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

Ekkor a célszöveg a szabványos kimeneten jelenik meg.

8. Az XML feldolgozása FreeMarker segítségével

Technikailag az XML feldolgozás is ugyanúgy működik a FreeMarkerrel, mint bármely más fájl feldolgozása. Csak ki kell olvasni az XML fájlból az adatokat és adatmodellként átadni a sablonnak. Az XML fájlok és a FreeMarkeres adatmodellek szerkezeti felépítése sok hasonlóságot mutat, mivel alapvetően mind a kettő fa szerkezetű.

A DOM fa: A DOM (Document Object Model) egy olyan nyelv és platformfüggetlen interfész, amely lehetővé teszi a programok és szkriptek számára egy dokumentum tartalmának, szerkezetének és stílusának dinamikus elérését és frissítését. A DOM fa egy logikai struktúrát határoz meg. A DOM fára jellemző tulajdonságok:

- A fa legfelső csúcsát gyökérnek hívjuk. XML dokumentumok esetében ez mindig a `dokumentum` csúcs, és nem az XML-beli legfelső elem.
- B csúcsot A csúcs gyermekének nevezzük, ha B a közvetlen utódja A-nak.
- A csúcsot B csúcs szülőjének nevezzük, ha A a közvetlen megelőzője B-nek, azaz ha B az A gyermeke
- Különböző komponensek jelenhetnek meg a dokumentumban, mint például elemek, szöveg, megjegyzés, feldolgozási utasítás stb. A DOM fa minden komponense csúcs, így tehát léteznek elem csúcsok, szöveg csúcsok, megjegyzés csúcsok stb. Alapvetően az elemek attribútumai is csúcsok, ők a gyermekei az elemnek, de többnyire nem számítjuk őket gyermekeknek.

Egy adott dokumentum DOM reprezentációja implementációtól függetlenül ugyanazt a struktúra modellt adja, ugyanazokkal az objektumokkal és kapcsolatokkal. Az XML dokumentumok nagyon jól kezelhetőek DOM-fák segítségével.

Példaként tekintsünk egy egyszerű XML fájlt, és az annak megfelelő DOM-fa reprezentációt. Legyen az XML a következő:

```
<TABLE>
<TBODY>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
```

```
</TR>
<TR>
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</TBODY>
</TABLE>
```

Az ennek megfelelő DOM-fát az 6. ábrán láthatjuk.

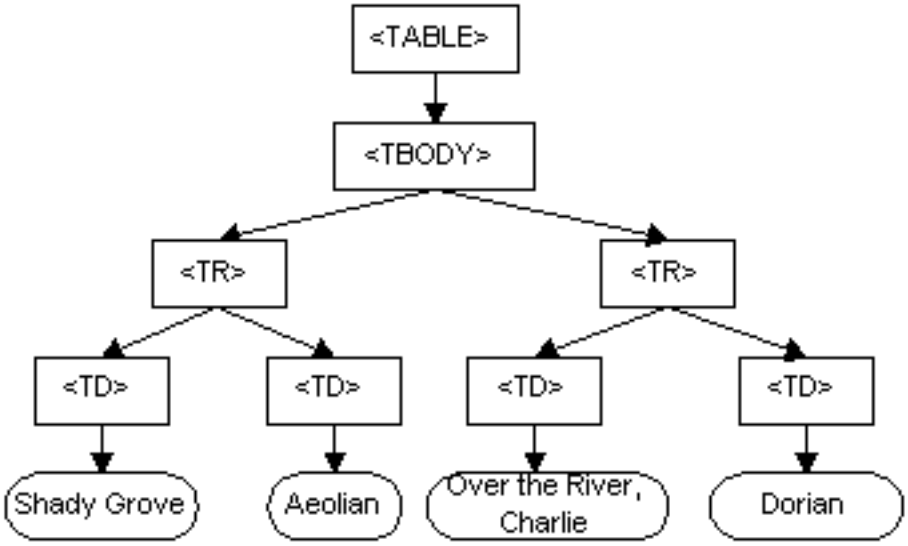
A programozó behelyezi a DOM fát egy adatmodellbe, és a sablonszöveg szerkesztője szépen bejárhatja azt. A DOM csúcsokat az FTL-ben csúcs változóknak feleltetjük meg. Ez egy változótípus, mint ahogyan a sztring, number, hash stb. is. A csúcs változó segítségével éri el a FreeMarker, hogy egy csúcs szülőjére, illetve a gyermekére lehessen hivatkozni. Ez gyakorlatilag ahhoz szükséges, hogy a sablonszöveg szerkesztője navigálni tudjon a csúcsok között. A FreeMarkerben található `freeMarker.ext.dom.NodeModel.parse(File f)` metódus segítségével könnyedén feldolgozhatunk egy XML dokumentumot, ahol a paraméterben a feldolgozandó XML fájlt adjuk meg. Ha a `parse()` metódus alapbeállításait használjuk, akkor mellőzi a megjegyzéseket és végrehajtja az utasítás csúcsokat.

8.1. Programkód generálása

A folyamatosan fejlődő programozási eszközök egyre jobban segítik a rendszerfejlesztők feladatát, segítségükkel egyre egyszerűbben létrehozhatóak a szoftverek. Léteznek eszközök, amiknek segítségével grafikus felületen állíthatjuk össze a rendszer elemeit, és léteznek olyanok is, amelyek a gyakran használatos elemeket kiemelik, általánosítják, így azok újrafelhasználásával időt és energiát spórolhat meg a programozó. Egy összetettebb, bonyolultabb rendszer esetében ezeknek az újrafelhasználható elemeknek a használata nehézkessé válhat, és az adott eszközök alapos ismeretét feltételezi. Erre nyújthat megoldást az, ha egy általános leíró nyelv segítségével tudjuk leírni a megvalósítandó rendszert, illetve annak egy alkotóelemét.

9. Hibernate

A Hibernate egy olyan objektum-relációs leképezés, amely lehetővé teszi adatbázisok és az abban lévő adatok objektumorientált módon való kezelését. A rendszer jelenleg egyaránt támogatja a Java és a .NET keretrendszert. Az eszköz számára megadhatjuk XML fájl segítségével az alkalmazási szakterület objektum sémáját és a relációs sémára történő leképezés módját. Egy ilyen XML fájlból az eszköz legenerálja a sémát, illetve létrehozza az objektumsémáknak



6. ábra

megfelelő osztályhierarchiát. Ezen osztályok segítségével kezelhetjük a az adatbázis egyes elemeit. Az adatbázis és az osztályok közötti kapcsolatot a Hibernate biztosítja, a programozónak csak az általa választott objektumorientált nyelven kell programoznia.

10. Grafikus felhasználói felület generálása

A rendszer felhasználói felületének leprogramozása időigényes, összetett feladat, még akkor is, ha a fejlesztő munkáját olyan eszközök segítik, mint az SWT, vagy a JFace. Ugyanakkor megfigyelhető, hogy adott csoportba tartozó alkalmazások esetén a felhasználói felület egy már bevált koncepciót követ, és ahhoz hasonlóan épülnek fel az elemei. Tehát például egy személyek adatait nyilvántartó program elemei nagy valószínűséggel olyan formok, amik segítségével a személyek adatait rögzíthetjük, illetve a rögzített adatok között kereshetünk, vagyis a felületen gyakran az adatbázis bizonyos elemeit jelenítjük meg, illetve módosítjuk. Ebből látható, hogy egy ilyen alkalmazás felhasználói felületének megalkotása nagyjából analóg módon történhet, mégis alkalmazásonként eltérő lehet, és az adatbázis felépítésétől is nagyban függ. Egy ilyen eszközrendszer a fentebb ismertetett HiberGUI. Lehetőség van eszköz elkészítésére, amely a Hibernate-hez hasonlóan, XML-ből generálja a szükséges kódot, amely jelen esetben egy grafikus felhasználói felületet valósít meg. Ennek az eszköznek a megalkotása volt a feladata a kétszemélyes csapatunknak. Az alkalmazás feladata, hogy a specifikációnak megfelelő XML séma alapján egy HiberGUI illetve SWT és JFace eszközöket alkalmazó grafikus felhasználói felületet állítson elő. Mivel egy ilyen eszköz kifejlesztése viszonylag nagy feladat, ezért ketten dolgoztunk a projekten.

A feladat alapvetően két nagy részre bontható:

- Az XML séma megalkotása, illetőleg egy ilyen sémát megvalósító XML fájl feldolgozása.
- A feldolgozott elemek felhasználása a felhasználói felület létrehozásához.

Ez utóbbi volt az én feladatom a projektben. A probléma megoldásához a FreeMarker eszközrendszerét alkalmaztam. Az elinduláshoz rendelkezésünkre állt egy már elkészített, SWT eszközrendszert alkalmazó szoftver. Ennek a GUI elemeket megvalósító osztályait vizsgálva kezdtük el a rendszerünk fejlesztését. A felületet megvalósító osztályokban előforduló közös részeket keresve találhatóak meg a programok vázát adó főbb utasítások. Ezen szövegrészek nagyban segítették a sablon megalkotását, amibe beillesztve az adott XML fájlból az adatokat, egy HiberGUI eszközöket alkalmazó Java forrásállományt kapunk. Egy általános Java fájlt megvizsgálva is megtaláljuk azokat az alapokat, amik több-

nyire minden Java alkalmazásban előfordulnak. Egy tipikus Java fájl szerkezete a következőképpen írható le:

```
package <csomagnév>;
[import <csomagnév>;]…
public class <osztálynév> [extends osztálynév]
[implements interfésznév [, interfésznév…]]{
 public <osztálynév> ([Osztálynév paraméternév]…){
}
…
}
```

Amint az látható, egy ilyen tipikus Java kódrészletben is sok olyan szövegrész van, amely statikus, mindig megtalálható. Az osztály nevéről például tudjuk, hogy kötelezően a fájl nevével kell hogy megegyezzen, úgyhogy ezek kölcsönösen egymáshoz rendelhetőek. Bizonyos szövegrészek ugyan feltételhez köthetően vannak jelen, de ha jelen vannak, akkor szövegbeli helyük fix, és paramétereik megadási módja is egyértelmű (ha vannak). Egy HiberGUI eszközrendszert felhasználó osztály ennél jóval specifikusabb, több eleme van rögzítetten jelen az osztályban. Például az import résznél biztosan hivatkozni fog az `org.eclipse.swt` csomag néhány osztályára, így egy import `org.eclipse.swt`.*; utasítás minden generált osztályban megjelenik. A sablonszöveg felépítésének magyarázatának egyszerűsítése érdekében nézzünk meg egy osztályt abból a rendszerből, amely alapján a HiberGUI alkalmazást generáló alkalmazás készült:

```
package zemplen.ui;
import java.util.HashSet;
import java.util.Set;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import static hibergui.HiberGUI.*;
import hibergui.Form;
import hibergui.assoc.O2MEdit_Button;
import hibergui.hibernate.HibernateForm;
import hibergui.property.StringEdit;
import zemplen.model.Allomas;
import zemplen.model.Reszido;
import zemplen.model.Verseny;
public class AllomasForm extends HibernateForm<Allomas> {

    VersenyCombo eVerseny;
    StringEdit eElnevezes;
    O2MEdit_Button<Reszido> eReszidok;

    public AllomasForm(Composite cptParent, int mode) {
        super(cptParent, mode);
```

```
        setWindowTitle("Állomások");
    }
    @Override
    protected void save(Allomas allomas) {
        super.save(allomas);
        Verseny v = eVerseny.get();
        Set<Allomas> allomasok = v.getAllomasok();
        if (allomasok == null)
             v.setAllomasok(allomasok = new HashSet<Allomas>());
        allomasok.add(allomas);
    }

    @Override
    protected void delete(Allomas allomas) {
      super.delete(allomas);
        eVerseny.get().getAllomasok().remove(allomas);
    }

    @Override
    protected Composite createFields(Composite cptParent) {
        final Composite cpt = new Composite(cptParent, SWT.BORDER);
        GridLayout gridLayout = new GridLayout();
        gridLayout.numColumns = 2;
        cpt.setLayout(gridLayout);
        GridData data;
        new Label(cpt, SWT.NONE).setText("Verseny:");
        eVerseny = new VersenyCombo(this, cpt, NOT_NULL);
        data = new GridData(GridData.FILL_HORIZONTAL);
        eVerseny.setLayoutData(data);
        new Label(cpt, SWT.NONE).setText("Elnevezés:");
        eElnevezes = new StringEdit(this, cpt, NOT_NULL, "elnevezes",
                                                          "elnevezés");
        data = new GridData(GridData.FILL_HORIZONTAL);
        eElnevezes.setLayoutData(data);
        eReszidok = new O2MEdit_Button<Reszido>(this, cpt, NONE,
                                                "reszidok", "Részidők") {
                    @Override
            protected Form<Reszido> createForm(Composite cpt, int
style) {
                                ReszidoForm f = new ReszidoForm(cpt,
style);
                                f.fixVerseny(eVerseny.get());
                                f.eAllomas.set(AllomasForm.this.get());
                                f.eAllomas.fix(true);
                                f.refresh();
                                return f;
            }
        };

        return cpt;
    }

    @Override
    protected void createColumnsForTable() {
        addColumn(SWT.LEFT, "elnevezes", "Elnevezés", 200);
    }

}
```

Amint az látható, az osztály a HibernateForm generikus osztályt terjeszti ki. Azért, hogy erre az osztályra hivatkozni tudjon, be kell importálni. A példány szintű elemeket, amik többnyire a GUI elemeit adják, a konstruktor előtt szokás definiálni. Ha egy grafikus elemet szeretnénk a felületre, akkor itt fog elhelyezkedni a deklarációja, mégpedig a következő formában:

```
osztálynév változónév;
```

Ezután következik az osztály konstruktora, aminek a paraméterlistát leszámítva szintén kötött formája van:

```
public osztálynév (változódeklarációk){
}
```

Mivel az osztály az SWT eszközrendszeren, illetve a HiberGUI HibernateForm osztályán alapszik, ezért a konstruktor első utasítása az ősosztály konstruktorának meghívása, amely maga után vonja a konstruktor első két paraméterét is. Ezek sorrendben egy Composite, és egy stílust meghatározó int. A konstruktorban elhelyezhetőek a globális beállítások, mint például az ablak elnevezése, amit a `setWindowTitle(String title)` metódussal végezhetünk el. Az osztály következő lényeges része a `createFields()` metódus felülírása. Ennek segítségével helyezhetjük el az elemeket az ablakban. Ez gyakorlatilag nem más, mint egy Composite, amit a szülő `Composite`-ra helyezünk, és elhelyezzük benne a használni kívánt elemeket. Az elrendezéshez egy két oszlopos `GridLayout`-ot használjuk, mert jelen esetben a koncepció az, hogy űrlapot töltünk ki, különböző mezők segítségével. A mezőket egy címke azonosítja, és így alakul ki a két oszlop: címke – szövegbeviteli mező. Ennek az elrendezésnek a megvalósítása minden esetben ugyanaz a négy sor. Módosítás esetén is csak apróbb változtatást igényel, tehát ez is beépíthető a sablonszövegbe. Ezután következnek a címke mező párosok, úgyhogy minden egyes mező elé automatikusan elhelyezhető egy ilyen Label példányosítás. A címkéhez tartozó eszközöket érdemes osztályszintű változóként kezelni, hiszen ezekből több is lehet, és ilyenkor csak egy-egy példányosítást kell végrehajtani. A különbözőféle elemek példányosításkor meghívandó konstruktora már kicsit összetettebb az eddigieknél. Természetesen minden példányosítás hasonlóképpen néz ki:

```
változónév = new típusnév(aktuális paraméterlista);
```

A paraméterlista típusonként eléggé eltérő lehet, de mivel a HiberGUI eszközrendszerében az az analógia, hogy az első paraméter a hozzá tartozó Form, a második tartalmazó Composite, a harmadik paraméter pedig a stílust meghatározó int érték, ezért ezt jelen esetben is feltételezhetjük. A további paraméterek létezéséről, és típusáról általánosságban már nem tudhatunk semmit, ami bizonyos nehézségeket is tárhat elénk. Az erre vonatkozó megoldást az XML séma leírásánál ismerhetjük meg. Az előző elemeket a `GirdData` segítségével helyez-

hetjük el a `Layout-`ban, amely szintén kiemelhető a sablonszövegbe, hiszen az esetleges változásokat megfelelő értékadásokkal majd generáláskor elvégezhetjük. Ezt egy olyan példányosítás követi, melyben egy metódus felüldefiniálás található. Ez a kódrészlet az adatbázisrendszer-beli módosítások miatt bonyolult, legenerálását egyelőre nem teszi lehetővé a HiberGUI. Hasonló a helyzet a `save()` és `delete()` metódusok esetén is. Az ilyen problémák kiküszöbölésére a programozónak lehetősége van elhelyezni az XML-ben olyan kódrészletet, amely módosítás nélkül bekerül a generált fájlba. A metódus befejezéskor visszatérési értékként átadja a létrehozott Composite példányt, amely tartalmazza az elemeket. Ez szintén egy fixen kiemelhető kódrészlet. Végezetül az osztály végén található egy `createColumnsForTable()` metódus, ami egy táblázat létrehozásának leegyszerűsítését célzó HibernateForm eszköz. Egyszerű használatából következően csak az `addColumn()` metódus paraméterei változnak, amit szintén kiolvashatunk majd az XML sémából.

A fenti kódrészlet megírásával az 7. ábrán látható felületet kapjuk:

10.1. Az XML séma

Mindezt még egyszerűbbé tehetjük, ha egy XML fájl segítségével írjuk le a felületet. Egy HiberGUI-t leíró XML-hez tartozik egy megfelelő DTD, amely a fájl helyességéről gondoskodik. Ennek megfelelően az XML séma a lent leírtaknak megfelelően néz ki. Legkülső szinten a `hibergui-mapping` elem áll, amelynek egyetlen attribútuma a `package`, amely megmondja, hogy a generált Java osztály mely csomagba fog tartozni. Az elemnek egyetlen gyermeke van, a `form`. Ez a következő attribútumokkal rendelkezhet:

- `name`: Kötelező megadni, ez lesz a generált osztály neve.
- `class`: Az osztály neve, amelyen a form alapszik. Ezt kapja paraméterként a generikus HibernateForm osztály. Megadása szintén kötelező.
- `title`: Az ablak címének megadása történik vele, kötelező attribútum.
- `extends`: Ha az osztály mégsem a HibernateForm osztályt terjesztené ki, akkor ennek megadásával lehet más osztályt megadni. Ha az attribútum elmarad, automatikusan a ”HibernateForm” értéket kapja.
- `layout`: ennek segítségével testre szabottabb megjelenést biztosíthatunk a form számára. Opcionális mező, alapértelmezetten gridlayout kiosztást kap. Lehetséges értékei: `formlayout`, `grid`, `rowlayout`

A `layout` elem a `griddata`, `formlayoutdata`, illetve a `rowlayoutdata` gyermekeket tartalmazhatja, annak megfelelően, hogy éppen mely elrendezést valósítja meg a form.

`GridData` esetén az oszlopok számát a `colnum`, a vízszintes távolságot a `horizontalspan`, a vízszintes kitöltést pedig a `horizontalalignment` attribútumok segítségével adhatjuk meg. Az első kettő egy számot, míg a harmadik egy SWT stílust vár értékül.

`FormlayoutData` esetén a `top`, `left`, `right`, és `bottom` gyermekek adhatók meg, melyek mindegyike egyaránt a `percent`, `pixel`, `element` attribútumokkal rendelkeznek. Ennek segítségével az elemek egymáshoz képesti elhelyezkedése alapján adható meg a layout. Az attribútumok lehetséges megadási módjai:

- `element`
- `element` és `pixel`
- `element`, `pixel`, és `percent`
- `pixel` és `percent`

Ezek a `FormAttachment`-nek megfelelően fogják beállítani az adott elemet.

`RowlayoutData` esetén a `fill`, `justify`, `pack` és `marginleft` attribútumok adhatóak meg, melyek közül az első három true vagy false értéket vehet fel, míg a harmadik egy pixelszámot meghatározó numerikus érték.

A `form` elemnek számos gyermeke létezhet, ezek a következők:

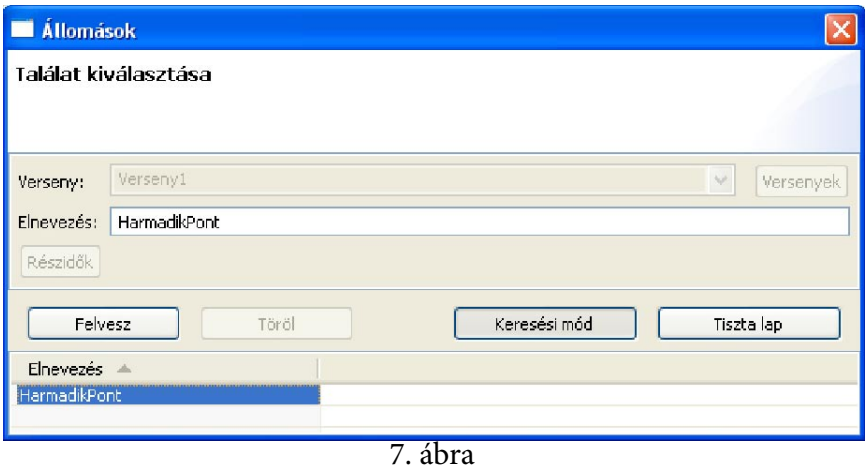
- `import`: Bizonyos osztályok importálása nem automatizálható. Ennek pótlását az import class gyermekeinek felsorolásával tehetjük meg.
- `layout`: Az aktuális layout testre szabását ennek segítségével tehetjük meg.
- `label`: Segítségével címkét helyezhetünk el a formon.
- `edit`: Az interaktív eszközöket, mint pl. szövegbeviteli mező, vagy rádiógomb, ezzel vihetjük fel a felületre.
- `code`: Ebben a mezőben a fejlesztő saját kódot helyezhet el, amely módosítás nélkül megjelenik a célszövegben

– `group`: Az ezen belül elhelyezett elemek egy SWT szerint értelmezett `Group`-ba tartoznak
– `composite`: Az ezen belül elhelyezett elemek egy SWT szerint értelmezett `Composite`-ba kerülnek.
– `constructor`: A fejlesztő a konstruktorban is helyezhet el saját kódot. Az ilyen mezőben írt kód módosítás nélkül megjelenik a konstruktorban.
– `table`: A formhoz tartozó táblázat létrehozásához szükséges elem.

Az `import` elemek `class` gyermekeinek egyetlen attribútuma a `name`, amely az importálandó osztály nevét jelenti. Az attribútum megadása kötelező.

A `label` elemnek van egy `name` attribútuma, ami a változó nevét hivatkozza, illetve egy `text` attribútuma, ami a megjelenítendő szöveget hordozza.

Az `edit` elem attribútumai lehetnek:
– `modifier`: Java módosítók lehetnek, lehetséges értékei: `public`, `static`, `protected`, `private`, `final`.



7. ábra

–
`name`: az elem, mint változó neve
– `type`: az elem, mint változó típusa
– `generic`: ha az elem generikus, akkor itt adhatjuk meg a generikus típusát
– `setargs`: lehetséges értékei: yes, vagy no. Amennyiben a gyermekekként megadott `arg` segítségével szeretnénk az összes argumentumot felüldefiniálni, yes–t kell megadnunk. Ellenkező esetben csak a negyedik argumentumtól kezdődő argumentumokat definiáljuk.

Az `edit` a következő gyermekeket tartalmazhatja:
– `arg`: A paraméterként megadott argumentumot definiálja

– `code`: Az `edit` mezőhöz szorosan kötődő programozói kód itt helyezhető el.

Az `arg` gyermekből tetszőleges számú elhelyezhető, attribútumai a `name` és a `value`. Az első a paraméter nevét, a második a paraméter értékét jelöli.

A `group` és `composite` mezők nagyban hasonlóak, ám a `group`-nak kötelező jelleggel meg kell adni egy címet a `title` attribútum segítségével. Mindkettőnek kötelezően meg kell adni egy nevet, amit a `name` attribútummal tehetünk meg. A `group` és `composite` elemeknek a következő gyermekeik lehetnek: `label`, `edit`, `code`, `layout`, `layoutdata`. Az első három az eddigiekkel azonos módon értendő. A `layout` a group vagy composite elrendezését hivatott beállítani, míg a `layoutdata` a szülő composite elrendezéséhez tartozó adatokat tartalmazza.

A programozó a saját kódját a megfelelő elemeken belül `CDATA` szekciókban helyezheti el.

Az előző fejezetben ismertetett `AllomasForm` osztály legenerálását a következő XML segítségével érhetjük el:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE hibergui-mapping SYSTEM “../../../../HiberGUI/src/hibergui/
dtd/hibergui-mapping-1.0.dtd”>
<hibergui-mapping package=”zemplen.ui”>
      <form name=”AllomasForm” class=”Allomas” extends=”HibernateForm”
title=”Állomás”>
              <import>
                     <class name=”zemplen.model.Reszido”/>
                     <class name=”zemplen.ui.AllomasForm”/>
                     <class name=”zemplen.model.Verseny”/>
                     <class name=”java.util.Set”/>
                     <class name=”java.util.HashSet”/>
                     <class name=”hibergui.Form”/>
              </import>

              <label text=”Verseny:”/>
              <edit type=”VersenyCombo” name=”eVerseny”/>
              <label text=”Elnevezés: “ />
              <edit name=”eElnevezes” type=”StringEdit”>
                     <arg name=”property” value=’”elnevezes”’/>
                     <arg name=”name” value=’”elnevezés”’/>
</edit>
              <edit name=”eReszidok” type=”O2MEdit_Button”
generic=”Reszido”>
                     <arg name=”property” value=’”reszidok”’/>
                     <arg name=”name” value=’”Részidők”’/>
                     <code><![CDATA[
                     {
                     @Override
            protected Form<Reszido> createForm(Composite cpt, int
style) {
                           ReszidoForm f = new ReszidoForm(cpt,
style);
                           f.fixVerseny(eVerseny.get());
                           f.eAllomas.set(AllomasFormGenerated.this.
```

```
get());
                                f.eAllomas.fix(true);
                                f.refresh();
                                return f;
                        }
                };
                        ]]></code>
                </edit>
<code><![CDATA[
                @Override
        protected void save(Allomas allomas) {
                super.save(allomas);
                Verseny v = eVerseny.get();
                Set<Allomas> allomasok = v.getAllomasok();
                if (allomasok == null)
                        v.setAllomasok(allomasok = new HashSet<Allomas>());
                allomasok.add(allomas);
        }

        @Override
        protected void delete(Allomas allomas) {
                super.delete(allomas);
                eVerseny.get().getAllomasok().remove(allomas);
        }]]>
                </code>
                <table>
                        <column label="Elnevezés" property="elnevezes"
                                                                align="LEFT"
width="200"/>
                </table>
        </form>

</hibergui-mapping>
```

Az XML-hez megfelelő DTD egy relatív elérési úttal van megadva. Mivel a form a `zemplen.ui` csomagban helyezkedik el, ezért ennek megfelelően ez lett a package attribútum értéke. A form extends attribútuma elhagyásával továbbra is érvényes maradna az XML, és ugyanazt a formot generálná eredményül az eszköz. Az import hivatkozások azért szükségesek, mert a formon található bizonyos részek nem kiemelhetőek, és az ott használt elemek importálása nem automatizálható. Az `eVerseny` elnevezésű, `VersenyCombo` típusú változót leíró `edit` elemhez nem tartozik `arg` gyermek, mivel paraméterként az alapértelmezett értékeket használja. Az `eReszidok` nevű, `O2MeniEdit_Button` elem generikus osztálya a `generic="Reszido"` attribútummal jelölve van, és az alapértelmezett három paraméteren túl a `property` és `name` paramétereket kapja. Ezek sztring típusú értékékek, ezért Javanak megfelelő sztring literálként kell őket megadni, amit a ' elválasztójelek között tehetünk meg. Mivel a belső osztály nem kiemelhető, ezért annak kódja az `edit` mezőn belüli `code` gyermek segítségével valósul meg. A save és delete metódusok szintén nem kiemelhető elemek, úgyhogy ezek a form `code` gyermekeiként lettek megvalósítva.

Látható, hogy az eredeti Java kód helyett egy egyszerűbb és áttekinthetőbb XML dokumentum szerkesztésével ugyanazt az eredményt előállító kódot kapjuk:

```
/*
* HiberGUI Form
* Generated by HiberGUI generator.
*/
package zemplen.ui;

import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;
import static hibergui.HiberGUI.*;

import zemplen.model.Reszido;
import zemplen.ui.AllomasForm;
import zemplen.model.Verseny;
import java.util.Set;
import java.util.HashSet;
import hibergui.Form;

import hibergui.assoc.O2MEdit_Button;


import hibergui.hibernate.HibernateForm;

import hibergui.property.StringEdit;



import zemplen.model.Allomas;

public class AllomasFormGenerated extends HibernateForm<Allomas>{

VersenyCombo eVerseny;
StringEdit eElnevezes;
O2MEdit_Button<Reszido> eReszidok;
             GridLayout gridLayout;
             GridData data;
             Composite cptFields;

      public AllomasFormGenerated (Composite cptParent, int mode){

                    super(cptParent, mode);
                    setWindowTitle("Állomás");
             }

      @Override
      protected Composite createFields(Composite cptParent) {
                    cptFields = new Composite(cptParent, SWT.BORDER);

             gridLayout = new GridLayout();
             gridLayout.numColumns = 2;
             cptFields.setLayout(gridLayout);
             new Label(cptFields, SWT.NONE).setText("Verseny:");
eVerseny = new VersenyCombo(this, cptFields, NOT_NULL
);
             data = new GridData(GridData.FILL_HORIZONTAL);
             eVerseny.setLayoutData(data);
             new Label(cptFields, SWT.NONE).setText("Elnevezés: ");
eElnevezes = new StringEdit(this, cptFields, NOT_NULL, "elnevezes" ,
"elnevezés"                                  );
             data = new GridData(GridData.FILL_HORIZONTAL);
             eElnevezes.setLayoutData(data);
eReszidok = new O2MEdit_Button<Reszido>(this, cptFields, NOT_NULL
, "reszidok" , "Részidők"                                    )
                    {
                    @Override
```

```
            protected Form<Reszido> createForm(Composite cpt, int
style) {
                                ReszidoForm f = new ReszidoForm(cpt,
style);
                                f.fixVerseny(eVerseny.get());
                                f.eAllomas.set(AllomasFormGenerated.this.
get());
                                f.eAllomas.fix(true);
                                f.refresh();
                                return f;
                        }
                };

                        data = new GridData(GridData.FILL_HORIZONTAL);
                        eReszidok.setLayoutData(data);
                        return cptFields;
        }

        /*
        * Programmer's code here
        */

                        @Override
          protected void save(Allomas allomas) {
                        super.save(allomas);
                        Verseny v = eVerseny.get();
                        Set<Allomas> allomasok = v.getAllomasok();
                        if (allomasok == null)
                                v.setAllomasok(allomasok = new HashSet<Allomas>());
                        allomasok.add(allomas);
          }

          @Override
          protected void delete(Allomas allomas) {
                        super.delete(allomas);
                        eVerseny.get().getAllomasok().remove(allomas);
          }


          @Override
        protected void createColumnsForTable() {
            addColumn(SWT.LEFT, "elnevezes", "Elnevezés", 200);
        }
}
```

Az XML még tovább egyszerűsíthető lenne, ha a HiberGUI-ban található beépített komponensekhez külön XML elemek lennének definiálva, ekkor ugyanis az argumentumok megadása egyszerűbbé válna, vagy esetenként el is lehetne hagyni azokat.

11. A projekt jövője

Amint az előző fejezet végén azt láthattuk, a célszöveg formázása még nem tökéletes, és az GUI elrendezéseit sem minden esetben valósítja meg megfelelően. Az alkalmazás ilyen jellegű hibái javítandóak, és hiányosságai kiegészítendőek. A HiberGUI projekt további előrehaladásával lehetővé válhat a rendszerek grafikus felhasználói felületének és mögöttes adatbázisrendszerének együttes legenerálása, ami nagyban segítheti a fejlesztők munkáját. Lehetséges fejlesztési javaslatok:

- Grafikus felhasználói felület az XML előállításához: Egy grafikus felületen az elemek egérrel való elhelyezése segítségével a fejlesztő létrehozhatja a kívánt felületet, majd egy program ebből legenerálja az XML-t, amiből a kész GUI előállítható.
- Automatikus dokumentáció generálása: A kifejlesztett szoftverrendszerekhez, és ezek felhasználói felületéhez is kötelező dokumentációt mellékelni a megrendelő számára. Ezen dokumentáció bizonyos elemeit generálni lehet, ami segíti a fejlesztők munkáját.
- Webes felületen elhelyezhető alkalmazás generálása: A létrehozott felhasználói felületet böngészővel megjeleníthető formába is át lehet ültetni, ami szintén egy automatizálható folyamat.

12. Köszönetnyilvánítás

Köszönöm segítségét a szakdolgozat létrejöttének támogatásában konzulensemnek, Espák Miklósnak, a projekten velem együtt dolgozó Gyenge Csabának, továbbá Tátrai Gábornak és Debreczeni Andrásnak, akik segítettek a megfogalmazási nehézségekben.

Irodalomjegyzék

1. Matthew Scarpino, Stephen Holder, Stanford Ng, Laurent Mihalkovic: *SWT/Jface in action*
   Manning Publications, 2005
2. Dékány Dániel: *FreeMarker Online Documentation*
   http://freemarker.org/docs/
3. World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*
   http://www.w3.org/TR/2006/REC-xml-20060816/
4. Christian Bauer, Gavin King: Java Persistence with Hibernate:
   Manning Publications, 2007
5. The Eclipse Foundation: SWT Documentation:
   http://www.eclipse.org/swt