Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Modell alapú tesztgenerálás iOS platformra

Szakdolgozat

Készítette
Szabó Csaba

Konzulensek
Dr. Ráth István, Ujhelyi Zoltán

2013. december 20.

SZAKDOLGOZAT-FELADAT

Szabó Csaba (BABFRK)
szigorló informatikus hallgató részére

Modell alapú tesztgenerálás iOS platformra

Napjainkban a szakterület-specifikus modellezési nyelvek fontos és hatékony segítséget nyújtanak számos szoftverfejlesztési területen, pl. beágyazott és mobil alkalmazások készítése során. Ezekkel a nyelvekkel lehetőség nyílik arra, hogy az alkalmazás számos aspektusát (pl. felhasználói űrlapok, üzleti logika bizonyos részei, és az ehhez kapcsolódó tesztesetek) magas szintű mérnöki modellek segítségével tervezzük meg, melyből a végső kód automatikus kódgenerálás során áll elő. Ilyen eszközök támogathatják az alkalmazások tesztelésének részleges automatizálását is, mivel nemcsak maga az alkalmazás, hanem az azt tesztelő kód is előállítható.

A feladat végrehajtása során a hallgató feladata, hogy kifejlesszen egy Eclipse Modeling Framework technológiákra épülő szakterületi modellező rendszert, melyek segítségével iOS platformon futó alkalmazások szcenárió alapú tesztelése modellezhető. Az eszközhöz tartozó kódgenerátor a modellezőnyelven leírt modellek alapján automatikus kódszintézissel állítsa elő a futtatható tesztkódot, mely az iOS platformon elterjedt KIF keretrendszer segítségével támogatja az alkalmazás szcenárió alapú (integrációs) tesztelését. Fontos, hogy a nyelvekre épülő fejlesztőeszközök kimenete illeszkedjen az Apple Xcode fejlesztőkörnyezethez.

A szakdolgozat kidolgozása a következő részfeladatok megoldását igényli:

- Végezzen irodalomkutatást, és mutassa be az iOS platformhoz jelenleg elérhető modellvezérelt fejlesztőeszközöket, illetve tesztelésre használható eszközöket.
- Tervezze meg a iOS alkalmazások szcenárió alapú tesztelését lehetővé tévő modellezőrendszert, és valósítsa meg az integrált fejlesztőeszközt.
- Tervezze és valósítsa meg a kódgenerátort, mely legyen képes a szcenárió alapú tesztesetek forráskódjának automatikus előállítására.
- Egy részletesen kidolgozott példaalkalmazáson keresztül mutassa be a modellezőrendszer működését, használatát és a fejlesztés menetét. Értékelje munkáját és vázolja fel a továbbfejlesztési lehetőségeket.

Tanszéki konzulens: Dr. Ráth István, tudományos munkatárs

Budapest, 2013. október 9.

.....
Dr. Jobbágy Ákos
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Szabó Csaba*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2013. december 20.

Szabó Csaba
hallgató

Kivonat

Napjainkban a szakterület-specifikus modellezési nyelvek fontos és hatékony segítséget nyújtanak számos szoftverfejlesztési területen, pl. beágyazott és mobil alkalmazások készítése során. Ezekkel a nyelvekkel lehetőség nyílik arra, hogy az alkalmazás számos aspektusát (pl. felhasználói űrlapok, üzleti logika bizonyos részei, és az ehhez kapcsolódó tesztesetek) magas szintű mérnöki modellek segítségével tervezzük meg, melyből a végső kód automatikus kódgenerálás során áll elő. Ilyen eszközök támogathatják az alkalmazások tesztelésének részleges automatizálását is, mivel nemcsak maga az alkalmazás, hanem az azt tesztelő kód is előállítható.

A feladat végrehajtása során az volt a feladatom, hogy kifejlesszek egy Eclipse Modeling Framework technológiákra épülő szakterületi modellező rendszert, melynek segítségével iOS platformon futó alkalmazások szcenárió alapú tesztelése modellezhető. Az eszközhöz tartozó kódgenerátor a modellezőnyelven leírt modellek alapján automatikus kódszintézissel állítsa elő a futtatható tesztkódot, mely az iOS platformon elterjedt KIF keretrendszer segítségével támogatja az alkalmazás szcenárió alapú (integrációs) tesztelését. Fontos, hogy a nyelvekre épülő fejlesztőeszközök kimenete illeszkedjen az Apple Xcode fejlesztőkörnyezethez.

Abstract

Nowadays, domain-specific modeling languages provide effective assistance in a number of important and complex software development area, for example in embedded and mobile application development. With these domain-specific languages, we have the opportunity to design high-level models for many application aspects (e.g. user forms, some parts of the business logic or the test cases), and we can automatically generate code from these models. Such tools can support a partial test automation of the applications, not only the application itself, but also the tester code can be obtained.

My task was to develop an Eclipse Modeling Framework technology based modeling system, that allows you to model scenario-based iOS platform application tests. This tool contains a code generator, which generate automatically the runnable test source code with code synthesis. The generated test code supports application scenario-based integration test on iOS platform with the popular KIF test framework. It's important to fit the tool's output to Apple's Xcode development environment.

1. fejezet

Bevezető

Napjainkban a mobil eszközök nagymértékű elterjedése figyelhető meg. Az okostelefonokat már nem csak telefonálásra használják, hanem e-mailezésre, böngészésre, játékra és sok egyéb tevékenységre. A modern mobil platformok között az iOS a második legelterjedtebb [1] mobil operációs rendszer az Android után, viszont fejlesztői oldalról vonzóbb lehet, hiszen az iOS felhasználók sokkal hajlandóbbak alkalmazásokat vásárolni, így manapság megéri iOS fejlesztőként is dolgozni, annak ellenére hogy sokkal több Androidos készülék van a piacon.

A mobil eszközökkel párhuzamosan a mobil alkalmazások száma is robbanásszerűen növekszik, emellett a mobil eszközök teljesítménye is évről évre nő. Ezeknek köszönhető, hogy manapság egy mobil szoftver bonyolultsága összevethető webes és asztali társaival. A komplexitás és projekt méret eredményezi azt, hogy a mobil alkalmazások tesztelése is egyre nagyobb szerepet kap. A manuális teszt mellett nagyon gyakran alkalmaznak automatizált unit teszteket és felhasználói felület teszteket is.

Felhasználói felület tesztek és ezeket támogató keretrendszerek már léteznek iOS platformra. A legfőbb ilyen szoftverek az alábbiak:

- Telerik Test Studio [2]: Saját leírónyelvvel rendelkezik. A teszteket manuálisan lehet rögzíteni és újra visszajátszani.
- Apple UI Automation [3]: JavaScript nyelven lehet megfogalmazni a teszteket, de az egyedi felhasználói felületek kezelése bonyolult.
- GorillaLogic MonkeyTalk [4]: MonkeyTalk és JavaScript nyelveken lehet megfogalmazni a teszteket, mely jól használható UI tesztek leírására és végrehajtására.
- KIF [5]: Objective-C nyelven lehet megfogalmazni a teszteket. A teljes felhasználói felület elérhető programkódból az AccessibilityLabel paraméteren keresztül. Nem szükséges más szoftver a használhatához, Xcode[1]-ba integrálódik.
- Calabash [6]: Gherkin [7] szakterület-specifikus nyelven, szövegesen lehet megfogalmazni a lépéseket, de az elvárt eredmény ellenőrzés nem egyszerű.

[1] iOS fejlesztőkörnyezet

A fenti szoftverek többsége jól használható UI tesztek írására. Számomra a KIF és a MonkeyTalk volt a leghatékonyabb eszköz, de a KIF az Xcode integrációja és a könnyű használata miatt bizonyult számomra a legjobb UI teszt eszköznek. A felsorolt szoftverekkel az a probléma, hogy programkódból kell írni a teszteket, ezáltal körülményes a felhasználói felületek változását programkódban kezelni és folyamatosan módosítani.

A tesztek leírására egy grafikus felület kényelmesebb lenne, amivel a teljes teszt folyamat könnyen leírható és szerkeszthető. A tesztek létrehozásának és utólagos változtatásának nehézségét próbálom megoldani a grafikus teszt szerkesztővel, mellyel csak egy grafikus teszt modellt kell létrehozni és abból már kódgenerátor segítségével a programkód automatikusan generálódik le.

A szakdolgozatomban először bemutatom és röviden jellemzem az általam felhasznált technológiákat és eszközöket (2. fejezet), majd a szoftver elkészítésének teljes tervezési és megvalósítási folyamatát bemutatom (3. fejezet). A 4. fejezetben egy általam készített iOS alkalmazáson keresztül bemutatom a szoftver használatát és funkcióit. A szakdolgozat végén röviden kifejtem a továbbfejlesztési lehetőségeket (5. fejezet), majd összességében értékelem és jellemzem az elkészült szoftvert (6. fejezet).

A szoftvert az Eclipse helyett más eszközben is elkészíthettem volna, vagy akár egy Xcode plug-inként is, de én azért választottam az Eclipse platformot a szoftver elkészítésére, mert az én különböző problémáimra és igényeimre egyszerre tud megoldást nyújtani, mint a modellezés, kódgenerálás, grafikus szerkesztő készítése és ezek közötti kommunikáció. Ezeken kívül egy Eclipse plug-in könnyen továbbfejleszthető és más projektekhez könnyen hozzá lehet kapcsolni. Egy saját grafikus szerkesztő és egy saját kódgenerátor készítése feleslegesen sok munka lenne, főleg mivel már léteznek megoldások ezekre a problémákra, melyek használata, testreszabása és együttműködése Eclipse-ben megvalósítható.

Az elkészített teszt szerkesztő az iOS platformtól függetlenül más mobil, asztali vagy webes alkalmazásoknál is használható kis átalakítással. A megoldásomban csak az Objective-C kódgenerátor és az Xcode projekttel való kapcsolat iOS platform specifikus.

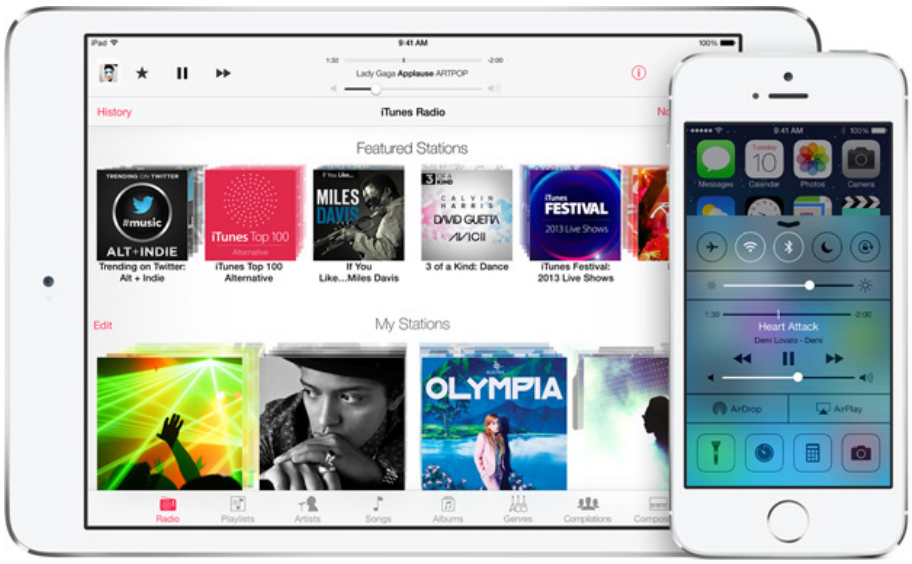
2. fejezet

Felhasznált technológiák

A fejezetben bemutatom a szakdolgozat alatt megismert és alkalmazott főbb technológiákat, eszközöket és keretrendszereket. Az egyes technológiák főbb részeit bemutatom, az általam felhasznált részekre fektetve a hangsúly. A technológiákról külön-külön írok, a következő fejezetben pedig ezek részletes kapcsolatára és együttműködésére térek ki.

2.1. Az iOS platform

Az iOS egy mobil operációs rendszer, mely 2007-ben jelent meg az iPhone bemutatásával együtt. Érintőkijelzős készülékekre lett tervezve, több ujjas gesztusokkal vezérelve. A felhasználói felület kialakítása során ezt tartották elsődleges szempontnak, ezért nézetek, listák, csúszkák és gombok találhatóak főképp a felhasználói felületen. Az iOS sikerének egyik oka a kiemelkedő felhasználói élmény, melynek alapjai a válaszidő és a reszponzivitás.



2.1. ábra. *Az iOS legújabb verziója az iOS7, mely főleg a felhasználói felületében változott meg. [8]*

2.2. Az iOS alkalmazások tesztelése

Az iPhone számítási és grafikus teljesítménye 2007 óta szinte minden évben duplájára nő [9], az alkalmazások száma folyamatosan nő az App Store-ban, 2013 januárjában ez a szám átlépte a 800000-es alkalmazásszámot [10]. A teljesítmény-, alkalmazászám-növekedés és az alkalmazások komplexitásának növekedése az alkalmazások tesztelési igényét is megnövelte. Létezik már fájlszinkronizáló, kép- és videószerkesztő alkalmazás is, melyek tudása és funkcionalitása vetekszik a desktop alkalmazásokéval, ezek hibamentes működéséhez szükséges az alapos tesztelés.

2.2.1 A manuális tesztelés

Minden szoftverfejlesztési projektben szükség van tesztelésre, egy mobil alkalmazás esetében is, erre több módszer is létezik. A legegyszerűbb és mindenki által használt mód a manuális tesztelés, amikor a fejlesztő vagy egy dedikált tesztelő kézzel végignézi az alkalmazást, kipróbálja a funkcióit és figyel a felhasználói felület szépségére. Ez a módszer hatékony, hiszen sok hiba van egy mobilalkalmazás esetében amit szinte csak manuális tesztelés során lehet észrevenni (például animációs szépsége, alkalmazás gördülékenysége és gyorsasága). A manuális tesztelés hatékony, de sok folyamat automatizálható a manuális tesztelésből, ezért is találták ki az automatizált teszteket.

2.2.2 Az automatizált tesztelés

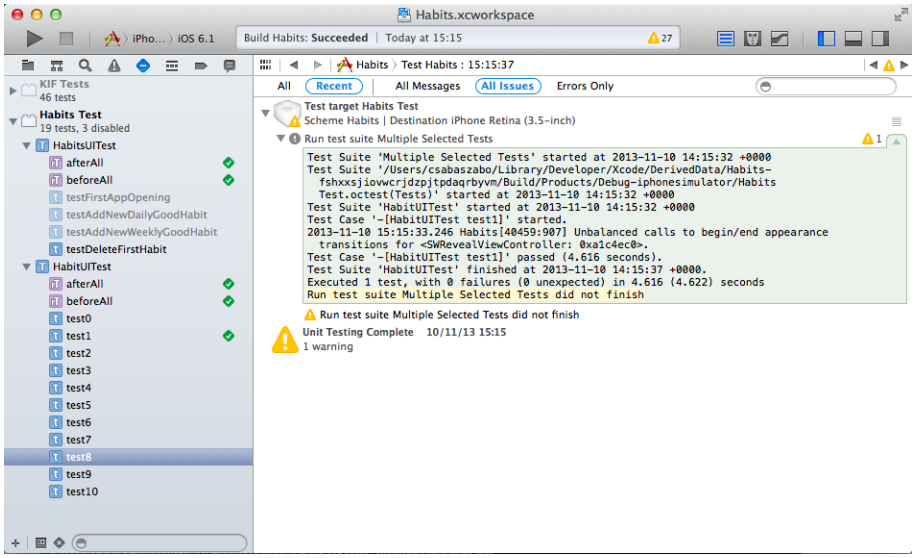
Az automatizált teszteknek több fajtája is van, ezekből én két típust mutatok be: unit tesztek és felhasználói felület tesztek. A unit tesztek - hasonlóan a webes és asztali platformokhoz - az alkalmazás egyes kis részeit tesztelik. Unit teszt írásához elég a fejlesztés alatt álló modul forráskódjának megléte, hiszen lehetőség szerint csak egy modult tesztel a unit teszt. A felhasználói felület tesztet tipikusan már akkor szokták használni, amikor az alkalmazás üzleti logika része kész és a felhasználói felület is részben vagy teljesen. A felhasználói felület teszt lényegében a manuális teszteléshez hasonlóan az alkalmazást „nyomkodja”, vagy a felhasználói gesztusokat szimulálja programozottan. A szakdolgozatomban felhasználói felület teszteket generálok, tehát ezután ha iOS automatizált tesztet említek a dolgozatban, akkor az az automatizált felhasználói felület tesztet fogja jelenteni.

A két legnépszerűbb automatizált felhasználói teszt program iOS platformra az Apple által készített UI Automation és a KIF nyílt forráskódú keretrendszer. A UI Automation [11] szoftver az Instruments[1] segédprogram része, mellyel felhasználói felület teszteket lehet vele végrehajtani javascript nyelven. Azért kerültem el ezt a módszert, mert saját készítésű, nem az iOS által biztosított beépített felhasználói felületeket helytelenül kezeli, képtelen minden felhasználói eseményt szimulálni. A KIF keretrendszert használom a szakdolgozatom során, egy későbbi alfejezetben részletesen be is mutatom.

[1] Az alkalmazás teljesítményének mérésére, kódanalízisre és tesztelésre használatos segédprogram.

2.3. Az Xcode fejlesztőkörnyezet

iOS platformra natívan Objective-C nyelven lehet fejleszteni, a natív fejlesztőkörnyezet az Apple által készített Xcode. Az Xcode tökéletes iOS alkalmazások fejlesztésére, mert magában foglal egy jól használható kódszerkesztőt, felhasználói felület szerkesztőt és sok egyéb hasznos modult mint a tesztek futtatására és azok eredményeinek jelzésére használt komponenst. Az alábbi ábrán egy sikeresen lefuttatott teszt látható Xcode-ban (2.2. ábra).



2.2. ábra. *Sikeresen futtatott „test1" teszteset az Xcode-ban.*

2.4. A KIF keretrendszer

A KIF nyílt forráskódú keretrendszer egyszerű automatizált tesztelést biztosít iOS alkalmazásokhoz, az *accessibilityLabel*[2] változójukon keresztül azonosítva a felhasználói felület elemeit. A KIF nem dokumentált Apple API-kat használ, de mivel az alkalmazás kódjába nem kerül ilyen bele, ezért az alkalmazásboltba (Apple AppStore) fel lehet tölteni, az Apple nem fogja visszautasítani nem hivatalos dokumentáció használata miatt. A teszteket Objective-C nyelven kell írni Xcode-ban a projekten belül, így nem kell külön nyelvet és eszközt használni. Futtathatóak az Xcode szimulátorában, illetve fizikai eszközön is.

A KIF-nek van néhány hiányossága, mint például az összes gesztus kezelésének hiánya, illetve a hiányos felhasználói felület ellenőrzés. A felhasználói felületek ellenőrzésére jelenleg csak olyan metódus áll rendelkezésre, mely megmondja hogy adott *accessibilityLabel*-lel rendelkező elem a képernyőn van-e, de hogy az pontosan hogy helyezkedik el azt nem lehet a KIF API-ját keresztül ellenőrizni, egyéb kódsorokkal viszont igen.

A KIF 2.0-át 2013 szeptemberében adták ki [12], melyeben a KIF teszteket ocunit[3] technológia felett valósították meg, mely teljes integrációt jelent az Xcode-dal, ezáltal a KIF tesztek futtatása sokkal felhasználóbarátabb.

[2]Minden felhasználói elemre megadható érték, melyet a látássérültek vagy vakok tudnak használni VoiceOver felolvasóprogrammal. [13]

[3]Az egyik első teszt keretrendszer Objective-C nyelven, melyet az Xcode is támogat.

2.4.1 A KIF keretrendszer funkciói

A tesztek általában két fő lépés fajtából állnak: teszt lépés és elvárt eredmény ellenőrzés. A két lépés nincs különválasztva a KIF-ben, viszont mindkettőre van lehetőség. Az alábbi listában összefoglaltam a legfőbb KIF tesztlépéseket:

- Adott *accessibilityLabellel* rendelkező UI elem megnyomása egyszer,
- adott *accessibilityLabellel* rendelkező UI elem megnyomása hosszan,
- képernyő meghatározott pontjának megnyomása,
- adott *accessibilityLabelel* rendelkező UI elemen szöveg bevitele,
- adott *accessibilityLabellel* rendelkező UI elemen szöveg törlése,
- adott *accessibilityLabellel* rendelkező switch vagy slider értékének beállítása,
- felugró ablak bezárása,
- adott fénykép kiválasztása a fényképek közül,
- adott *accessibilityLabellel* rendelkező listából meghatározott elem kiválasztása,
- adott *accessibilityLabellel* rendelkező UI elemen elemelés gesztus végrehajtása,
- adott *accessibilityLabellel* rendelkező UI elemen görgetés gesztus végrehajtása.

Elvárt eredmény ellenőrzésre a lehetőségek tárháza már nem olyan széles, de ezeket kiegészítve teljes értékű ellenőrzést tudunk végezni. Az alábbi listában összefoglalom a legfőbb KIF elvárt eredmény ellenőrzéseket:

- Adott *accessibilityLabellel* rendelkező nézetre várakozás,
- adott *accessibilityLabellel* rendelkező nézet meg nem jelenése,
- adott *accessibilityLabellel* rendelkező és megnyomható nézetre várakozás.

A fenti ellenőrzéseknél kötelező megadni a nézet *accessibilityLabelét*, és opcionálisan további paraméterekkel is pontosítható az ellenőrzés.

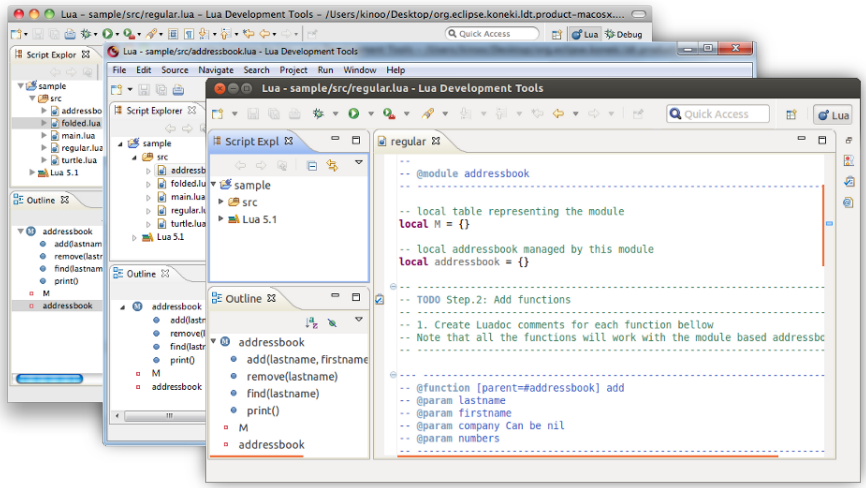
2.4.2 A KIF keretrendszer használata

A KIF keretrendszer használata az új Xcode séma létrehozása után már egyszerű, hiszen az ocunit/xcunit teszteket megvalósító KIF teszteket is könnyű az Xcode-on belül indítani, azok eredményéről értesülni. Emellett könnyen integrálható más teszt keretrendszerekbe, mint például a Specta[4] keretrendszerbe.

2.5. Az Eclipse fejlesztőkörnyezet

Az Eclipse nyílt forráskódú, platformfüggetlen szoftverkeretrendszer, melyet jellemzően Java intergrált fejlesztőkönyezetként szoktak használni, de más kliensalkalmazások fejlesztésére is jó, illetve magához az Eclipe keretrendszerhez is lehet modult (plug-int) fejleszteni.

[4]TDD keretrendszer iOS tesztekhez [14]



2.3. ábra. *Az Eclipse elérhető Linux, Windows és OS X operációs rendszerekre is [15].*

2.6. Az Eclipse plug-in fejlesztés

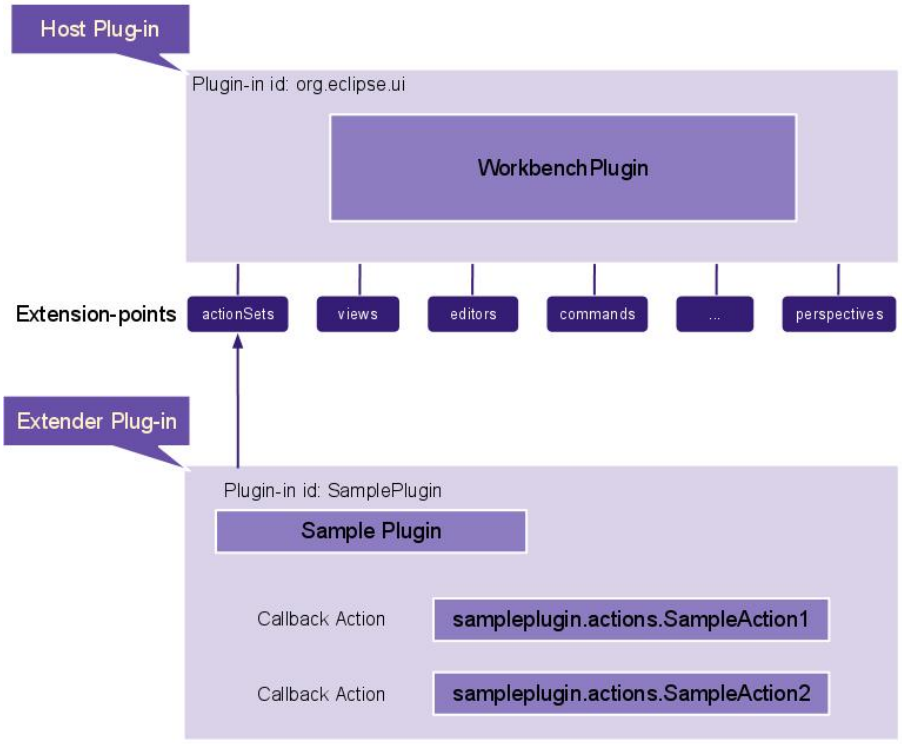
Amikor elindítjuk az Eclipse-et, akkor amíg elindul láthatjuk hogy milyen sok plug-int tölt be az elinduláshoz, hiszen az Eclipse sok-sok plug-inből áll össze, egy moduláris szoftver. Mi is tudunk írni plug-int, melyet az Eclipse-ben lehet használni. A plug-inek között tudunk létrehozni kapcsolatokat *extension point*-okon keresztül. Az általunk fejlesztett plug-in fel tudja használni a többi plug-in *extension point*-jait, így tudunk például az Eclipse grafikus felületéhez új ablakokat hozzáadni vagy épp új projekt típust és ahhoz varázslót létrehozni. Az alábbi ábra (2.4. ábra) példát mutat arra, hogy hogy lehet kiegészíteni az Eclipse felhasználói felületét (Plug-in id: *org.eclipse.ui*) extension pointon keresztül egy általunk készített plug-innel (Plug-in id: *SamplePlugin*). A *SamplePlugin* az *org.eclipse.ui* egy extension pointját valósítja meg, így jön létre a kapcsolat. Ha azt szeretnénk, hogy más plug-in a mi plug-inünket használni tudja, akkor pedig *extension point*-ot kell definiálnunk.

Egy plug-in projekt tipikusan deklaráció, implementáció és erőforrás részekből áll. A deklaráció részbe tartozik a *manifest.mf* és a *plugin.xml* fájl, melyekben a plug-in beállításai találhatóak és beállíthatóak. Az implementáció rész Java fájlokból áll, az erőforrás részbe pedig minden más tartozik, például képek vagy szöveges fájlok.

Mivel az általunk fejlesztett plug-in egy modul, mely beépül az Eclipse-be, ezért nem célszerű a fejlesztésre felhasznált Eclipse-be integrálni a fejlesztett plug-int, hanem egy úgynevezett *Run-Time Eclipse* verziót kell indítani, mely már tartalmazza a plug-inünket és abban kipróbálhatjuk, így nem okoz gondot ha valamit elront a mi munkánk. Ezt a *Run-Time Eclipse* verziót az Eclipse önmagától elindítja, ha egy plug-in projektet futtatunk.

2.7. Az Eclipse BPMN2 szerkesztő

A BPMN2 modell szerkesztő a SOA platform Eclipse projekt egy része. A SOA projekt célja egy keretrendszer elkészítése, mellyel tervezni, konfigurálni, összerakni, fejleszteni, monitorozni és menedzselni lehet a szoftver projekteket SOA alapokon. A BPMN2 egy ilyen



2.4. ábra. *Eclipse modulok kapcsolata [16].*

SOA grafikus szerkesztő, mellyel folyamatokat lehet leírni és végrehajtani BPEL szerint. A BPMN2 egy olyan Eclipse plug-in projekt, melyet már további kiegészíthetőségre és egyedi átalakításra is felkészítettek, ezért választottam én is ezt az iOS tesztek leírásához.

Az eszköz dokumentáltsága viszont sajnos nagyon szegény, néhány példaprogramon kívül alig találtam hozzá dokumentációt, ezáltal a BPMN2 szerkesztő használata és kiegészítése nem túl egyszerű feladat, és nagy programozói közösség sem található a szoftver mögött, tehát hibák és elakadás esetén nem bízhatunk abban, hogy biztos találunk rá választ az interneten keresgélve.

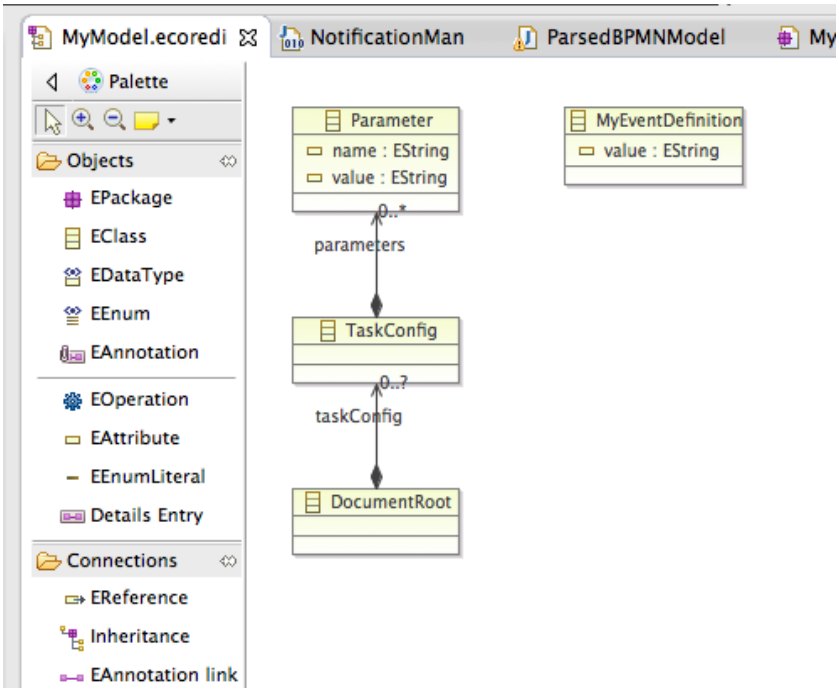
2.8. Az Eclipse Modeling Framework (EMF)

Az EMF egy modellezési keretrendszer, mely egy hasznos eszköz alkalmazások fejlesztéséhez. Az EMF tulajdonképpen az Eclipse metamodellező eszköze, mely szakterület-specifikus nyelvek létrehozása mellett képes a metamodell elkészítésére és szerkesztésére és abból forráskód legenerálására is.

Az EMF modellezése a metamodelleken alapszik, metamodelleket lehet készíteni vele. A metamodellek saját leírónyelve az Ecore. A modellek szerkesztése grafikus módon és Tree Editorral[5] is végrehajtható. Érdemes a modell fő vázát a grafikus felületen (2.5. ábra) elkészíteni, majd a Tree Editorban módosítani az apróbb részleteket. A grafikus modellt nevezik *ecore diagram*nak, míg az utóbbit szimplán *ecore*-nak. Átjárás van a két modell között, a grafikus modell mentésekor automatikusan legenerálódik az ecore fájl, de ez visszafelé nem igaz. Létezik megoldás arra, hogy az ecore fájlból diagramot generáljunk le, de ez komple-

[5]Fa struktúrájú szerkesztő, mellyel könnyen áttekinthető az egész modell.

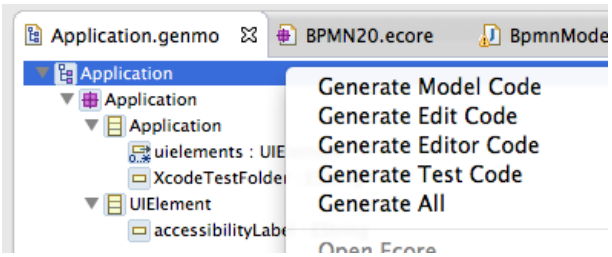
xebb modellek esetén nem teljesen megbízható. Az ecore modellünk szintaktikai helyességét az EMF beépített validációjával könnyen tudjuk ellenőrizni, mely gyorsmenüből könnyen indítható.



2.5. ábra. *Az Ecore diagram szerkesztője.*

Kódgeneráláshoz az ecore modellhez fel kell venni egy generátor modellt, melyet az EMF biztosít. A generátor modell segítségével pár kattintással az alábbiakat tudjuk legenerálni (2.6. ábra):

- **Model**: Az elkészített metamodellhez tartozó össze modell legenerálása, beleértve a factory osztályokat is. Külön generálja le az interfész és implementációs fájlokat.
- **Edit**: A modell manipulálhatóságát biztosítja.
- **Editor**: Az általunk készített metamodellt megvalósító modellhez készít egy szerkesztőt, mellyel a felhasználói felületen könnyen tudjuk szerkeszteni a modell tartalmát.
- **Test**: A modellhez tartozó tesztek.



2.6. ábra. *Az ecore generátorban az egyes részek egy kattintással legenerálhatóak.*

A BPMN2 modell egy EMF modell, melyet a BPMN2 grafikus szerkesztője megjelenít. Mivel a BPMN2 modell egy EMF modell, ezért EMF modellként lehet kezelni, az EMF programozói felületén keresztül könnyen lehet olvasni és akár módosítani is.

2.9. Az Xtend kódgenerátor

Az Xtend célja egy olyan Javahoz hasonló nyelv teremtése, melyben könnyebben tudjuk leírni ugyanazt a kódrészletet mint Javaban, illetve több nyelvi elemet is tartalmaz, például lambda kifejezés vagy operátor felüldefiniálás. Az Xtend konkrét Java kódra fordul minden mentés után, mely szépen formázott és könnyen olvasható. Teljesítménye is megegyezik a tisztán Javaban írt kóddal, hiszen nem az Xtend kód fut, hanem az Xtendből generált tisztán Java kód. Az Xtend célja a produktivitás és kód olvashatóság javítása. Népszerű eszköz android és web fejlesztők körében is.

Az Xtend tartalmaz egy sablon alapú kódgenerátort is, mellyel Java vagy bármilyen más nyelvre lehet kódot generálni template kifejezések segítségével, tehát Objective-C nyelvre is lehet így kódgenerátort írni, ezt használom én is a szakdolgozatomban. Egy template kifejezés lényegében egy string összefűzés. Erre egy példát mutat az alábbi kódrészlet, mely egy KIF teszteset fejléce:

2.1. Forráskód. *Sablon alapú Xtend kódgenerátor példa.*

```
        def testFileBegin(String fileName, String projectName)'''
//  <<fileName>>.m

#import <KIF/KIF.h>

@interface <<fileName>> : KIFTestCase
@end

@implementation <<fileName>>
'''
```

A beillesztett Xtend kódrészletből (2.1 forráskód) Java kód generálódik (2.2 forráskód). Látszik, hogy a template kifejezés egy Java *StringConcatenation* objektumot használ, mely a template kifejezésben lévő szövegrészeket vágja össze:

2.2. Forráskód. *A fenti sablon alapú Xtend kódgenerátorból (2.1 forráskód) legenerált Java kódrészlet.*

```
public CharSequence testFileBegin(final String fileName, final String projectName) {
    StringConcatenation _builder = new StringConcatenation();
    _builder.append("//  ");
    _builder.append(fileName, "");
    _builder.append(".m");
    _builder.newLineIfNotEmpty();
    _builder.newLine();
    _builder.append("#import <KIF/KIF.h>");
    _builder.newLine();
    _builder.newLine();
    _builder.append("@interface ");
    _builder.append(fileName, "");
    _builder.append(" : KIFTestCase");
    _builder.newLineIfNotEmpty();
    _builder.append("@end");
    _builder.newLine();
    _builder.newLine();
    _builder.append("@implementation ");
    _builder.append(fileName, "");
    _builder.newLineIfNotEmpty();
    return _builder;
  }
```

3. fejezet

Tervezési és megvalósítási lépések

A szakdolgozatom célja egy olyan szoftver készítése, mellyel grafikus módon lehet iOS felhasználói felület teszteket készíteni, azaz a grafikus modellből megkapjuk a legenerált tesztek kódját. Ebben a fejezetben részletesen bemutatom a szoftver keretrendszer tervezését, felépítését és működését.

3.1. A szoftver tervezése

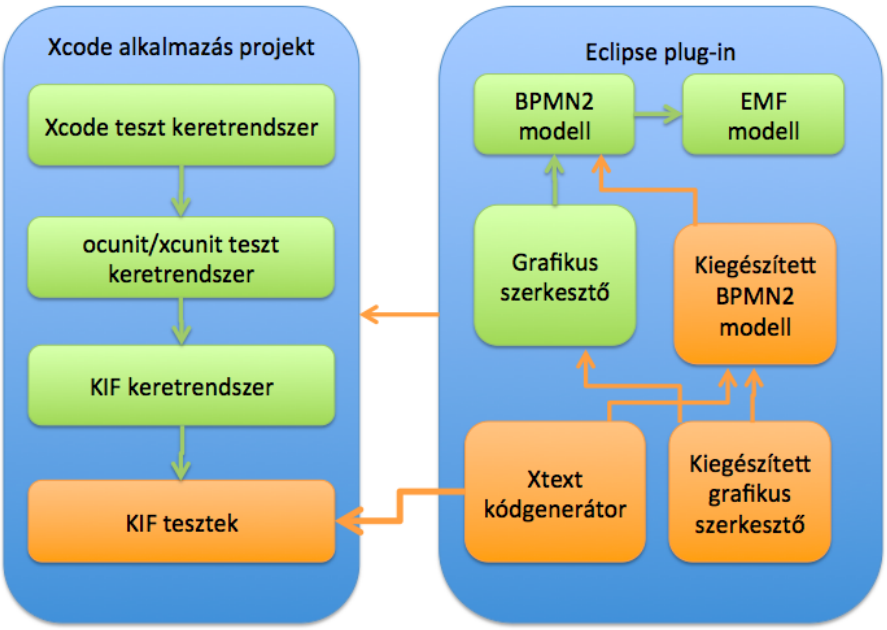
Az Xcode-ban a KIF keretrendszerrel Objective-C nyelven lehet felhasználói felület teszteket írni, de ha ezt grafikus felületen szeretnénk összeállítani, ahhoz az Xcode már nem használható, mert nem bővíthető további modulokkal[1]. Más eszközzel kell a teszt szerkesztőt elkészíteni, és ennek együtt kell működnie az Xcode-dal. Az Xcode programot nem szeretném helyettesíteni más programmal, mert a fejlesztők nagy százaléka ezt a programot használja iOS alkalmazások fejlesztésére. Az Eclipse egy olyan moduláris szoftverkeretrendszer, mely egy ilyen grafikus szerkesztő készítésére is alkalmas.

Több eszköz együttműködését kell megoldanom, mely pontosan két programot és ezen belül több keretrendszert jelent. A két fő eszköz az Xcode és az Eclipse, melyek között egyirányú kapcsolat áll fenn: az Eclipse olvassa és írja az Xcode projektben lévő fájlokat. Az Xcode-on belül a KIF keretrendszert használom felhasználói felület tesztekre, az Eclipse-en belül pedig a BPMN2 szerkesztőt a tesztek leírására. A 3.1. ábra bemutatja a keretrendszerem arcitektúrális ábráját, melyen a narancssárgával jelzett elemeket és kapcsolatokat készítettem el.

A szoftver tervezésekor az alábbi szempontokat tartottam szem előtt:

- A fejlesztőknek vagy tesztelőknek ne okozzon sok plusz munkát a teszt keretrendszer beüzemelése,
- a grafikus szerkesztőfelületen kívül legyen más hasznos funkciója is,
- könnyen továbbfejleszthető legyen más teszt keretrendszerek és platformok irányába,
- ne legyen annyira bonyolult, hogy a szakdolgozat során ne tudjam befejezni.

[1] Hivatalosan nem bővíthető tovább, de létezik már rá megoldás [19].



3.1. ábra. *A teszt generáló keretrendszer felületes architektúra ábrája.*

A fejezetben ennek a rendszernek a felépítését mutatom be részletesen, a következő fejezetben pedig egy példán keresztül mutatom be használatát és funkcióit.

3.2. Az Xcode projekt

Egy iOS alkalmazáshoz tartozik egy (vagy több) Xcode projekt, mely tartalmazza az alkalmazás tesztjeit is. Ezekbe a teszt mappákba kell nekem a teszt kódot generálni. Ahogy azt az első fejezetben bemutattam, az Xcode-ban létezik beépített teszt keretrendszer, ehhez csatlakozik közvetlenül a KIF keretrendszer, mely végzi az automatizált felhasználói felület teszteket.

Az Xcode projekt fájlstruktúrája független a konkrét projekt fájlstruktúrájától (de lehet teljesen megegyező is), tehát ha az Xcode projekthez hozzá szeretnék adni egy új teszt fájlt, akkor ahhoz nem elég csak a teszt fájlt bemásolni a megfelelő helyre, hanem az Xcode projekt fájlt is módosítani kell vagy csak egyszerűen kézzel hozzá kell adni a fájlt a projekthez az Xcode-on belül.

A KIF keretrendszer használatához első lépésként a projekt teszt sémáját kell beállítani, melyek főleg projekt és séma konfigurációs beállítások. Ezután a KIF már egyszerűen használható. Egy új *KIFTestCase*-ből leszármaztatott teszt fájlt kell létrehozni, mely tesztjei már futtathatóak is az Xcode-ban. Az Xcode-hoz ezután már könnyű hozzáadni új teszt fájlokat (akár egy Eclipse programból is).

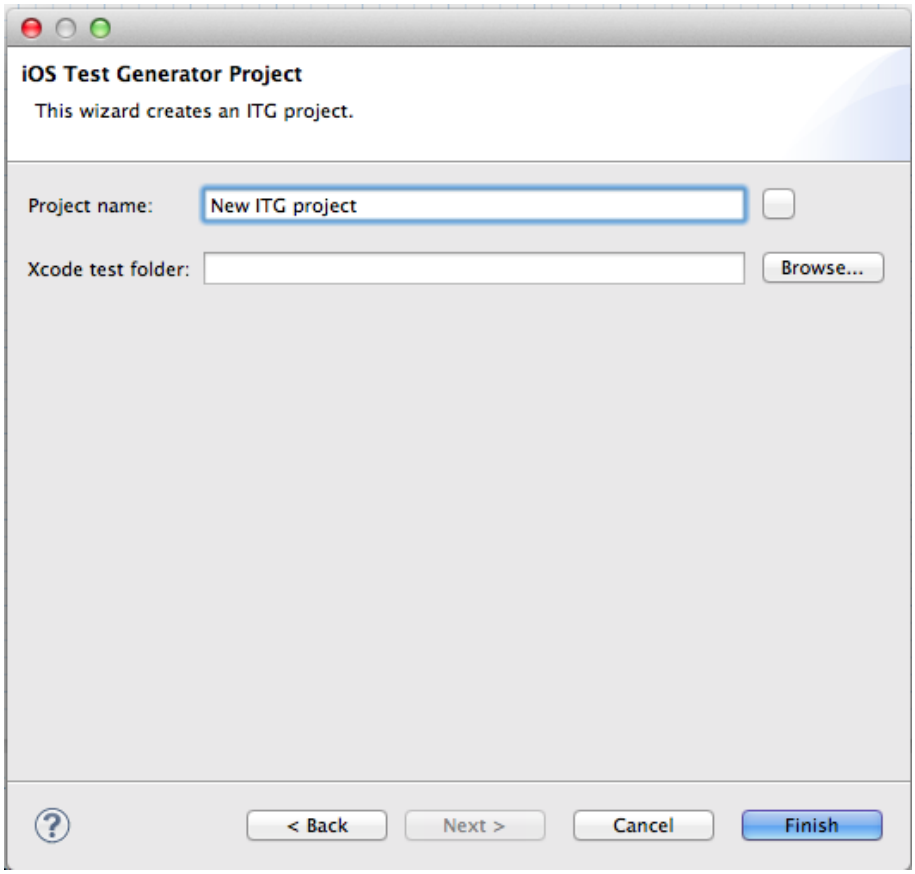
3.3. Az Eclipse projekt felépítése

Egy Eclipse plug-int készítettem, mellyel egy olyan *iOS Test Generator* nevű projektet lehet létrehozni, mellyel felhasználói felület teszt kódot lehet generálni a kapcsolatban álló

iOS projekttel, vagyis annak Xcode projektjével. A plug-in tartalmaz egy grafikus teszt szerkesztőt, melyből generálja a plug-in a teszt kódokat.

Az plug-in részeként létrehoztam egy új projekt varázslót (3.2. ábra), melyben megadható a projekt neve és az Xcode-ban lévő teszt mappa elérési útvonala. Ehhez a *Wizard* osztályból kellett leszármaztatnom, ahol létrehozhattam a saját *WizardPage* osztályból leszármaztatott varázsló oldalamat. A varázslóban két dolgot lehet megadni: a projekt nevét és az Xcode projektben lévő teszt mappa elérési útvonalát. Ezután meghatároztam a projekt kezdeti felépítését, hogy milyen fájlok és mappák generálódjanak le a projekt létrehozása után:

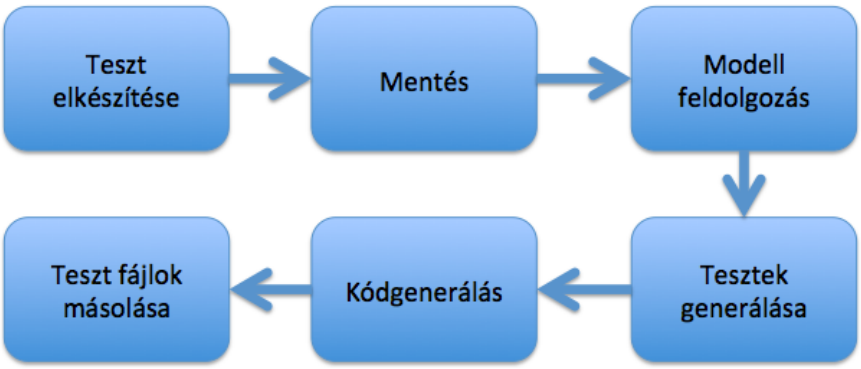
- **model** mappa, mely a grafikus modelleket tartalmazza
  - **model.bpmn** fájl, egy üres BPMN2 fájl, mellyből ki lehet indulni teszt készítésekor
- **src-gen** mappa, ide kerülnek a modellből generált Objective-C teszt
- **appSettings.application** konfigurációs fájl



3.2. ábra. *iOS teszt generáló projekt varázslója.*

A konfigurációs fájl (*appSettings.application*) az alkalmazás-specifikus beállításokat tartalmazza, mely az Xcode teszt mappájának elérési útvonala, illetve az alkalmazás felhasználói felület elemeinek az *accessibilityLabel* paramétereit tartalmazza. A varázslóban beállított Xcode teszt mappa elérési útvonala ebben a fájlban írható át.

A projekt a teljes tesztgenerálás folymatért felelős. A teljes folyamat hat fő lépésből áll, melyet az alábbi ábra is mutat (3.3. ábra).



3.3. ábra. *A teljes folyamat, melyet az Eclipse projekt elvégez.*

3.4. A BPMN2 szerkesztő

A BPMN2 szerkesztő lehetőséget nyújt testreszabhatóságra és kiegészíthetőségre, emiatt is esett erre a választásom. Létrehoztam egy saját plug-int, melyben kiegészítettem a BPMN2 egyes funkcióit. Két fő kiegészítést eszközöltem: hozzáadtam saját grafikus elemeket a grafikus szerkesztőhöz és ezekhez saját változtatható paramétereket adtam hozzá, illetve a modell mentésekor lefuttatok egy kódgenerálást a modell alapján.

A BPMN2 szerkesztőt bár kiegészíthetőre tervezték, mégsem olyan egyszerű a kiegészítése vagy továbbfejlesztése. Elsősorban amiatt, mert üzleti folyamatok leírására és azok kiegészítésére tervezték, nem tesztek leírására. Nehézséget jelentett a dokumentáció hiánya, így sok funkciót egyáltalán nem, sok funkciót pedig csak nehézségek árán tudtam megvalósítani, viszont kiindulásnak hasznos volt a BPMN2 oldalán található tutorial.

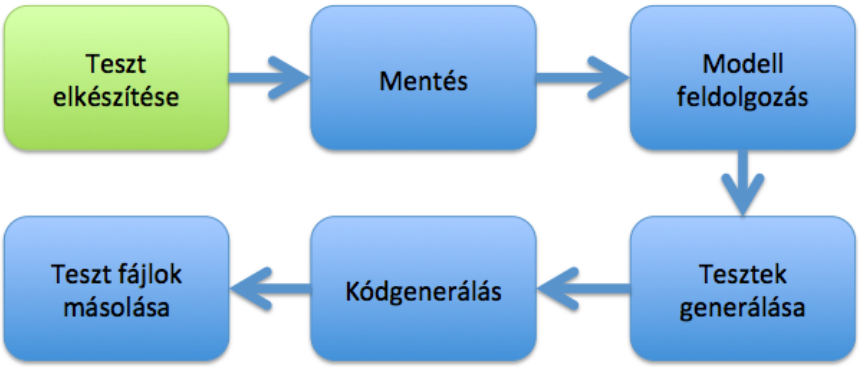
3.4.1 A BPMN2 szerkesztő kiegészítése

A BPMN2 könnyen használható tesztek készítésére. Én egyedi KIF teszt lépést és elvárt eredmény ellenőrzés elemeket akartam használni. Ehhez először is létrehoztam egy saját projektet *hu.bme.mit.mobilegen.iostestgenerator.bpmneditor* néven, melyben kiegészítettem a beépített szerkesztőt.

Következő lépésként megterveztem a két egyedi BPMN2 feladatomat, melyek a teszt lépés és az elvárt eredmény ellenőrzés. Mindkét feladathoz két paramétert szükséges megadni a szerkesztőben:

- **accessibilityLabel**, mely a felhasználói felület *accessibilityLabelét* határozza meg
- **KIFAction**, mely egy KIF keretrendszerbeli esemény. Az *accessibilityLabellel* meghatározott felhasználói felületen KIFAction esemény lesz végrehajtva.

Mivel a két feladat paraméterei megegyeznek, ezért egy közös EMF modellt hoztam létre. A modell tartalmazza a BPMN2 feladathoz tartozó kötelező elemeket, mint a *Do-*



3.4. ábra. *A folyamat első lépése a teszt szerkesztése.*

cumentRoot, *MyEventDefinition* vagy a *TaskConfig*, ezen kívül tartalmazza az accessibilityLabel és KIFAction paramétereket is. Az EMF modelt ecore fájlban hoztam létre a BPMN2 ecore fájlhoz hasonlóan (mely az *org.eclipse.bpmn2/model/BPMN20.ecore* útvonalon található a BPMN2 projekten belül).

Ezután a saját projektemhez hozzáadtam egy Target Runtime-ot, mely tartalmazza az általam készített kiegészítéseket, így a normál BPMN2 szerkesztőt nem kellett módosítanom. A Target Runtime-ot a .bpmn kiterjesztésű modell fájl beállításai között lehet megtalálni, és ha az általam készített *iOS Test Generator Runtime Extension* beállítást választjuk ki, akkor megjelennek a teszt feladatok, ellenkező esetben csak a normál BPMN2 szerkesztő áll rendelkezésünkre.

3.4.2 Az teszt lépés és az elvárt eredmény ellenőrzés

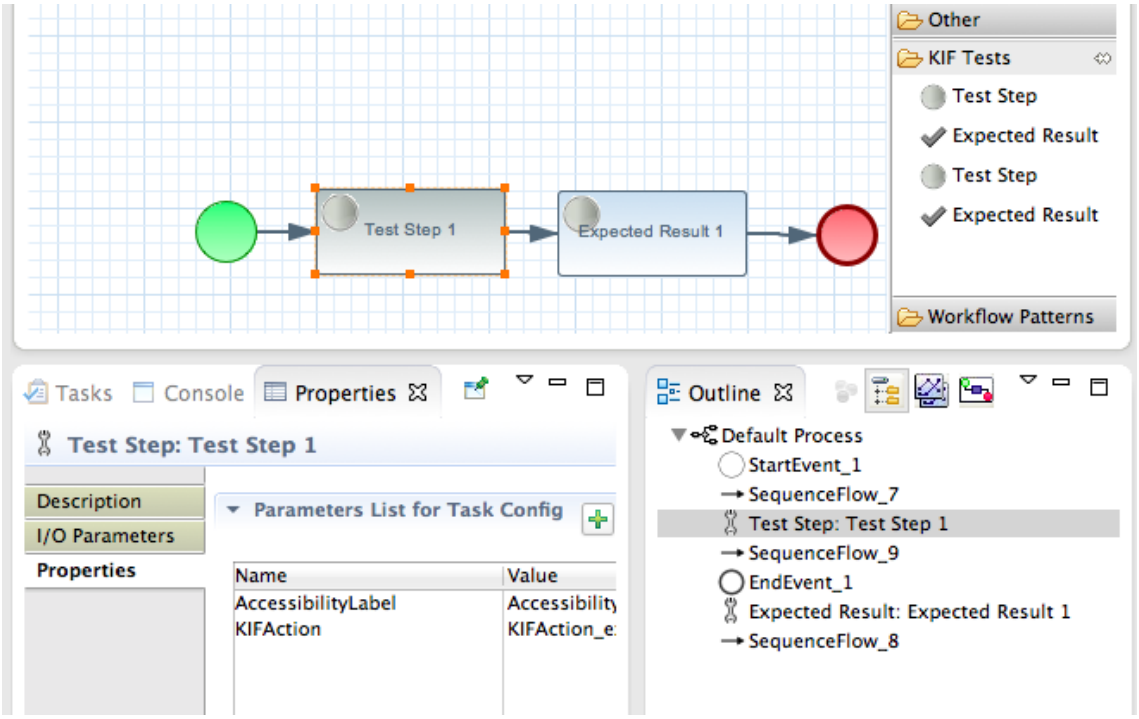
A saját Target Runtime-hoz hozzáadtam két új feladatot (task), a teszt lépést és az elvárt eredmény ellenőrzést. Míg az elágazás elemeket fel tudtam használni a BPMN2 eszközkészletéből, addig a fenti két feladatot létre kellett hoznom, mert csak így tudtam az általam meghatározott paramétereket hozzáadni a feladathoz. Új BPMN2 elemek hozzáadására a BPMN2 projekt *extension point*-jain keresztül van lehetőség, melynek működése a 2.6. fejezetben megtalálható.

Az alábbi *extension point*-okat kellett létrehoznom a két új BPMN2 elem definiálásához, az *iOS Test Generator Runtime Extension*-ön kívül:

- **model**: saját resource factory, mely felüldefiniálja a *Bpmn2ModelerResourceFactoryImpl* osztályt.
- **modelEnablement**: *model*hez tartozó kiegészítő *extension point*
- **propertyTab (Properties)**: egy egyedi beállítások nézet ahol meg lehet adni az accessibilityLabel és a KIFAction paraméterek értékeit
- **customTask (Test Step)**: a teszt lépést megvalósító BPMN2 elem, melyben sok egyéb mellett az alábbi fő paramétereket állítottam be:
  - *category* = *KIFTests*, mely azt jelenti, hogy a szerkesztő palettáján a KIFTests kategória alatt lesz a Test Step

- *propertyTab = Properties*, a fent említett beállításokat tartalmazó nézet
- *icon*, mely az elemhez tartozó ikont jelenti, ami megjelenik a szerkesztő felületen
- *extensionValues = KIFAction, accessibilityLabel*, melyek a KIFTests-hez tartozó paramétereket jelentik

- **customTask (Expected Result)**: az elvárt eredményt megvalósító BPMN2 elem, melyben a teszt lépéshez hasonlóan állítottam be a paramétereket:
  - *category = KIFTests*, mely azt jelenti, hogy a szerkesztő palettáján a KIFTests kategória alatt lesz az Expected Result
  - *propertyTab = Properties*, a fent említett beállításokat tartalmazó nézet
  - *icon*, mely az elemhez tartozó ikont jelenti, ami megjelenik a szerkesztő felületen
  - *extensionValues = KIFAction, accessibilityLabel*, melyek a KIFTests-hez tartozó paramétereket jelentik



3.5. ábra. *Példa egy teszt lépésre és egy elvárt eredmény ellenőrzésre.*

A fenti ábrán (3.5. ábra) már az általam kiegészített BPMN2 szerkesztőben készített teszt folyamat látható. Két hiányossága van, melyekre már sok időt fordítottam, de nem sikerült megoldani. Az egyik hiba az, hogy az elvárt eredmény elemnek megegyezik az ikonja a teszt lépésével, pedig különböző ikon van az extension point-ban beállítva, így BPMN2 belső hibára gyanakszok. A másik hiba a kép jobb oldalán található: a palettán az alapértelmezett kategóriák (pl. *Other*, *Workflow Patterns*) is megjelennek, pedig az ezekben lévő elemeket nem lehet használni a tesztek összeállításához. Utóbbi problémát nem sikerült megoldanom. A BPMN2 extension pontjait

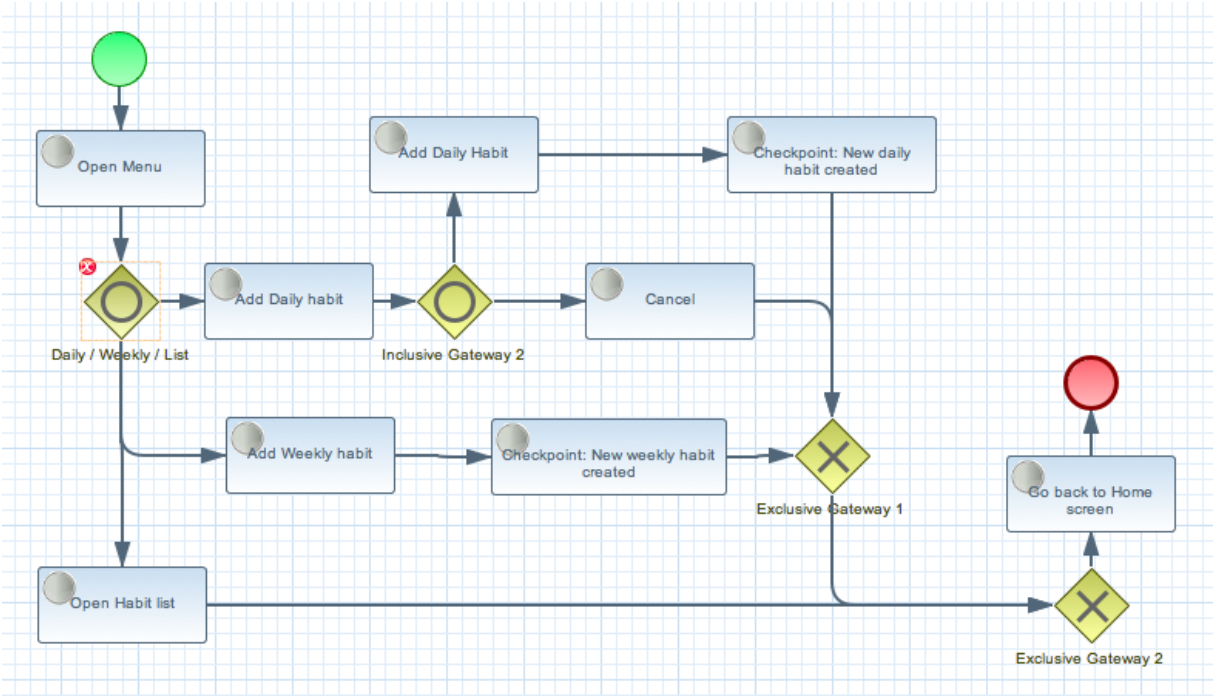
próbáltam módosítani és törölni, majd az erdeti BPMN2 szerkesztőből is próbáltam kitörölni ezeket az elemeket, de nem jártam sikerrel.

3.4.3 A teszt elágazás

Már a BPMN2 szerkesztő kipróbálása előtt terveztem elágazást a tesztek leírásába, a BPMN2 szerkesztő pedig beépített grafikus elemmel rendelkezik (*Gateway*), mely alkalmas a tesztek leírásában az elágazás megvalósítására. Elágazás nélkül egy elkészített teszt valójában egy konkrét tesztet jelent, viszont ha a tesztbe elágazásokat teszünk, akkor több tesztet lehet generálni. Ezen ötlet alapján használtam fel a BPMN2 szerkesztő *Inclusive Gateway* és *Exclusive Gateway* elemeit elágazások megvalósítására.

Unit tesztek esetén nincs olyan nagy jelentősége az elágazásnak, de felhasználói felület tesztek esetén annál inkább. Például, ha a teszt során olyan helyre jutunk ahol több úton is tovább lehet menni az alkalmazásban (pl. mentés vagy mégse gomb, menü több opciója), akkor a teszt során csak egy elágazást kell beiktatni. Ha a teszt további lépései megegyeznek, akkor az elágazást be is lehet egyből zárni, ha pedig különböző úton megy tovább a teszt például a menü különböző elemeit kiválasztva, akkor az elágazás bezárását csak ezen tesztlépések után kell betenni a grafikus modellbe.

A BPMN2 szerkesztőnek van saját validációja, de ez igencsak szűkös. Alap esetben a *Gatewayekre* csak azt ellenőrzi, hogy a típusuk *Converging* vagy *Diverging*-e, ha nem akkor hibásnak jelzik a modellt, hiába érvényes a modellünk (3.6. ábra). Az ábrán a modell érvényes, de a BPMN2 szerkesztő hibásnak jelzi, mert az első elágazás Gateway Direction paramétere nem megfelelő. Itt is megmutatkozik a tervezési különbség a BPMN2 szerkesztő és az általam készített teszteket leíró szerkesztő között.

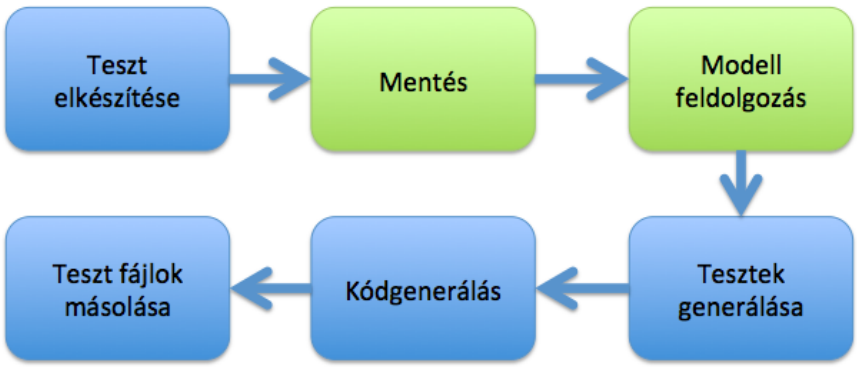


3.6. ábra. *Példa érvényes modellre, melyet a szerkesztő mégis hibásnak jelez egy beállítás miatt.*

A BPMN2 saját validációjának köszönhetően, ha nem megfelelően definiálom az elágazásokat, akkor a modell hibát jelez, mely egyben értelmetlen tesztet is jelent. Ez egy teljesen életszerű felhasználói eset, hiszen nagyon nagy és bonyolult teszteket is le lehet írni a szerkesztőben, könnyen rossz helyre lehet kötni egy folyamatot. Ilyen eset lehet például az, ha egy elágazásból három út megy tovább, de a háromból csak kettő egyesül, egy pedig elvarratlan szálon marad, vagy rossz szálba csatlakozik vissza.

3.5. A BPMN2 modell feldolgozása

A BPMN2 modellt, mint grafikus modellt önmagában elég nehéz lenne feldolgozni. A modellnek természetesen van valamilyen tárolt változata, melyet újra megnyitva ugyanazt a modellt kapjuk vissza, de ez sok grafikus modell esetében nem egyértelmű, például az *Xcode Storyboard* szerkesztője egy XML-ben tárolja a storyboardot, melynek sémája nem publikus. A BPMN2 szerencsére a grafikus modellt EMF modellben tárolja, mely könnyen olvasható és írható. Számomra csak az olvasás szükséges.



3.7. ábra. *A mentés hatására indul el a modell feldolgozása.*

A modellfeldolgozást a BPMN2 szerkesztő mentés folyamata közben lehetne elindítani, vagy egy callback függvénnyel, mely akkor hívódik meg, amikor a fájl mentése végrehajtódik. A második opciót választottam, mert az Eclipse a fájlok mentéséről az *IResourceChangeListener* interfészen keresztül egyszerűen lehet értesülni, míg a BPMN2 szerkesztő mentési folyamatát valamivel nehezebb felüldefiniálni. Az *IResouceChangeListner* képes értesítést küldeni az aktuális projekt állapotáról, melyek az alábbiak lehetnek:

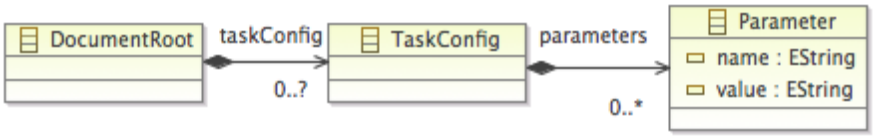
- **PRE_CLOSE**: projekt bezárása előtti állapot
- **PRE_DELETE**: projekt törlése előtti állapot
- **POST_CHANGE**: a projektben belül egy vagy több fájl változása utáni állapot
- **PRE_BUILD**: projekt fordítása előtti állapot
- **POST_BUILD**: projekt fordítása utáni állapot
- **POST_REFRESH**: projekt frissítése utáni állapot

Nekem a projekt fordítása utáni állapotra van szükségem, mert a BPMN2 szerkesztő is lefordítja EMF modell szintre a diagramot, nekem pedig a fordítás után előállított EMF modellre van szükségem. A *POST_BUILD* állapotról értesülés után, még lehetőségem van a fájlok változásáról, létrehozásáról vagy törléséről pontos állapotot kapni az *IResourceChangeListener*-től lekérdezhető delta objektummal (mely az *IResourceDeltaVisitor* interfészt valósítja meg). A változott fájlok közül nekem csak a model mappában lévő, *bpmn* kiterjesztésű fájlokra van szükségem, tehát csak az ilyen fájlokra végzem el a BPMN2 modell EMF modell beolvasását és feldolgozását.

3.5.1 Az EMF modell

A BPMN2 diagram szerkesztő egy EMF modell, melynek az *Editor* részét, azaz a modellben tárolt adatok szerkesztéséért felelős részt valósították meg egy diagram szerkesztővel. A *Model* rész felépítése teljesen megegyezik bármilyen más EMF projekt modelljével a modell komplexitásán kívül, mivel a BPMN2-nek rendkívül bonyolult EMF modellje van, mely több mint 100 osztályt tartalmaz. Ennek természetesen csak kis részét kell felhasználnom, mert a palettáról is csak néhány elemet használok, illetve a teljes grafikus modell megjelenésért felelős részt figyelmen kívül hagyhatom.

Két saját elemet adtam hozzá a szerkesztőhöz, a teszt lépést és az elvárt eredmény ellenőrzést, melyeknek a modell része azonos, ezért nem kellett két modellt is létrehoznom, elég volt csak egyet. Mivel a BPMN2 EMF alapokra épült, ezért ha új elemet akarok hozzáadni a szerkesztőhöz, akkor egy saját EMF modellel is ki kell egészítenem a BPMN2 szerkesztőt. Ezt a *hu.bme.mit.mobilegen.iostestgenerator.bpmneditor* projektben tettem meg. Létrehoztam egy *MyModel* ecore modellt (3.8. ábra), melyben kiegészítettem a BPMN2 EMF modelljét egy *TaskConfig* és egy *Parameter* osztállyal. A *TaskConfig* biztosítja azt, hogy a BPMN2 *Task* eleméből leszármaztatva, egy saját elemet hozhassak létre, mely a *Task* elem minden tulajdonságaival rendelkezik, illetve ezt ki is egészíthetem sajátokkal. A kiegészítés a *Parameter* osztály, mely tartalmazza az újonnan megadható paraméterek nevét és értékét.



3.8. ábra. *A BPMN2 EMF modelljének kiegészítése.*

Az BPMN2 modellt a feldolgozás során ugyanúgy kell kezelni mint egy EMF modellt. A *Bpmn2ResourceFactoryImpl* osztályt példányosítva készítek egy BPMN2 erőforrást (Resouce), melybe betöltöm a BPMN2 modellt, és innentől kezdve tudom használni a BPMN2 legenerált modell osztályait (*org.eclispe.bpmn2* projekt, *org.eclipse.bpmn2* mappájában vannak), illetve az általam készített *MyModel* modellből generált osztályokat is.

A modell feldolgozást a *hu.bme.mit.mobilegen.iostestgenerator* projekt *BpmnModelProcessor* osztályának *parseBPMN* metódusa végzi, ebből mutat be egy részletet az alábbi forráskód (3.1 forráskód). A *Bpmn2ResourceFactoryImpl* által készített és a fájlból betöltött erőforrásból (*Resrouce resource*) az első folyamatot (*Process process*) dolgozom fel, mert csak egy folyamatot kezelem a diagramon, ezután a folyamat összes elemének listáját meg tudom kapni, azt feldolgozva pedig a *parsedBPMNModel* objektumba rakom a megfelelő paraméterek beállítása után az elemeket.

3.1. Forráskód. *A BPMN2 EMF modelljének feldolgozása.*

```
Bpmn2ResourceFactoryImpl resFactory = new Bpmn2ResourceFactoryImpl();
Resource resource = resFactory.createResource(modelFileURI);
resource.load(Collections.EMPTY_MAP);

Definitions d = (Definitions) resource.getContents().get(0).eContents().get(0);
Process process = (Process) d.getRootElements().get(0);
List<FlowElement> flowElements = process.getFlowElements();

ParsedBPMNModel parsedBPMNModel = new ParsedBPMNModel();

for (FlowElement possibleFlowElement: flowElements) {

        ParsedBPMNNode parsedBPMNNode = null;

        if (possibleFlowElement instanceof org.eclipse.bpmn2.EndEvent) {
                EndEvent bpmnEndEvent = (EndEvent) possibleFlowElement;

                ParsedBPMNNodeEndEvent endEvent =
                        new ParsedBPMNNodeEndEvent("bpmn2:endEvent", "");
                endEvent.setId(bpmnEndEvent.getId());
                endEvent.setIncoming(bpmnEndEvent.getIncoming().get(0).getId());
                parsedBPMNModel.add(endEvent);
        }
        ...
}
...
```

Az BPMN2 EMF modelljének feldolgozása után kapok egy *ParsedBPMNModel* objektumot, mely tartalmazza a grafikus szerkesztő összes elemét, illetve azoknak a fontos paramétereit. Ezt az objektumot fogom feldolgozni a tesztek generálása során, ahol már az elágazásokat fel kell oldani, több tesztet kell generálni egy folyamatból.

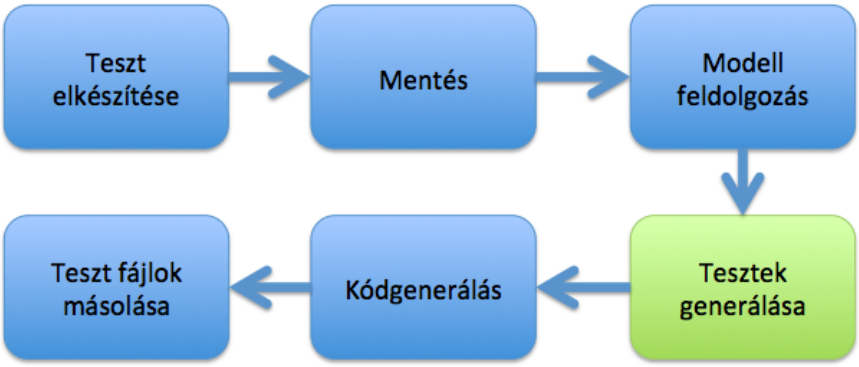
3.5.2 Tesztek generálása

Mivel a tesztek tartalmazhatnak elágazást is, ezért egy modell több tesztet is leír. Az összes elágazás kimenetének számával megegyező számú teszt fog legenerálódni a modellből, mert az elágazás első ága nem generál több tesztet, csak az eddigi tesztet folytatja. Ezek alapján a teszt generálás folyamata az alábbiak szerint zajlik:

1. startEvent outgoing kimenetének megkeresése, ennek megfelelő első elem megkeresése
2. ha az elem teszt lépés elem, akkor azt el kell tárolni egy teszt sorozatban
3. ha az elem elvárt eredmény ellenőrzés, akkor azt el kell tárolni egy teszt sorozatban

4. ha az elem elágazás kezdete, akkor a jelenlegi tesztet elágazás kimenetének száma minusz egyszer kell lemásolni
5. ha az elem elágazás vége, akkor az elágazás összes ága ugyanazon teszt lépés értékeket kapja a további lépésekben
6. ha az elem összeköttetés, akkor tovább lépünk az összeköttetés célelemére
7. ha az elem endEvent, akkor vége a tesztek generálásának

A fenti teszt generálás után előáll az összes teszt – tekintsünk rájuk egy tömbként –, egy tesztben teszt lépések és elvárt eredmény ellenőrzések sorozata található, illetve azok paraméterei. Ezekből a kódgenerátor ki tudja olvasni az összes szükséges adatot a teszt kódok generálásához.



3.9. ábra. *Tesztek generálása.*

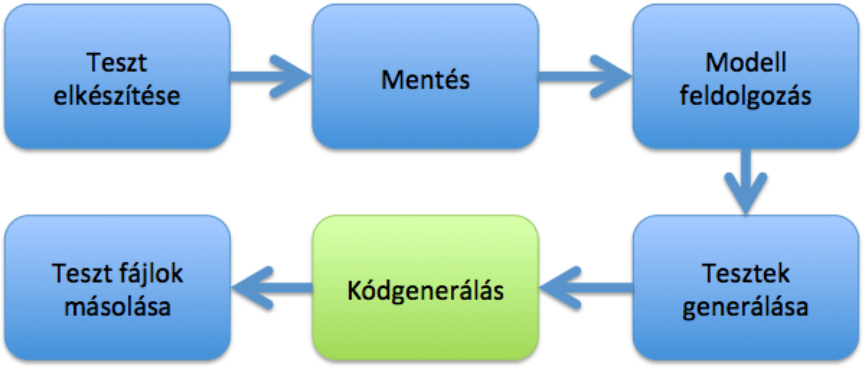
3.6. A kódgenerátor

A BPMN2 grafikus modell feldolgozás és tesztgenerálás után a tesztekből kódgenerátor segítségével Objective-C kódot kell generálni. Erre a feladatra az Eclipse Xtend technológiát használtam, mert támogatja a sablon alapú kódgenerálást. Ez azt jelenti, hogy olyan sablonokat kell létrehoznom, melyek tartalmaznak egy pontos és szintaktikai hiba mentes Objective-C teszt osztály deklarációt, illetve egy KIF keretrendszer tesztet.

A tipikus architektúra az kódgenerálas esetén, hogy a forráskódot és a generált kódot különválasztjuk és más módon kezeljük. A legtöbb Eclipse projektben például található src és src-gen mappa, ahol az src tartalmazza a forráskódot, az src-gen pedig a generált kódot. A modellem nekem is hasonló: a model mappámban lévő grafikus modellből generálom le a teszt kódokat az src-gen mappába, melyet már nem szerkesztek, mert a következő generáláskor (mely minden modell mentéskor megtörténik) teljesen felülírjuk az új generált kóddal.

A kódgenerálást a BPMN2 modellből generált tesztek tömbjére futtatom az alábbi lépésekben:

1. KIF tesztfájl fejlécének generálása (3.2 forráskód)



3.10. ábra. *A kódgenerálás az utolsó előtti lépés a teljes folyamatban.*

3.2. Forráskód. *KIF tesztfájl fejlécének generálása.*

```
def testFileBegin(String fileName, String projectName)'''
//
//  <<fileName>>.m
//  <<projectName>>
//
//  Created by Csaba Szabo on 20/10/13.
//  Copyright (c) 2013 SSK Development. All rights reserved.
//

#import <KIF/KIF.h>

@interface <<fileName>> : KIFTestCase
@end

@implementation <<fileName>>
'''
```

2. minden generált tesztre az alábbiak

   (a) teszt metódus fejléce (3.3 forráskód)

3.3. Forráskód. *KIF teszt metódus fejlécének generálása.*

```
def testMethodBegin(String methodName) '''
- (void) test<<methodName>>
{
'''
```

   (b) minden teszt metódus lépésre

      i. teszt lépés vagy elvárt eredmény lépés generálása, mely a többi sablonhoz hasonló KIF esemény sablonokat használja

   (c) teszt metódus lezárása (3.4 forráskód)

3. KIF tesztfájl lezárása (3.5 forráskód)

A sablon által generált kódon nem futtatunk szintaktikai ellenőrzést, tehát ha bármelyik sablonban hibát vétünk, akkor szintaktikai hibával rendelkező kódot kapunk, ezért a sablonokra nagyon nagy figyelmet kell fordítani.

3.4. Forráskód. *KIF teszt metódus lezárásának generálása.*

```
def testMethodEnd() '''
}
'''
```

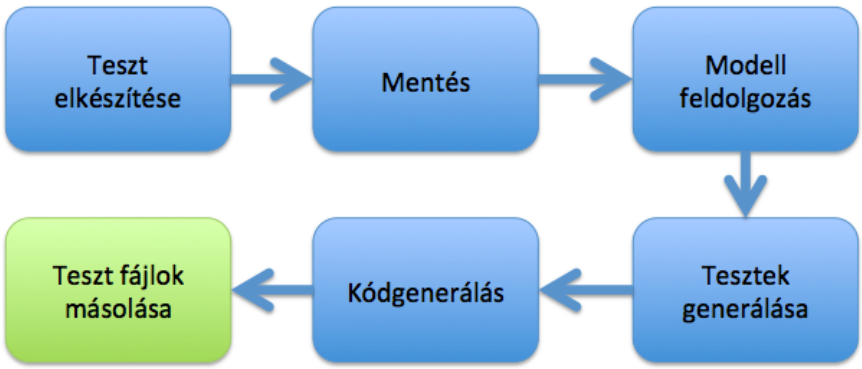
3.5. Forráskód. *KIF tesztfájl lezárásának generálása.*

```
def testFileEnd() '''
@end
'''
```

3.7. Az Eclipse és az Xcode közötti kapcsolat

Az architektúra tervezésénél már figyelembe vettem, hogy együtt kell működnie az Xcode és az Eclipse projekteknek. Ez a kommunikáció egyirányú, tehát csak az Eclipse projekt olvassa és írja az Xcode projektet.

3.7.1 Teszt fájlok másolása



3.11. ábra. *A teljes folyamat utolsó lépéseként átmásoljuk a legenerált teszt fájlokat a Xcode teszt mappájába.*

Az Xcode projekt fájl (*.xcodeproj*) több leíró fájlból áll, melyekből a *project.pbxproj* fájl tartalmazza a projekt logikai fájlstruktúráját. Ez egy XML leíró fájl, melynek kézzel történő szerkesztése nem egyértelmű. Létezik egy népszerű megoldás erre a problémára, az XcodeEditor[20], mely egy Xcode projekt, melyet iOS alkalmazásokhoz szoktak használni, tehát az Eclipse szoftveremen belül konkrétan nem tudom használni. Ezt a problémát az Eclipse plug-inből megoldani sajnos nem is lehet anélkül, hogy az Xcode projektben kézzel módosítsunk. Tudnék olyan Objective-C kódot generálni, mely hozzáadja a teszt fájlokat az Xcode projekthez, viszont ezt a kódot tartalmazó fájlt már kézzel kell hozzáadnom a projekthez, így nincs értelme ezt a módszert használni. Ezen okok miatt a szoftverem által generált teszt fájlokat kézzel kell hozzáadni az Xcode projekthez, majd ezután a meglévő teszt módosítása és újragenerálása már automatikusan megjelenik az Xcode projekten belül. Erre a problémára létezik

megoldás, például ha az *XcodeEditor* forráskódját tanulmányozva (lehetséges, mert nyílt forráskódú a projekt) a fájl hozzáadás metódust megvalósítanánk Java nyelven az Eclipse projekten belül.

3.7.2 AccessibilityLabelek kezelése

A KIF keretrendszer a felhasználói felület elemek *accessibilityLabelein* keresztül éri el az alkalmazás felhasználó felületét. A grafikus teszt szerkesztő hozzáadott értéke lehet az, hogy csak olyan *accessibilityLabelt* lehet megadni egy lépésnél, ami létezik is az alkalmazásban, ellenkező esetben a teszt eredménye sikertelen lesz.

Ahhoz hogy ellenőrizni tudjam az *accessibilityLabelek* létezését, először az alkalmazás *accessibilityLabeleit* kell összegyűjteni. Erre az alábbi lehetőségek állnak rendelkezésre:

- Az *accessibilityLabelek* listájának megadása külön fájlban az Eclipse projekten belül,
- az Xcode storyboard fájlból az *accessibilityLabel* XML elemek beolvasása, mely csak akkor használható, ha az alkalmazás felhasználói felületét csak storyboard fájlok írják le,
- az alkalmazás kódjának beparszolása, majd ebből az *accessibilityLabelek* kigyűjtése, mely csak akkor használható, ha az alkalmazás felhasználói felülete csak programkódból készül el.

Mivel a második és harmadik lehetőség csak együtt igazán használható és ezek együttes megvalósítása nagyon komplex és bonyolult, így az első pontban szereplő lehetőséget használom, melyre az *appSettings.application* fájlban van lehetőség.

A kódgenerálás után a plug-in leellenőrzni, hogy a grafikus modellben megadott *accessibilityLabel* létezik-e az *appSettings.application* fájlban. Az *appSettings.application* fájl a projekt létrehozása során legenerálódik, de kézzel is hozzá lehet adni a *.application* kiterjesztésű fájlt az új fájl varázslóból. A fájlban az *Application* elemhez kell hozzáadni új *UI Element* elemeket, melyek paraméterében megadható az *Accessibility Label* érték. A diagramon megadott accessibilityLabel értéket keresi ki ebből a listából a program. Ha nem létezik, akkor a forráskódban megjelenik egy szintaktikai hibát tartalmazó sor (`HIBAS ACCESSIBILITYLABEL: accessibilityLabel`). Erre mutat példát az alábbi generált teszt forráskód (3.6 forráskód).

A forráskódba írt szintaktikai hiba helyett alkalmazható lett volna a grafikus modellen a hibás *accessibilityLabelt* tartalmazó elem fölött egy hibát jelző X jel, de a BPMN2 szerkesztő ilyen szintű módosítása nem volt egyértelmű, viszont az *accessibilityLabel* hiba a szintaktikai hiba esetén az Xcode-ban is egyértelműen látszódik.

3.6. Forráskód. *A hibás accessibilityLabel miatt szintaktikai hibás kód generálódott.*

```
- (void) test7
{
        [tester tapViewWithAccessibilityLabel:@"DailyHabitAddButton"];
        [tester enterText:@"daily habit"
                intoViewWithAccessibilityLabel:@"NewHabitNameTextfield"];
        [tester tapViewWithAccessibilityLabel:@"NewHabitNameFreqChanger1"];
        HIBAS ACCESSIBILITYLABEL: NewHabitCancelButtoon

        // Expected result
        [tester waitForViewWithAccessibilityLabel:@"HabitCell"];

}
```

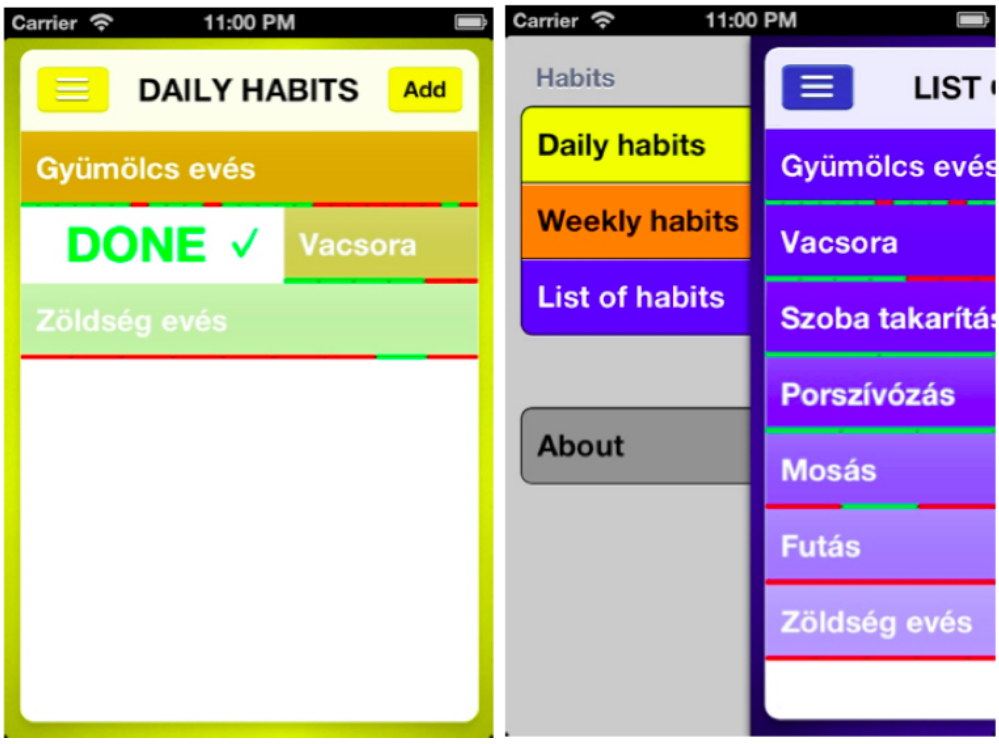
4. fejezet

Példaalkalmazás bemutatása

Ebben a fejezetben bemutatom a szoftver funkcióit egy általam készített Habits nevű iOS alkalmazáson. A szoftver használatának bemutatásával reményeim szerint a szoftver tervezési lépéseinek okai és a működése (3. fejezet) jobban érhetőek lesznek az olvasó számára.

4.1. A Habits alkalmazás

A Habits egy szokások menedzselésére szolgáló alkalmazás. Az emberek mindennapjait kb. 40%-ban határozzák meg a szokások, ezeket a szokásokat tudjuk nyomon követni, alakítani és új szokásokat kezdhetünk el. Minden nap a következő nap szokásait automatikusan legenerálja, illetve gesztussal vezérelve lehet őket törölni. Jelenleg kétféle szokást kezel az alkalmazás: napi és heti szokást. Az ötletet a Chains.cc alkalmazás adta, de nem annak a másolata, hanem egy hasonló célú, de más funkcionalitású szoftver a Habits.



4.1. ábra. *A Habits alkalmazás néhány képernyőképe.*

Új szokásainkat fel tudjuk venni napi vagy heti szokásként, melyek napváltáskor újra legenerálódnak. A szokásokat lehet törölni vagy teljesíteni, illetve minden szokás alatt zöld és piros sávban lehet látni (4.1. ábra), hogy a kezdés óta eltelt napokban mikor teljesítettük (zöld) a szokást és mikor nem (piros).

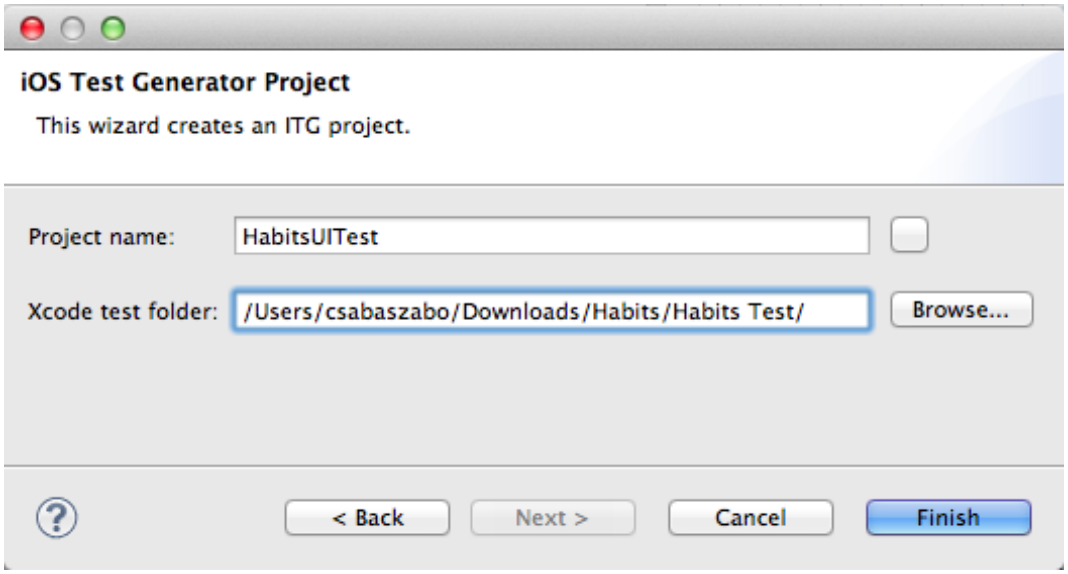
4.1.1 Felhasználói felület tesztek

Az alkalmazás SQLite adatbázist használ és napváltáskor az új szokás elemek generálása viszonylag bonyolultabb üzleti logika lépés, tehát ennek a tesztelése hasznos lenne, de felhasználói felület tesztek bemutatására is alkalmas az alkalmazás. Több képernyőből áll, a felhasználó több funkciót is elvégezhet a felületen, melyek tesztelése szintén hasznos lehet. A teljesség nélkül az alábbi felhasználói felület teszteket mindenképp érdemes kipróbálni:

1. A menüben szerepelő összes aloldal (napi-, heti-, összes szokás, egyéb) megnyitása, majd a menü újbóli megnyitása,
2. új napi/heti szokás felvétele annak mentése/elvetése, létrejöttének ellenőrzése,
3. új napi/heti szokás teljesítése/törlése, azoknak ellenőrzése a napi/heti/összes szokások listájában,
4. a szokásteljesítések múltjának (zöld-piros csíkok) ellenőrzése.

A fenti listából látszik, hogy már a tesztek szöveges megfogalmazásakor több opció előfordult (pl.: *Új napi/heti szokás felvétele annak mentése/elvetése, létrejöttének ellenőrzése.*), ezek leírására tökéletes eszköz a teszt szerkesztő elágazás eleme.

4.2. Új ITG projekt készítése



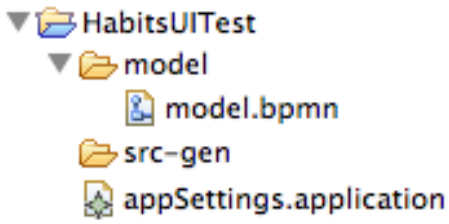
4.2. ábra. *Az ITG projekt varázslója.*

A felhasználói felület teszt generátor szoftverem feltételezi az iOS alkalmazás forráskódjához való hozzáférést, tehát mikor az ITG projektet létrehozzuk, akkor már feltételezzük

az Xcode projekt létezését, az abban lévő teszt mappa létezését és hogy az Xcode projektbe importálva van a KIF keretrendszer. A KIF keretrendszer importálása a KIF GitHub projekt oldalán megtalálható[1]. Az ITG projekt varázslójában két dolgot kell megadnunk: projekt nevét és az Xcode teszt mappájának elérési útvonalát (4.2. ábra).

Miután a varázslóval létrehoztuk az új projektet, az alábbi projekt struktúrával készül el a projekt (4.3. ábra):

- **model** mappa, mely a grafikus modelleket tartalmazza. Hozzá lehet adni több BPMN2 modellt is,
  - **model.bpmn** fájl, egy üres BPMN2 fájl, melyből ki lehet indulni az első teszt készítésekor,
- **src-gen** mappa, a modell(ek)ből generált Objective-C teszt(ek) kód másolata, mely az Xcode teszt mappába is kerül,
- **appSettings.application** konfigurációs fájl, mely tartalmazza az Xcode teszt mappa elérési útvonalát, illetve az accessibilityLabelek listáját.



4.3. ábra. *Az újonnan létrehozott ITG projekt struktúrája.*

Miután létrehoztuk az új ITG projektet, első lépésként be kell állítani a *BPMN2 Target Runtime*-ot az alábbi lépésekkel. Ez ahhoz szükséges, hogy a BPMN2 szerkesztőben az általam hozzáadott KIF elemek megjelenjenek.

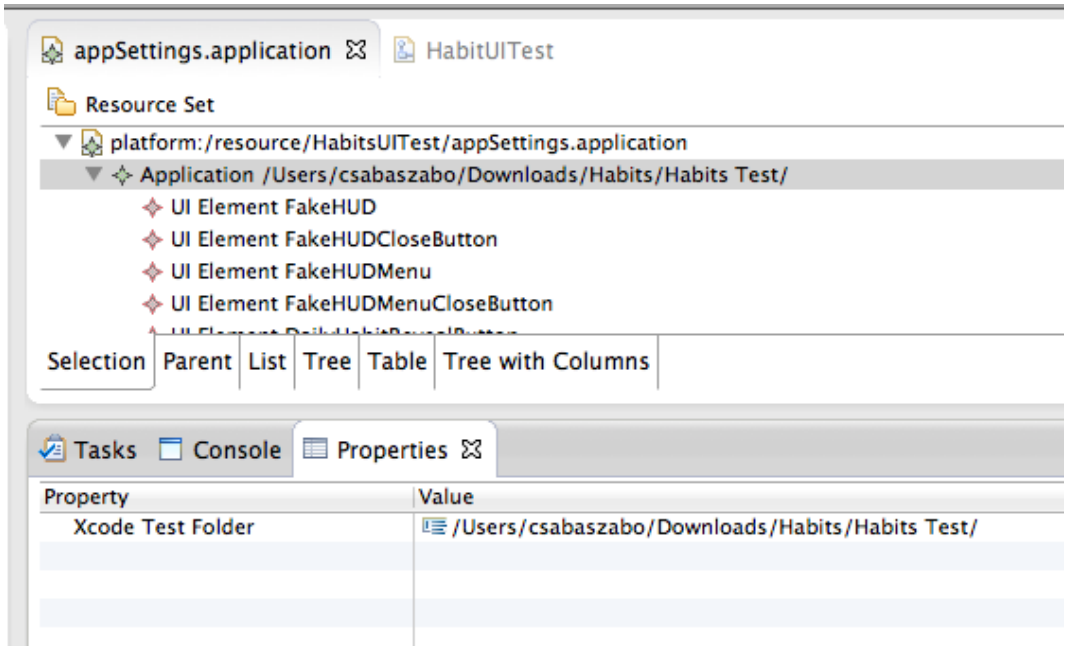
1. Jobb klikk az ITG projekten, majd *Properties* opció kiválasztása
2. *BPMN2* beállítások oldal kiválasztása
3. a *Target Runtime* beállítása *iOS Test Generator Runtime Extestionre*

Első lépésként érdemes az *appSettings.application* fájlban a felhasználói felület elemeket felvenni és megadni a hozzájuk tartozó accessibilityLabel értéket. A kódgenerálás során ellenőrzi a program, csak az ebben a listában szereplő accessbilityLabelű elemekből lesz szintaktikai hiba mentes generált teszt kód. Az *Application* jobb egérrel kattintva megjelenik az új UI Element hozzáadásának lehetősége. Erre rákattintva létrejön az új UI Element, melyre ha rákattintunk, akkor a Properties nézetében megadható az accessibilityLabel értéke.

[1] A KIF projekt oldalán [5] az „Add KIF to Your Workspace" fejezete.

4.3. Az Eclipe és Xcode projektek kapcsolata

A varázslóban megadott teszt mappa elérési útvonallal felállítottuk az Eclipse-Xcode egyirányú kapcsolatot. Az *appSettings.application* fájlban bármikor átírható a teszt mappa elérési útvonala. Az *appSettings.application* fájlt megnyitva, a modell struktúráját kibontva, ha az *Application* elemet jelöljük ki, akkor a tulajdonságai nézetben (Properties) megadható az Xcode teszt mappa elérési útvonala (4.4. ábra).



4.4. ábra. *Az appSettings.application fájl szerkesztő felülete.*

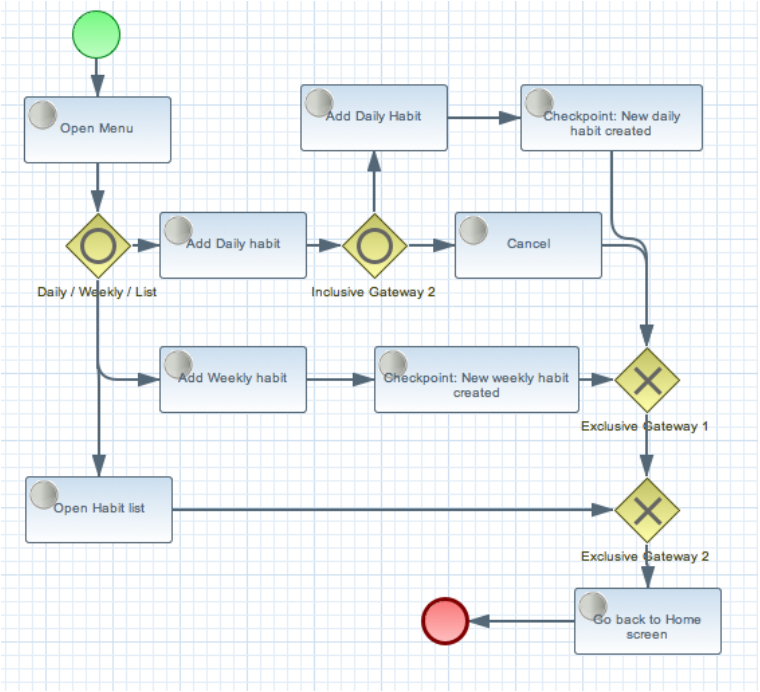
Ahhoz, hogy az Xcode projekt teszt mappájában megjelenjen a teszt fájl, ahhoz be kell importálni az Xcode projektbe, ekkor az Xcode elkészíti a referenciát. Ezután, ha változtatuk a grafikus teszt fájlokon és mentünk, akkor a teszt fájl új verziója található meg az Xcode-on belül is, nem kell újra importálni a teszt fájlt. Új grafikus teszt modell létrehozása esetén természetesen újra importálni kell a legenerált teszt fájlt az Xcode projektbe.

4.4. A 2D grafikus modell

A projekt létrehozása után a *model.bpmn* fájl szerkesztésével elkezdhetjük a teszt modell szerkesztését. A teszt készítéséhez az alábbi BPMN2 elemeket használhatjuk fel:

- **Test Step**: teszt lépés, amiben megadható az accessibilityLabel és a KIFAction
- **Expected Result**: elvárt eredmény ellenőrzés, amiben megadható az accessibilityLabel és a KIFAction
- **Inclusive Gateway**: elágazás kezdete, ahonnan több ágban mehet tovább a teszt
- **Exclusive Gateway**: elágazás vége, ahol több teszt ág egy ágban folytatódik

- **Sequence Flow**: a többi elem összeköttetésére és a teszt irányának megadására szolgáló elem
- **Start Event**: a tesztek kezdeti lépése, melyből csak egy szerepelhet a modellben, innen indul ki az összes teszt
- **End Event**: a tesztek utolsó lépése, melyből csak egy szerepelhet a modellben, ide érkezik be az összes teszt



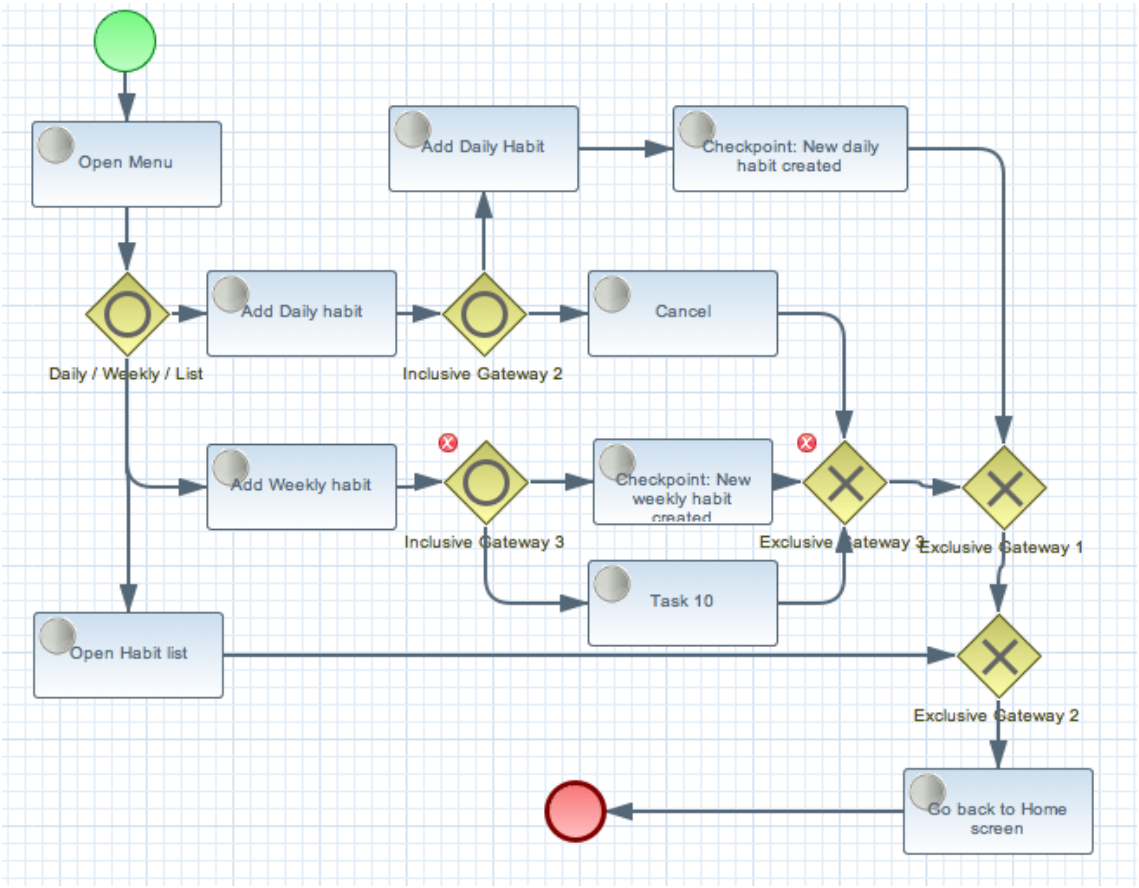
4.5. ábra. *Helyes teszt modell, ahol az elágazás kezdetből kiinduló teszt ágak a megfelelő elágazás végben érnek véget.*

A BPMN2 szerkesztő palettáján több BPMN2 elem is megtalálható és felhasználható, hibát nem fog írni a modell szerkesztő, viszont ezen elemek felhasználása esetén nem fog lefutni a kódgenerálás.

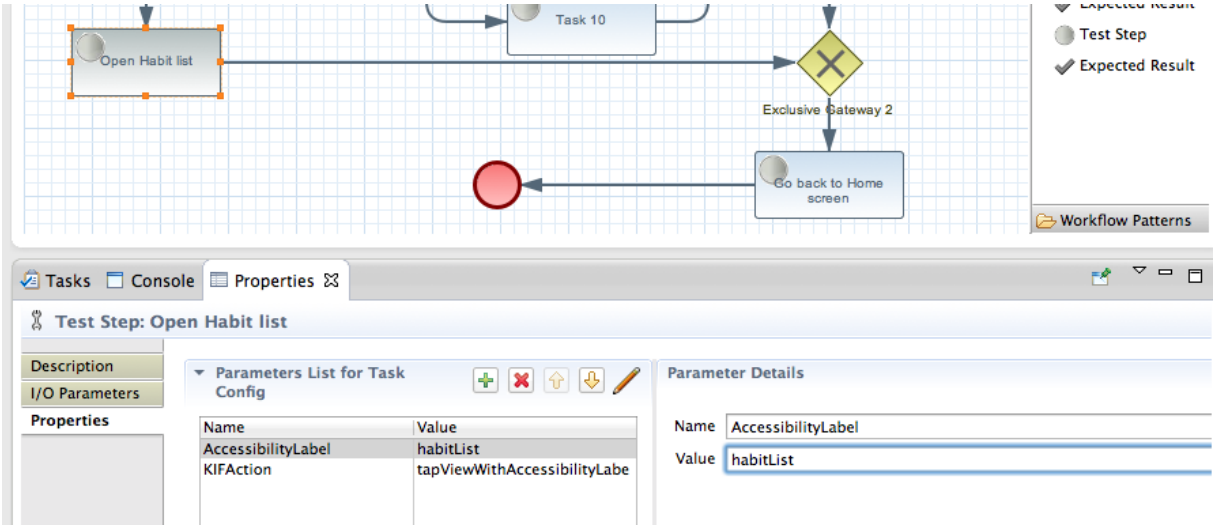
A 4.5. ábrán lévő teszt modell a Habits alkalmazás menüjének teszelését írja le. A zöld kör jelzi a tesztek kezdeti lépését, a piros kör a tesztek utolsó lépését, a szürke téglalap a teszt lépéseket és elvárt eredmény ellenőrzéséket, a citromsárga elem körrel a közepében az elágazás kezdetét, a citromsárga elem x-el a közepében pedig az elágazás végét jelöli. A modell helyes, mert az elágazás kezdetből kiinduló teszt ágak a megfelelő elágazás végben érnek véget, míg a 4.6. ábrán lévő teszt modell nem helyes (ezt piros x jelzi is az elágazás sarkában), mert a Cancel teszt lépés nem megfelelő elágazás vége elemben ér véget.

A teszteket a grafikus felületen össze tudjuk állítani, de a teszt lépések és elvárt eredmény ellenőrzések esetében további paramétereket is meg kell adnunk a tesztek megfelelő legenerálásához. Ezt a *Properties* Eclipse nézeten tudjuk megtenni[2], ha az adott teszt lépés vagy elvárt ellenőrzés elem ki van jelölve. A *Properties* nézeten a *Properties* fülön lehet az accessibilityLabel és KIFAction paramétereket megadni (4.7. ábra).

[2]Eclipse -> Window -> Show View -> Properties



4.6. ábra. *Hibás a teszt modell, mert a Cancel teszt lépés nem megfelelő elágazás vége elemben ér véget.*



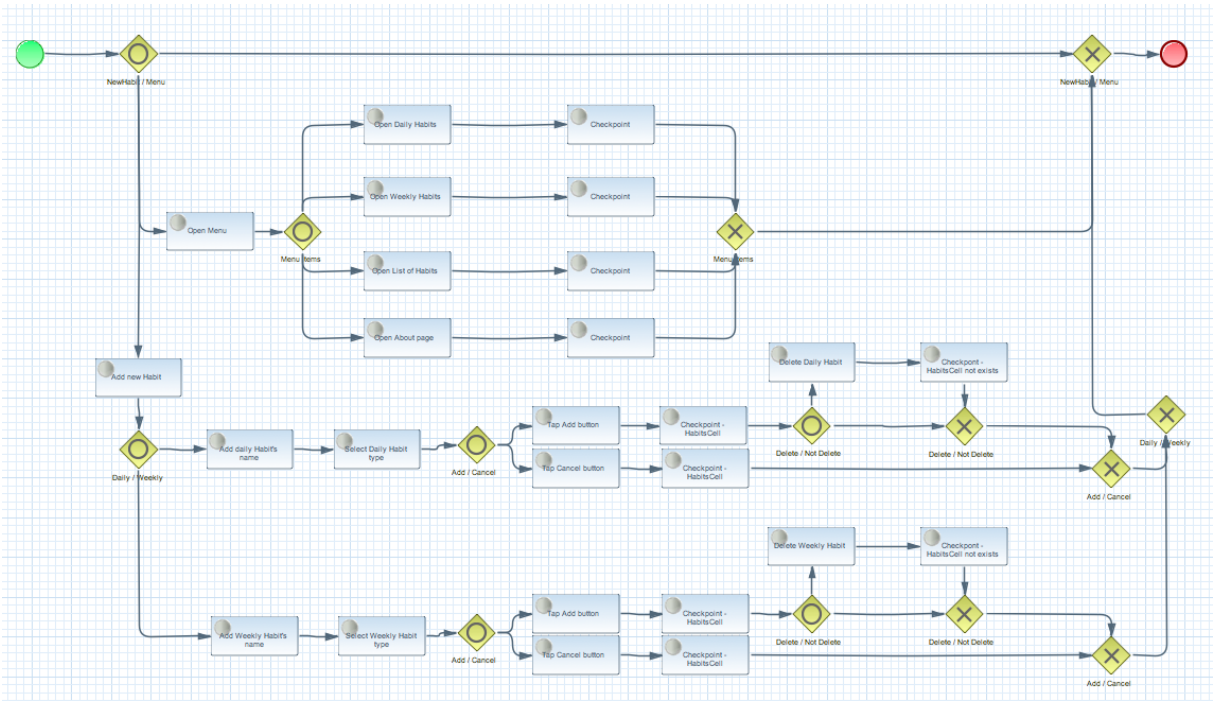
4.7. ábra. *A teszt lépés paramétereinek megadása a Properties fülön.*

4.5. Kódgenerálás

A grafikus szerkesztőben a modell mentése után automatikusan elkezdődik a kódgenerálás, a generált kód pedig megjelenik a projekt *src-gen* mappájában, illetve bemásolódik az Xcode teszt mappájába, ha a teszt mappa elérési útvonala meg van adva. Ha nincs megadva

teszt mappa, akkor csak az *src-gen* mappába másolódik. Utóbbi felhasználásának akkor lehet értelme, ha még nem akarjuk futtatni a teszteket, csak szerkesztjük a teszteket és kíváncsiak vagyunk a már legenerált forráskódra.

Az alábbi ábrán (4.8. ábra) látható modellből generált kód utolsó két tesztjét részletesen is bemutatom (4.1 forráskód). A kódból látszódik, hogy a teszt metódusok elnevezése nincs túlbonyolítva. Mivel az Xcode teszteknek *test* kulcsszóval kell kezdődniük, ezért a *test* szó után a generált teszt sorszámát írom, így áll össze az egyedi metódusnév minden teszt számára. Mivel 11 tesztesetet generáltunk a jelenlegi példában, ezért az utolsó két teszt metódusneve *test9* és *test10* lett.



4.8. ábra. *A Habits alkalmazás felhasználói felület tesztjére egy példa.*

A generált forráskódban megtalálhatóak a grafikus modellszerkesztőben megadott pontos accessibilityLabel és KIFAction értékek, viszont már Objective-C nyelven, teljes KIF tesztek formájában. Mivel a teszt lépések és az elvárt eredmény ellenőrzések a KIF keretrendszerben azonos lépésekként néznek ki, ezért minden elvárt eredmény ellenőrzés elé kommentben szerepel egy sor, mely kiemeli a teszt lépések közül az elvárt eredmény ellenőrzéseket.

A legenerált tesztek hibamentesek, kivéve ha az accessibilityLabel paraméter értékéből hiányzik az idézőjel. Ezt kiszűrő validáció nincs a szoftverben.

4.1. Forráskód. *Új napi és új heti szokást létrehozó majd azt teljesítő felhasználói felület tesztek generált forráskódja.*

```
- (void) test9
{
        [tester tapViewWithAccessibilityLabel:@"DailyHabitAddButton"];
        [tester enterText:@"daily habit"
                intoViewWithAccessibilityLabel:@"NewHabitNameTextfield"];
        [tester tapViewWithAccessibilityLabel:@"NewHabitNameFreqChanger1"];
        [tester tapViewWithAccessibilityLabel:@"NewHabitSaveButton"];

        // Expected result
        [tester waitForViewWithAccessibilityLabel:@"HabitCell"];

        [tester swipeViewWithAccessibilityLabel:@"HabitCell"
                inDirection:KIFSwipeDirectionLeft];

        // Expected result
        [tester waitForAbsenceOfViewWithAccessibilityLabel:@"HabitCell"];

}

- (void) test10
{
        [tester tapViewWithAccessibilityLabel:@"DailyHabitAddButton"];
        [tester enterText:@"weekly habit"
                intoViewWithAccessibilityLabel:@"NewHabitNameTextfield"];
        [tester tapViewWithAccessibilityLabel:@"NewHabitNameFreqChanger0"];
        [tester tapViewWithAccessibilityLabel:@"NewHabitSaveButton"];

        // Expected result
        [tester waitForViewWithAccessibilityLabel:@"HabitCell"];

        [tester swipeViewWithAccessibilityLabel:@"HabitCell"
                inDirection:KIFSwipeDirectionLeft];

        // Expected result
        [tester waitForAbsenceOfViewWithAccessibilityLabel:@"HabitCell"];

}
```

5. fejezet

Kiértékelés

Az előző fejezetekben bemutattam az általam készített szoftver tervezési és megvalósítási lépéseit, illetve bemutattam a használatát egy példa alkalmazáson keresztül. Az alkalmazás jelen állapotában használható, de természetesen sok továbbfejlesztési lehetőség rejlik a szoftverben, melynek részletes kifejtését és a szoftver használhatóságának elemzését tartalmazza az alábbi fejezet.

5.1. A szoftver használhatósága

A szoftver célközönsége tipikusan szoftverfejlesztők és olyan szoftvertesztelők, akik a projekt forráskódjához is hozzáférnek. Mivel az szoftverem csak Xcode fejlesztőeszközzel tud együttműködni, így csak olyanok használhatják akik Xcode-ot használnak, viszont ez elég magas százalék az iOS fejlesztők körében. Az Xcode mellett csak egy Eclipse-et kell elindítani, és használható is a szoftverem. Az Eclipse egy népszerű Java fejlesztőkörnyezet, emiatt sokaknak ismerős.

Az ITG első beállítása igényel kicsit több időt a fejlesztőtől, amíg elkészíti az ITG projektet és azt megfelelően beállítja, ezután új tesztek létrehozása csak az új tesztfájlt kell importálni az Xcode projektbe, ez minimális plusz munka.

A diagram szerkesztése a fejlesztők számára körülményesebb, mint programkód írása, viszont a grafikus modell könnyen átrendezhető és másolható. A grafikus modell validációja több hibát ki tud szűrni, mintha szimplán Objective-C KIF teszt kódokat írnánk.

Az iOS projekt változása esetén az ITG teszt modellt is könnyen lehet módosítani. A fejlesztés során tipikusan vagy új felhasználói elemek kerülnek a szoftverbe, vagy a meglévők felhasználói elemeket átalakítját át vagy szüntetik meg. Első esetben csak az *appSettings.application* fájlt kell feltölteni a fehasználói felület elemek accessibilityLabel értékeivel és új teszt lépéseket kell felvenni. Ha a felhasználói felület elemeket módosítjuk, de az accessibilityLabelét nem módosítjuk, akkor a teszt modell módosítás nélkül tovább használható. Törlés esetén az *appSettings.application* fájlból törölni kell a felhasználói elemek accessibilityLabel értékeit és a megfelelő teszt lépéseket törölni kell a grafikus modellből.

A fentiek alapján kijelenthetjük, hogy akkor produktív az ITG, ha a fejlesztő a fejlesztés közben folyamatosan használja az ITG-t, frissíti az accessibilityLabeleket és időközönként átalakítja a grafikus modellt. Szoftvertesztelő esetén pedig minden nagy release előtt érde-

mes a grafikus modellt részletesen átnézni és kijavítani, majd azt a regressziós teszt során minden új javítás után futtatni.

5.2. Továbbfejlesztési lehetősegek

A szoftverem funkcionalitása és komplexitása miatt sok irányba lehet továbbfejleszteni, ezek közül a legfontosabb és leghasznosabb lehetőségeket bemutatom.

5.2.1 Az iOS alkalmazás accessibilityLabeleinek beolvasása

A szoftver használatát nagyban segítené az, ha az *accessibilityLabeleket* nem kellene kézzel megadni, hanem az Xcode projektből meg tudnánk kapni. Erre több megoldás is létezik:

- Az Xcode storyboard fájlból az *accessibilityLabel* XML elemek beolvasása, mely csak akkor használható, ha az alkalmazás felhasználói felületét csak storyboard fájlok írják le,
- Az alkalmazás kódjának beparszolása, majd ebből az *accessibilityLabelek* kigyűjtése, mely csak akkor használható, ha az alkalmazás felhasználói felülete csak programkódból történik
- Az előző két módszer együttes használata.

Ha az *accessibilityLabeleket* megkaptuk, akkor egy további lépésként a teljes felhasználói felület hierarchiát meg tudjuk kapni. Erre a KIF keretrendszer tesztlépései biztosítanak megoldást. Érdemes lenne elkészíteni egy olyan KIF tesztlépésekből álló programot, mely megkapja az *accessibilityLabelek* listáját, és ebből minden képenyőképen végigpróbálná, hogy melyik felhasználói elem létezik, majd faszerűen bejárja a program a teljes felhasználói felületet. A felhasználói felület hierarciát így el tudná készíteni ez a program, ezt felhasználva pedig a grafikus teszt modell validációja még pontosabb lehetne.

5.2.2 A teszt fájlok másolása

A 3.7.1 alfejezetben bemutattam azt, hogy hogyan másolom át a legenerált teszt fájlokat az Eclipse projektből az Xcode projekt teszt mappájába. A problémát az okozza, hogy ha a teszt mappába a fájlt bemásolom, attól még az Xcode projektbe ez nem tartozik bele, ezáltal fordítás közben ezt a fájlt nem fordítja le az Xcode fordítója. Megemlítettem egy létező megoldást, az XcodeEditort, mely Objective-C nyelvű. Két megoldást is el tudok képzelni erre a problémára:

- Az XcodeEditor forráskódjának tanulmányozása, és a forráskód hozzáadás metódus megvalósítása Java nyelven,
- Eclipse-ből közvetlenül Objective-C kód futtatása.

A második megoldás egyszerűbbnek tűnik, ha létezik modul Eclipse-hez, mely tud Objective-C kódot futtatni. Az *objectiveclipse*[21] egy ehhez hasonló plugin, de a fej-

lesztését befejezték és nem található róla érdemi dokumentáció, tehát ennek a használata sok problémával járhat. Az első megoldás kivitelezhető, viszont ez sem egyértelmű, mert az XcodeEditor forráskódjának egy részét meg kell érteni hozzá, viszont lényegében csak egy XML fájl szerkesztését végzi a kérdéses metódus.

5.2.3 AccessibilityLabel paraméter validációja

A 4.5 alfejezetben egy példa alkalmazáson és egy példa teszt modellen keresztül mutattam be a kódgenerálást, majd a szekció végén említést tettem a paraméterek validációjának hiányáról.

Ha idézőjel szerepel bármelyik paraméterben, akkor az a KIF függvény paramétereként megadott sztring végét jelenti, és a további része az accessibilityLabel paraméternek pedig szintaktikai hibát okoz.

Meg kell tehát vizsgálni a paraméterek értékeit, és csak akkor kell elfogadni, ha nem tartalmaznak idézőjelet (vagy egyéb speciális, meg nem engedett karaktert), ellenkező esetben a grafikus felületen a teszt lépés vagy elvárt eredmény lépések mellett megjelnik a hibát jelző piros „x” jel.

5.2.4 Egyéb továbbfejlesztési lehetőségek

Az alábbi továbbfejlesztési lehetőségek is hasznosak lehetnek a szoftver kapcsán, ezért felsorolásszinten megemlítem őket:

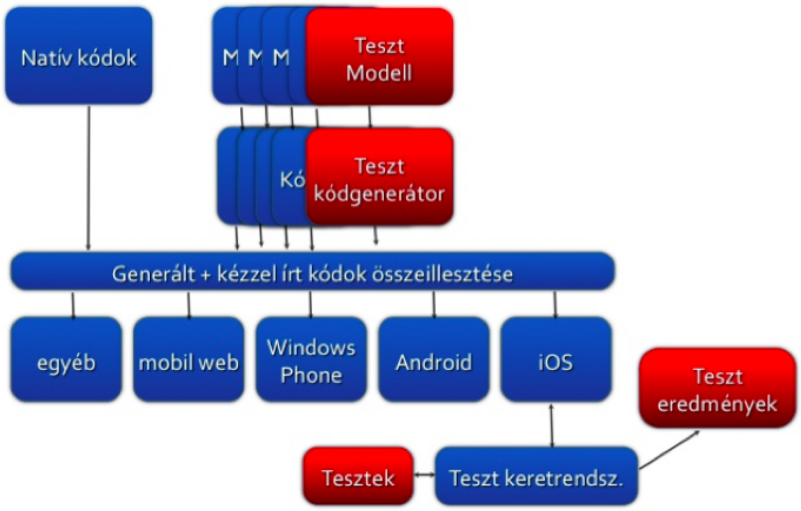
1. Egyablakos Java alkalmazás az Eclipse projekt helyett
2. A felhasználói felület felépítése alapján automatikus teszt ajánlás
3. Parametrizálható és testreszabható monkey-teszt[1] a teljes alkalmazásra

5.3. Kitekintés

A szakdolgozatom témáját az önálló laboratórium alatt elvégzett hasonló témájú szoftver adta. Önálló laboratóriumként egy egyetemi projektben vettem részt, mely egy multiplatform mobilalkalmazás fejlesztő keretrendszerről szól. Ezt első körben öt részre osztottuk fel:

- Perzisztencia
- Üzleti logika
- Felhasználói felület design
- Storyboard
- Szcenário teszt

[1] Olyan teszt, ahol a program megadott időközönként egy véletlen felhasználói elemen egy véletlen felhasználói eseményt hajt végre. A monkey-teszt célja alkalmazás összeomlások keresése.



5.1. ábra. *Több modulból álló multiplatform mobilalkalmazás fejlesztő keretrendszer nagyvonalú architektúrábrája, kiemelve a tesztelési részeket.*

A szakdolgozat témám már önálló szoftverként lett megtervezve, függetlenül a fentiekben ismertetett multiplatform keretrendszertől. Ennek ellenére az Xcode és Eclipse projektkapcsolatot viszonylag könnyedén le lehet cserélni más keretrendszeri modulra – például storyboard modulra –, így hasznos lehet a projekt további fázisaiban is a szoftverem.

6. fejezet

Összefoglalás

A szakdolgozatom során elkészítettem egy iOS felhasználói tesztek készítésére alkalmas szoftvert, mely hatékony validációs ellenőrzéseket is tartalmaz. Az eszköz együttműködik az Xcode fejlesztőeszközzel. A dolgozatomban kifejtem a szoftver tervezési és megvalósítási lépéseit, és egy példa alkalmazáson végigvezetve mutatom be a szoftver használatát. A szoftver elkészítése során sokat tanultam a modell alapú szoftverfejlesztésről, a kódgenerálásról és az Eclipse plug-in fejlesztésről.

A szakdolgozat kezdetekor egy olyan alkalmazás készítése volt a célom, mely nagyban segíti az iOS fejlesztők és tesztelők munkáját, akik eddig a KIF felhasználói teszteket programkódként írták. A tesztek gyorsan szerkeszthetőek és másolhatóak, így grafikus felületen valósítottam meg a tesztek leírását. A szoftver a félév eleji célkitűzéseimet teljesíti. A szoftver legfőbb képességei és szolgáltatásai az alábbiak:

- iOS alkalmazásokhoz felhasználói felület tesztek készítése grafikus modell segítségével
- Egy grafikus teszttel több teszt készítése az elágazás modell elem segítésével
- A grafikus tesztből KIF tesztek generálása
- A generált KIF tesztek átmásolása az Eclipse projektből az Xcode projektbe
- Grafikus modell validáció

Az Xcode és Eclipse projektek összekapcsolása után az ITG használata már nagyon egyszerű. Egy konkrét teszt elkészítése grafikus felületen tovább tart, mint ugyanazt a tesztet Objective-C nyelven megírni, de az elágazásoknak köszönhetően sok teszt megfogalmazása esetén már jóval hatékonyabb a grafikus modell, mint a forráskód írás.

Rövidítésjegyzék

1. API = Application Programming Interface, alkalmazásprogramozási felület
2. BPEL = Business Process Execution Language, üzleti folyamatok modellezésének végrehajtható nyelve
3. BPMN2 = Business Process Model and Notation, üzleti folyamat modell és jelölés
4. EMF = Eclipse Modeling Framework, Eclipse modellezési keretrendszer
5. ITG = iOS Test Generator, az általam elkészített projekt neve
6. KIF = Keep It Functional framework, Keep It Functional keretrendszer
7. SOA = Service-oriented architecture, szolgáltatás orientált architektúra
8. TDD = Test Driven Development, teszt vezérelt fejlesztés
9. UI = user interface, felhasználói felület

Irodalomjegyzék

[1] Gartner kimutatás 2013 harmadik negyedéves mobil eladásairól operációs rendszerekre bontva, http://www.gartner.com/newsroom/id/2623415, Gartner hivatalos statisztikai kimutatás. Publickáció dátuma: 2013 November 14.

[2] Telerik Test Studio, http://www.telerik.com/automated-testing-tools/ios-testing/ios-application-testing.aspx, a szoftver hivatalos promóciós és bemutató oldala.

[3] Apple UI Automation, https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html, fejlesztői dokumentáció. Utolsó módosítás dátuma: 2013. Szeptember 18.

[4] GorillaLogic MonkeyTalk, https://www.gorillalogic.com/monkeytalk, a szoftver hivatalos promóciós és bemutató oldala.

[5] KIF keretrendszer, https://github.com/kif-framework/KIF, projekt honlap.

[6] Calabash, https://github.com/calabash/calabash-ios, projekt honlap.

[7] Gherkin szakterület-specifikus nyelv, https://github.com/cucumber/gherkin, projekt honlap.

[8] iOS 7 sajtóképek, http://www.apple.com/ios/, az iOS7 hivatalos bemutatóoldala.

[9] iPhone 5S A7 chip telesítménynövekedés, http://www.apple.com/pr/library/2013/09/10Apple-Announces-iPhone-5s-The-Most-Forward-Thinking-Smartphone-in-the-World.html, hivatalos sajtóközlemény. Publickáció dátuma: 2013. szeptember 10.

[10] Az iOS 6.1 megjelenésekor több mint 800 000 alkalmazás volt megtalálható az AppStore-ban, http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html, hivatalos sajtóközlemény. Publickáció dátuma: 2013. január 28.

[11] Apple UI Automation szoftver, https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UsingtheAutomationInstrument/UsingtheAutomationInstrument.html, fejlesztői dokumentáció. Utolsó módosítás dátuma: 2013. szeptember 18.

[12] KIF 2.0, https://github.com/kif-framework/KIF/releases/tag/v2.0.0, a KIF keretrendszer 2.0-ás verziója. Utolsó módosítás dátuma: 2013. szeptember 8.

[13] AccessibilityLabel, https://developer.apple.com/library/ios/documentation/uikit/reference/UIAccessibility_Protocol/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008786-CH1-DontLinkElementID_1, fejlesztői dokumentáció. Utolsó módosítás dátuma: 2013. szeptember 18.

[14] Specta, https://github.com/specta/specta, projekt holnap.

[15] Multiplatform Eclipse, http://www.eclipse.org/koneki/images/ldt/carousel/screenCrossPlatform.png, kép.

[16] Eclipse plug-in architektúra ábra, http://www.programcreek.com/2011/09/eclipse-plug-in-architecture-extension-processing/, programozói blogbejegyzés, mely tartalmazza az architektúraábrát. Publickáció dátuma: 2011. szeptember.

[17] BPMN2 példa diagram, http://www.tuicool.com/articles/I3mENr, programozói blogbejegyzés, mely tartalmazza a diagramot. Publickáció dátuma: 2012. május 21.

[18] Ecore meta-metamodell, http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, fejlesztői dokumentáció.

[19] Xcode5 plugin template, https://github.com/kattrali/Xcode5-Plugin-Template, projekt honlap.

[20] XcodeEditor, https://github.com/jasperblues/XcodeEditor, projekt honlap.

[21] Objectiveclipse, https://code.google.com/p/objectiveclipse/, projekt honlap.

Linkek elérési időpontja: 2013.12.20.