

Tartalom

BEVEZETÉS.....	2
1. FELADAT ELEMZÉSE, PROBLÉMA BEMUTATÁSA	3
1.1. PROGRAMKÖRNYEZET, PROGRAMOZÁSI NYELV	3
1.1.1. A Windows Workflow Foundation szerepe	3
1.1.2. Programkörnyezet.....	5
1.2. FELADAT SPECIFIKÁCIÓ, ELEMZÉS	6
1.2.2. Az alkalmazással szemben támasztott követelmények	6
1.2.3. Tervezés során lehetséges problémák	7
2. SZOFTVER TERVEZÉSE	8
2.1. PROGRAM RÉSZEGYSÉGEI	8
2.1.1. A program blokkvázlata	8
2.1.2. Használati eset diagramok.....	9
2.1.3. Felhasználói felület tervezése.....	10
2.2. HASZNÁLT ADATSZERKEZETEK, OBJEKTUMOK, INTERFÉSZEK.....	12
2.2.1 Adatbázis tervezés.....	13
2.2.2. Osztály diagramok.....	15
2.3. PROGRAMSZERKEZET BONYOLULTSÁGA ÉS A FUTÁSI IDŐ.....	16
3. SZOFTVER KIALAKÍTÁSA	18
3.1. MAINFORM OSZTÁLY	18
3.1.1. Az osztály tagfüggvényei és eseményei.....	21
3.2. FORMHISTORY OSZTÁLY	23
3.3. FORMDB OSZTÁLY	24
3.4. IORDERINGSERVICE INTERFÉSZ.....	26
3.5. ORDERPROCESSINGWORKFLOW OSZTÁLY	26
3.5.1. Rendelés inicializálási szakasz implementálása	28
3.5.2. Rendelés kiszolgálása készletről szakasz implementálása.....	30
3.5.3. Rendelés kiszolgálása központból és lezáró szakasz implementálása	33
4. FELHASZNÁLÓI DOKUMENTÁCIÓ ÉS TESZTELÉS	37
4.1. FELHASZNÁLÓI DOKUMENTÁCIÓ.....	37
4.2. WORKFLOW TESZTELÉSE ÉS FUTÁSI FORGATÓKÖNYVEK	39
4.3. EREDMÉNYEK KIÉRTÉKELÉSE	41
4.4. PROGRAMFEJLESZTÉSI LEHETŐSÉGEK	42
4.5. A WORKFLOW MEGVALÓSÍTÁSA A .NET 4.0-VAL.....	43
ÖSSZEFOGLALÁS	49
IRODALOMJEGYZÉK.....	50

Bevezetés

Többrétegű alkalmazások fejlesztése során általában azt tapasztaljuk, hogy az alkalmazás üzleti logikájának implementálása a legkomplexebb, legkevésbé átlátható feladat.

Pár éve a Microsoft a .NET 3.0-ás kertsziszterrel bevezette a Windows Workflow Foundation API-t. Ez az API lehetővé teszi számunkra, hogy különböző alkalmazásokban használt üzleti folyamatokat modellezzünk, konfiguráljunk, monitorozzuk és futtassunk. Szoftverkészítés közben a WF által biztosított beépített megoldások előnye, hogy többé nem kell manuálisan összetett infrastruktúrákat fejlesztenünk a munkafolyamatokat engedélyező alkalmazásokhoz.

Szakedolgozatomban egy **rendelési folyamat szimulációs program** készítését fogom bemutatni. Az alkalmazásban használt üzleti logika ugyan nem egy valós folyamatot mutat be, viszont implementálható akár létező üzleti környezetbe is. A folyamat adminisztrációja Windows Form felületen történik.

A dolgozatban célok az alkalmazás elkészítésének részletes ismertetése és a WF alapú tervezés előnyeinek bemutatása. Ismertetem a Workflow Foundation és az egyéb felhasznált technológiák (SQL, ADO.NET) együttes használata során felmerült problémákat és javaslatokat teszek a problémák megoldására.

1. Feladat elemzése, probléma bemutatása

Adott egy technológia, amivel megvalósítható a program üzleti logikája. A legelső dolog ennek az üzleti feladatnak az elképzelése, mivel erre épül a program. Az egyszerűség kedvéért én egy termékrendelési folyamatot veszek alapul. Ezt kell majd leegyszerűsítve implementálni egy szoftveres környezetbe, ahol a WF fogja kiszolgálni a rendeléseket adatbázis műveletek és felhasználó felé történő kommunikáció segítségével.

Ebben a fejezetben szó lesz a probléma mibenlétéről, a program követelményeiről valamint a probléma megoldásához használt WF-ről. Emellett kitérek a használt módszerek alkalmazásának miértjeire is.

1.1. Programkörnyezet, programozási nyelv

A program tervezéséhez és futtatásához szükség van a **NET 3.5** vagy újabb környezetre, tervező programra, és egyéb hardveres és szoftveres követelményekre, ami ebben a részben van taglalva.

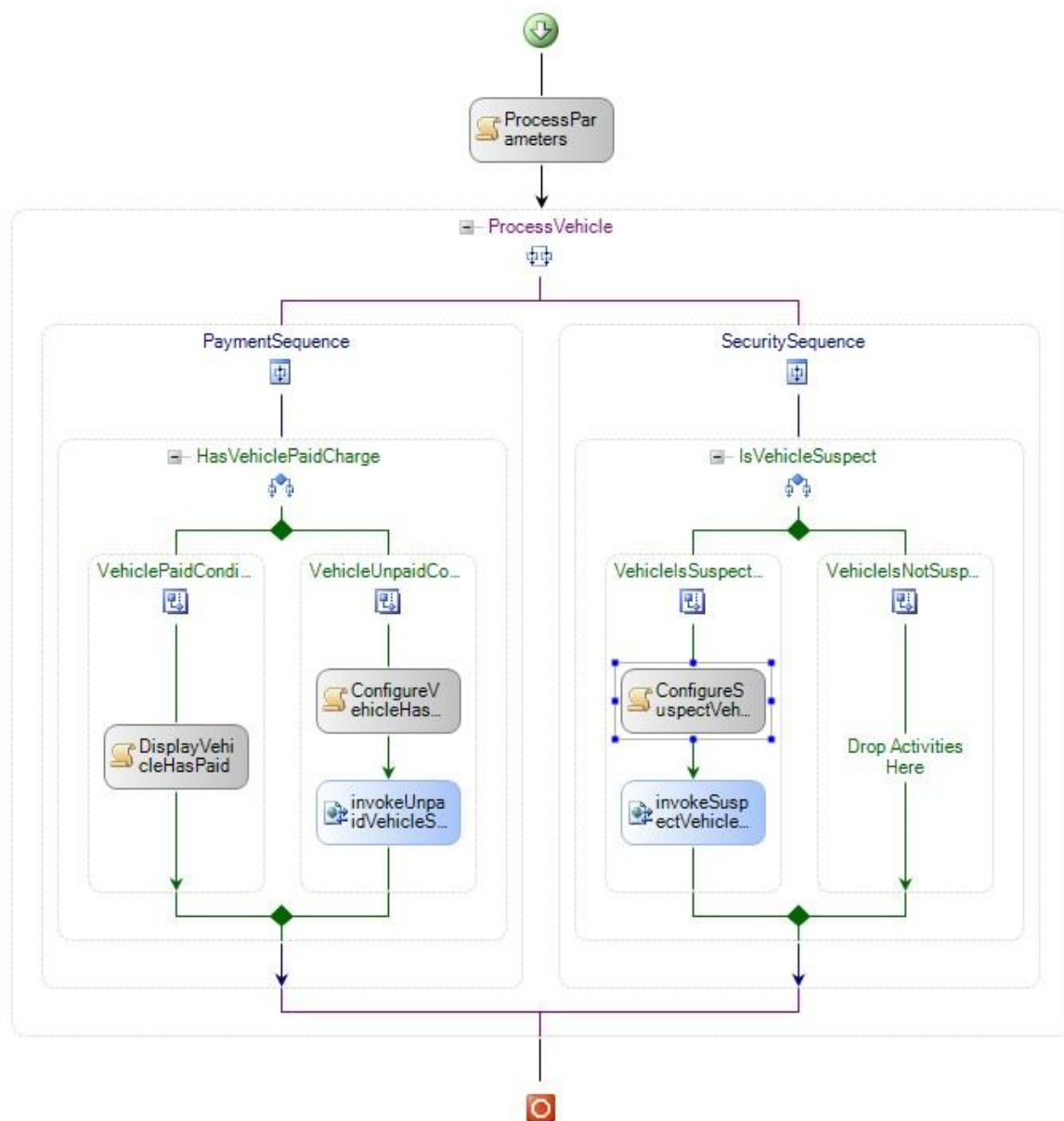
1.1.1. A Windows Workflow Foundation szerepe

„A WF a programozók számára deklaratív módon teszi lehetővé üzleti folyamatok tervezését, előre elkészített tevékenységek felhasználásával. Ahelyett, hogy az adott üzleti tevékenység és a szükséges infrastruktúra megjelenítéséhez szerelvények egyedi készletét építenénk fel, a Visual Studio 2010 WF- tervezőjével létrehozhatjuk saját üzleti folyamatunkat a tervezés ideje alatt. Így a WF lehetővé teszi egy üzleti folyamat vázának felépítését, amelyet aztán a kódban egészíthetünk ki.

Amikor a WF API -val programozunk, egységesen képviselhetjük az egész üzleti folyamatot, valamint a folyamatot definiáló kódot. Mivel egyetlen WF- dokumentummal megjeleníthetjük a folyamatot vezérlő kódot, valamint létrehozhatjuk a folyamat barátság-

gos vizuális ábrázolását, kizárható, hogy a dokumentumok szinkronizálása megbomoljon. A WF- dokumentum világosan bemutatja magát a folyamatot. ” [1]

Az 1. ábrán láthatjuk, hogy is néz ki egy általános üzleti folyamat a Workflow designer ablakban. A folyamat egyébként egy személygépkocsi nyilvántartó és ellenőrző rendszert ábrázol.



1. ábra. Szekvenciális Workflow példa²

¹ Forrás: Andrew Troelsen: A C# 2008 és a .NET 3.5. Szak kiadó, Budapest, 2009., 347. oldal

² Forrás: <http://consultingblogs.emc.com/howardvanrooijen/archive/2006/01/01/2530.aspx>

Az 1. ábra értelmezését fentről kell kezdeni. Miután a **ProcessParameters** blokk inicializálja a kezdőértékeket a végrehajtás két szálon folytatódik tovább. Az első szál a fizetéssel kapcsolatos, a második szál pedig biztonsági kérdésekkel kapcsolatos feladatokat hajt végre. Végrehajtás közben találkozunk if-else elágazásokkal, kód blokkokkal és külső websevice hívással is feltéve, hogy a megfelelő ágban történik a végrehajtás (baloldal fut le, ha a feltétel igaz).

Így néz ki egy rövid lefutású szekvenciális munkafolyamat. Az itt használt eszközök többsége segítségemre lesz a termékrendelési munkafolyamat elkészítésében.

A WF-nek két főbb típusa van a **szekvenciális** és az **állapotgép** (illetve a WF 4-ben flowchart és sequential). A szekvenciális típus egy kezdő és végponttal rendelkezik és sorban hajtja végre a feladatokat azaz Activityket, az állapotgép pedig ugrál az állapotok között.

A tervezéshez Visual Studio 2010 ULTIMATE változatát használok, ami alapesetben rendelkezik a WF API-val.

1.1.2. Programkörnyezet

A tervező valamint a program futtatásához legfontosabb a NET 3.5 keretrendszer, valamint a 4.5-ös fejezetben lévő verzióhoz már a NET 4.0. Ajánlott Windows 7 operációs rendszer. A szerkesztéshez a Microsoft Visual Studio 2010-es változatát használok és azon belül pedig a C# programozási nyelvet s Visual Basic-el szemben. A C# választása egyrészt a grafikus kezelő felület, másrészt a programozási nyelvben való jártasságom miatt indokolt. A Basic számomra kevésbé ismert és átlátható.

Időközben a Microsoft a NET 4.0-ás Framework-el teljesen újraírta a WF API-t, ami egy XML bázisú WPF-hez hasonló deklaratív kódolással (XAML) még hatékonyabban támogatja a munkafolyamatok vizuális tervezését. A baj csak az, hogy egy **xaml** fájlba nem tudunk C# szintaktikával dolgozni. Viszont cserébe az új tervezőfelület biztosítja, hogy ne kelljen közvetlen a XAML fájlba kódolni. Másik probléma hogy a WF 4-es API grafikus tervezőfelülete **csak** Visual Basic szintaktikájú kifejezéseket tud használni. Ezt a problémát csak a NET 4.5-ben fogják orvosolni.

Az adatbázis egy **mdb** fájl lesz, amit szerkeszthetek a Microsoft Access segítségével, a projekten belül pedig használhatom az OLEDB adatbázis szolgáltatót a lekérdezésekhez és adatbázis kezelési műveletekhez.

1.2. Feladat specifikáció, elemzés

Egy munkafolyamat alapú alkalmazást fogok készíteni, ami bemutatja a .NET 3.5-ös Workflow működését és hasznosságát. A hangsúly ebből kifolyólag a **host** és a **workflow** együttműködésén valamint a munkafolyamat alapú grafikus tervezés bemutatásán lesz. A WF 4 rengeteg újítást tartalmaz elődjeihez képest, de az új grafikus környezet és egyéb változtatások miatt nem könnyű az átállás. Ebből kifolyólag az alapprogram megvalósítása a 3.5-ös keretrendszerrel történik.

1.2.2. Az alkalmazással szemben támasztott követelmények

Az alkalmazással szemben támasztott követelmények:

- A Windows Workflow Foundation integrálása
- Egyszerű és átlátható felhasználói felület
- Megfelelő hiba és kivételkezelés
- Adatbázis szerkesztési lehetőség beépítése

Kezdsnek, ezeknek a követelményeknek kell majd eleget tenni. Ezek tulajdonképpen a minimális elvárások, amiken túl persze szeretnék egyéb funkciókat is beépíteni. A WF beépítése, mint legfőbb követelmény szerepel a listán, tehát a programban ennek kell kapni a legnagyobb szerepet, a felhasználói felület pedig a WF és a felhasználó közötti kommunikációhoz szükséges. Hiba és kivételkezelésre főleg az adatbázis műveletekhez van szükség. Az adatbázis lokális SQL adatbázissal működik majd, aminek szerkesztéséhez külön adminisztrációs/szerkesztési felületet késztek.

1.2.3. Tervezés során lehetséges problémák

A legnagyobb gond véleményem szerint az adatbázis kezelés és a kommunikáció implementálása lesz. A lekérdezéseket és egyéb adatbázis műveleteket ADO.NET-es módszerrel csinálom, így az SQL parancsok gépelésekor nem számíthatok az **IntelliSense** segítségére. A kommunikáció megvalósítása a WF és a Host alkalmazás között nem egyszerű, mivel külön szálon futnak, azonban igyekszem majd egyszerűbb, egyoldalú és nem kérés, válasz alapú kommunikációt használni.

Mivel a Workflow valósítja meg a program üzleti logikáját ezért törekedni kell arra, hogy a WF diagram minél átláthatóbb legyen. Tehát az összetettebb műveleteket, amik tartalmaznak ciklusokat, változókat és kifejezéseket (mit például az adatbázis műveletek) külön kell szedni egy saját kód blokkba.

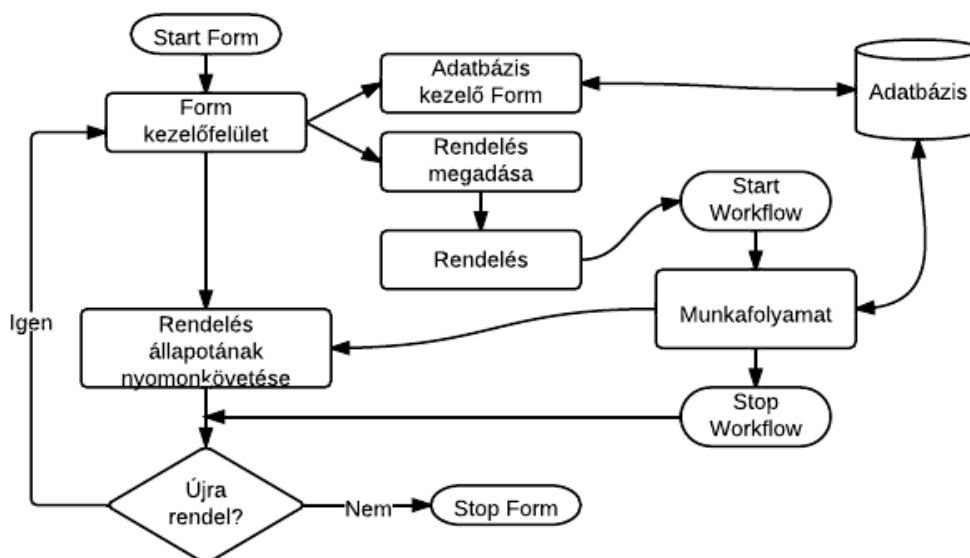
2. Szoftver tervezése

A tervezés során kitérek a program részegységeire és azok funkcióira, bemutatom a használt adatszerkezeteket, objektumokat és interfészeket és az azok közötti kapcsolatot UML diagramok segítségével. Végül elemzem a program szerkezetét és a futási időt. UML diagramok és blokkvázlatok készítéséhez a lucidchart.com³ valamint a yuml.me⁴ weboldalon található online szerkesztő alternatívákat használom.

2.1. Program részegységei

A részegységeket blokkvázlat, valamint használati-eset és osztály diagramok segítségével fogom bemutatni. A diagramokkal jól átláthatóvá válik a program viselkedése és szerkezete.

2.1.1. A program blokkvázlata



2. ábra. A program blokkvázlata

³ Forrás: <http://www.lucidchart.com/>

⁴ Forrás: <http://yuml.me/>

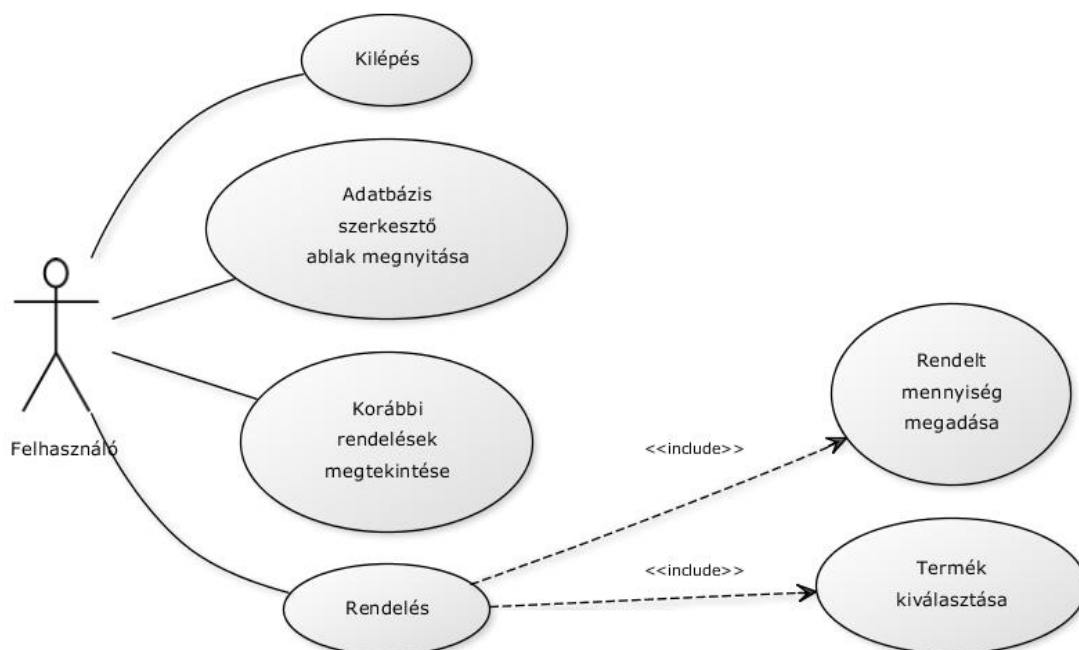
A leegyszerűsített vázlaton jól látható a program működése. A Host és a Workflow egymással, valamint az adatbázissal is kommunikál. Az adatbázis kezelő ablak nem feltétlenül szükséges a program működéséhez, de kényelmi szempontból mégis szükség van rá, hiszen tesztelésnél fontos tudnunk, hogy miből mennyi van készleten. Másik fontos funkciója ennek az ablaknak, hogy mutatja az információkat a leadott rendelésekről.

A program lényegi működése (forgatókönyv):

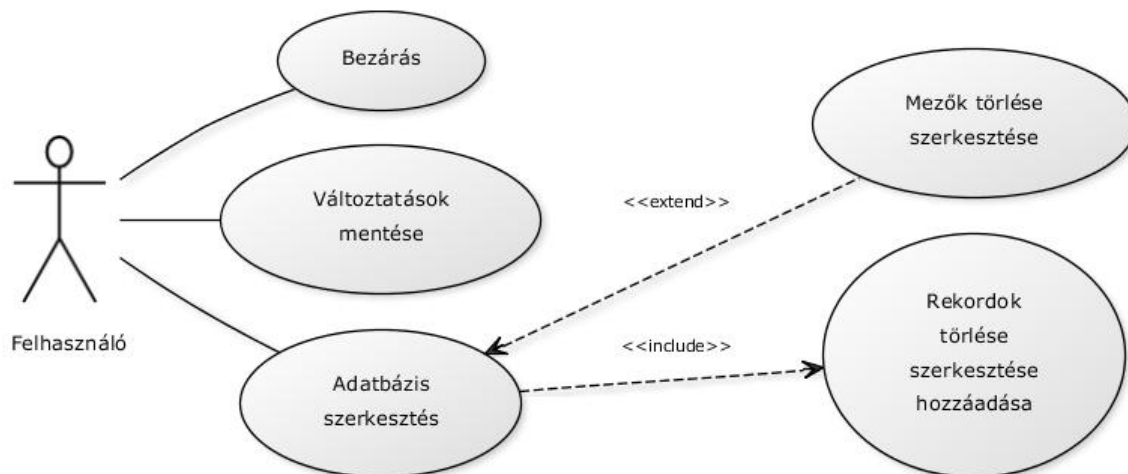
- Form elindítása és inicializálása
- Megadjuk a rendelést (terméknév, mennyiség)
- A rendelés elindítja a Workflow-ot átadva a rendelés paramétereit
- A Workflow végrehajtja a munkafolyamatokat, adatbázis műveleteket, miközben üzeneteket küld a Hostnak
- A Workflow befejezéséről tájékoztatja a Hostot
- Kilépés vagy újabb rendelés indítása

2.1.2. Használati eset diagramok

A használati eset diagramokkal a felhasználó által kiváltható eseményeket modellezem. Ezeket két részre bontom, a főablak, valamint az adatbázis szerkesztő ablak szerint.



3. ábra. Főablak használati esetei



4. ábra. Adatbázis kezelő ablak használati esetei

A 3. ábrán látszik, hogy a rendelés paramétereinek megadása nélkülözhetetlen. A 4. ábra az adatbázis szerkesztési lehetőségeket taglalja, amihez Valós üzleti környezetben adminisztrátori jogosultság szükséges.

2.1.3. Felhasználói felület tervezése

A GUI tervezés a Windows Form sablonok és eszközök adta lehetőségekre épül. Az elkészített program felülete három Form ablakból áll.

A főablak felülete űrlaphoz hasonló Form (5. ábra), ahol lehetőség van adatbevitelre, az adatbázis MenuStripButton-on keresztül korábbi rendelések lekérdezésre, az adatbázis szerkesztésére, a munkafolyamat elindítására valamint a WF-től kapott visszajelzések megtekintésére. Az ablak a következő gombokat tartalmazza:

- **Rendel:** Ez indítja a WF szerelvényt
- **Adatbázis/Adatbázis Szerkesztő:** Megnyit egy adatbázis szerkesztő ablakot
- **Adatbázis/Korábbi rendelések:** Megjeleníti a korábban leadott rendelések adatait tartalmazó dialógus ablakot
- **Bezárás:** Kilépés az alkalmazásból
- **Súgó, Névjegy:** Kiegészítő segítséget és hitelesítést nyújtó általános program funkciók.

5. ábra. Az alkalmazás főablaka

Az adatbázis szerkesztő (6. ábra) felületén keresztül láthatóak a relációs adatbázis táblái. Lehetőség van a mezők szerkesztésére, hozzáadására, törlésére. Az adatbázistáblák megjelenítése DataGridView eszköz segítségével történik.

Ezen a felületen csak két gomb található:

- **Változtatások mentése:** Véglegesíti a végrehajtott szerkesztési műveleteket
- **Bezár:** Elrejt az ablakot (a bezárás azért nem jó, mivel az ablak a programindicializáláskor jön létre, és nem célszerű törölni és újra kreálni az ablak osztályát)

Rendelések

	orderid	Terméknév	Mennyiség	Megkapva	Feldolgozási idő	WF_befejezve
	123	XBOX 360	1	22:20:32	3,37 nap közpon	22:20:35
▶	124	XBOX 360	1	22:21:03	készlethiány	22:21:03
*						

Készlet

	id	terméknév	mennyiség
▶	1	XBOX 360	0
	2	Playstation 3	0
	3	Sony PSP	1
	4	Nintendo Wii	5
*			

Központi raktár

	id	terméknév	mennyiség
▶	1	XBOX 360	0
	2	Playstation 3	8
	3	Sony PSP	10
	4	Nintendo Wii	10
*			

6. ábra. Adatbázis szerkesztő ablak

Az utolsó ablak egy dialógusablak, ami csupán a rendeléstörténet megjelenítésére szolgál. (7. ábra)



The screenshot shows a Windows form titled 'frmHistory'. It contains a table with the following data:

	orderid	Terméknév	Mennyiség	Megkapva	Feldolgozási_idő	WF_befejezve
▶	123	XBOX 360	1	22:20:32	3,37 nap központ...	22:20:35
	124	XBOX 360	1	22:21:03	készlethiány	22:21:03
	126	Playstation 3	2	11:53:21	3,419 nap közpo...	11:53:25

7. ábra. Korábbi rendelések ablak

2.2. Használt adatszerkezetek, objektumok, interfészek

A Visual Studioban számos szerelvény található, amik tartalmazzák a programozáshoz felhasznált, alapvetően szükséges elemeket. Ilyen elem például az adatbázisokhoz nélkülözhetetlen **System.Data**, ami az ADO.NET alapja és az adatbázis műveleteket hajtja végre. Mivel fontos szempont a hordozhatóság, az OLEDB adatbázis szolgáltató használatára van szükség (**System.Data.OleDb**). Ezzel lehetőségünk van hozzáférni egy Microsoft Access relációs adatbázishoz. A programtervezés kezdeti szakaszában még Microsoft SQL Szerveret használtam.

A Formok megjelenítéséhez is szükség van bizonyos szerelvényekre, amik az ablakok megjelenítéséhez kellenek (**System.Windows.Forms**). Erről a Visual Studio gondoskodik a projekt létrehozásakor.

A Szekvenciális Workflow alap szerelvényei a következők:

- System.Workflow.ComponentModel
- System.Workflow.Runtime
- System.Workflow.Activities

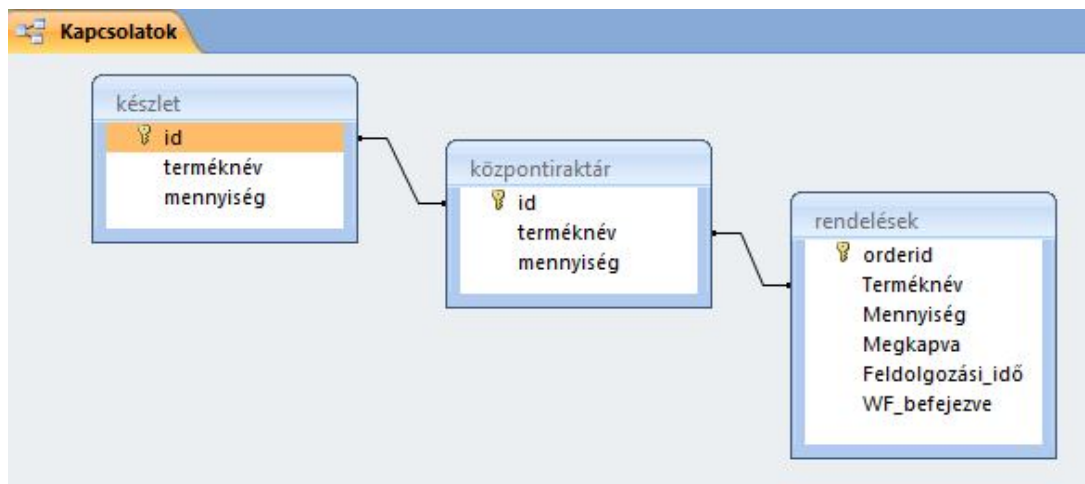
Az interfész az osztályok és objektumok közötti kommunikációban kap szerepet. A Workflow és a Formok közötti adatcserére három lehetséges módon kerülhet sor:

- Workflow példány létrehozásakor
- Futás közben interfészen keresztül call és handle external event activity segítségével
- Workflow példány futásának befejezésekor

Mindhárom megoldást felhasználtam a program tervezése során.

2.2.1 Adatbázis tervezés

Az adatbázis megtervezése a Microsoft Access 2007 programmal történik a program üzleti logikájában használt elvek figyelembevételével. Három táblára lesz szükség.



8. ábra. Használt adatbázis szerkezete

Az adatbázis (8. ábra) **mdb** formátumban van elmentve. A tábla- és oszlopnevek az ábrán magyarul szerepelnek, hogy átláthatóbbak legyenek.

A legfontosabb tábla a **készlet**, ami egy helyi raktárkészletet szimbolizál. Innen történik elsősorban a rendelt termék kiszolgálása a felhasználónak.

A **készlet** tábla a következő elemekből áll:

- **id**: termék azonosítója (int) (elsődleges kulcs)
- **terméknev**: készleten lévő termék neve (text)

- **mennyiség:** készleten lévő termék mennyisége (int)

A **központiraktár** a központi raktárt szimbolizálja, ahonnan akkor történik a kiszolgálás, ha a készletből már nem lehetséges. Erre akkor kerül sor, ha a felhasználó, aki az adott esetben a megrendelő, többet rendel az adott termékből, mint amennyi a készleten található. Persze logikusabb lenne időnként újrarendelni a központi raktárból, hogy ne legyen készlethiány, de mint említettem, nem célokom egy valós környezetben használt rendelési folyamat bemutatása.

A **központiraktár** tábla a következő elemekből áll:

- **id:** termék azonosítója (int) (elsődleges kulcs)
- **terméknév:** központi raktárban lévő termék neve (text)
- **mennyiség:** központi raktárban lévő termék mennyisége (int)

Az **rendelések** táblára a rendelések nyomon követhetősége miatt van szükség. Ez a tábla a rendelések azonosítására, és nem a megrendelők azonosítására hivatott. Mivel a programban a WF és a Form együttműködésének tesztelése a cél, így nem kell a felhasználókat külön azonosítani.

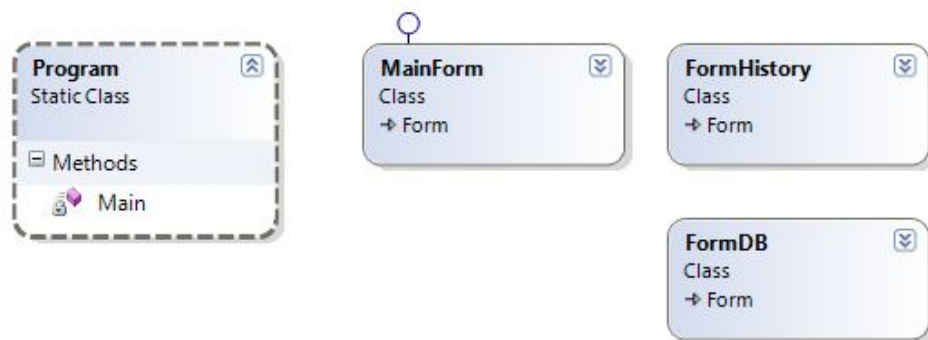
Az **rendelések** tábla a következő elemekből áll:

- **orderid:** rendelt tételek azonosítószáma (int)(elsődleges kulcs)
- **Terméknév:** rögzített termék neve (text)
- **Mennyiség:** rögzített termék mennyisége (int)
- **Megkapva:** rendelés megérkezésének és Workflow indításának ideje (datetime)
- **Feldolgozási_idő:** rendelés kiszolgálásának ideje, és módja (text)
- **WF_befejezve:** Workflow befejezésének ideje (datetime)

2.2.2. Osztály diagramok

Az osztálydiagramok a legalapvetőbb objektumorientált modellező eszközök, melyekkel a rendszert felépítő objektumokat és a közöttük lévő statikus kapcsolatokat írhatjuk le. Az osztálydiagramok terjedelme miatt, a feladatukat és csak fontosabb elemeiket fogom bemutatni.

A **Solution** azaz a program egésze, két projektből áll. Az első a **Host**, ami tartalmazza a GUI-t valamint kapcsolatot teremt a felhasználó és a gép között. A második egység/projekt az **OPWorkflow** (Order Processing Workflow), ami a háttérben dolgozik és végrehajtja az üzleti folyamatot. A következő ábrákon a **Host** és a **OPWorkflow** projekt fő osztályait láthatjuk. (9. és 10. ábra)



9. ábra. A Host osztálydiagramja



10. ábra. Az OPWorkflow osztálydiagramja

Ezek az osztálydiagramok csak a projekteket felépítő főbb osztályok láthatóak kapcsolatok nélkül, a harmadik fejezetben majd részletesebben kitérek a szerkezetükre. Az ábrák a VS tervezőjével készültek.

A **Host** látszólag több elemből épül fel, mint az **OPWorkflow**. De a valóságban, a **Program** statikus osztály csak a **MainForm** ablak létrehozásához használta. Valamint a három ablak osztálya ugyanúgy a **System.Windows.Form** ősosztályból öröklődik.

A Workflow felépítésben megtalálható az **OrderProcessingWorkflow** osztály, ami tartalmazza az üzleti folyamatban használt activityket, függvényeket, adattagokat. Az **OPWorkflow** projekten belül található még az interfész, ami mindkét projekthez hozzátartozik, de itt van definiálva. A **Host** úgy tud hozzáférni, hogy referenciának hozzáadjuk az **OPWorkflow** projektet.

2.3. Programszerkezet bonyolultsága és a futási idő

A program szerkezetileg nem mondható komplikáltnak. A szoftver blokkvázlatán látható az alapszerkezet (2. ábra), véleményem szerint jól érthető. A tervezés során olyan eszközöket használok, amik javítják a program átláthatóságát. A Form design felületének tervezésénél például külön designer.cs fájlba generálódik a kód, így a felesleges kód jól elkülönül. Ugyanez történik a Workflow tervezésnél.

A szerkezet egyszerűsítéséhez hozzájárul az is, hogy a program külön projektnek veszi a hostot és a workflowot, így elkülönül a két fő blokk. Az interfésznek is lehetne külön projektet szentelni, de mivel csak pár sorból áll, beágyazható a Workflow projektbe.

A program futási ideje is két részre bontható. A host futási ideje felhasználófüggő, tehát a program elindításától a bezárásáig tart. A futási idő másik részét a Workflow futása teszi ki. Ez két dologtól függ:

- egyrészt az adatbázis elérésének idejétől
- másrészt a workflow futási forgatókönyvétől, ami annak a függvényében változik, hogy milyen a raktárkészlet és a rendelt mennyiség viszonya.

Mivel én lokális adatbázist használok ezzel az elérési idő elenyésző. Ez probléma lehet, ha tekintetbe vesszük, hogy a készletről sokkal hamarabb ki lehetne szolgálni a terméket, mint a központi raktárból. Hogy ez a különbség érzékelhető legyen, olyan alternatívát alkalmazok, mint a késleltetés, amit a **Delay Workflow Activity** segítségével valósítok meg.

Nem beszéltem még a Form és a Workflow inicializálásának idejéről. Ez az idő főként a használt hardver és szoftverkörnyezettől függ, de nem nagy részét teszi ki a futási időnek. Ennek függvényében nem tartottam lényegesnek külön oszlopban rögzíteni a Workflow indításának idejét.

A Workflow tervezésénél próbálok minden fontosabb szakaszt külön részletezni, de az olyan szerkezetileg összefüggő és külső szemmel bonyolult kódhalmazokat, mit az adatbázis szerkesztési műveletek, külön kódban definiálok. Ezzel a program sokkal átláthatóbbá válik, és ez végső soron a Workflow szemléletű programtervezés lényege.

3. Szoftver kialakítása

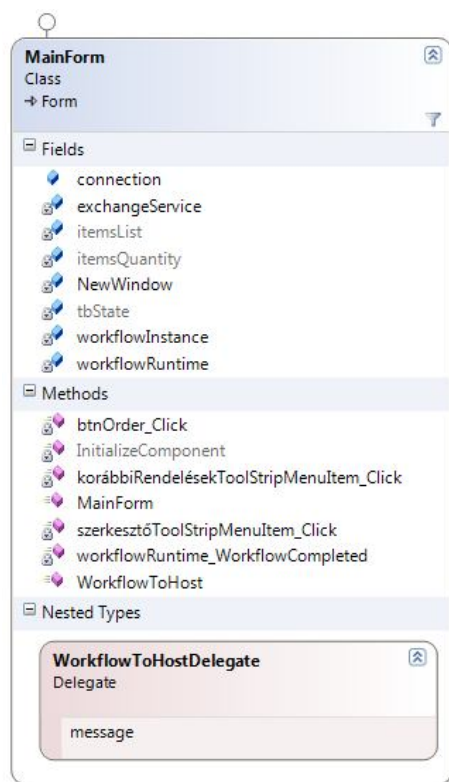
A program megvalósításhoz először létrehoztam a **WFOOrderandSQL** solution alatt a **Host** projektet, amihez a WindowsForms sablont használtam és a **MainForm**, **FormHistory** és **FormDB** főbb osztályokat tartalmazza (9. ábra). Miközben végeztem a felület alapjainak megvalósításával létrehoztam ugyanezen solution alatt az **OPWorkflow** projektet (10. ábra), aminek osztályai:

- **OrderProcessingWorkflow**
- **IOrderingService**

A következő alfejezetekben részletezem ezen osztályok szerkezetét és ahol szükséges ismertetem az adattagok és tagfüggvények funkcióját. Kitérek a tervezés során felmerült problémákra és az alkalmazott megoldások miértjeire. Az osztályok tervezési sorrendje nagyjából megegyezik a fejezetek sorrendével.

3.1. MainForm osztály

Feladata: Rendelés felvétele, átadása a Workflownak



11. ábra. MainForm osztálydiagram

A 11. ábrán jól láthatóak a **MainForm** osztály adattagjai, metódusai. A kevésbé fontos tagok (címkék és gombok) nincsenek megjelenítve. A halványan rajzolt neveket a program automatikusan hozza létre. A helyes működéshez az alapméretezett szerelvényeken kívül a következő **using** direktívákra van szükség:

```
using System.Workflow.Runtime;  
using System.Workflow.Runtime.Hosting;  
using System.Workflow.Activities;  
using OPWorkflow;  
using System.Data.OleDb;
```

Az első három elem a **workflowRuntime** és **workflowInstance** működéséhez szükséges. Az első magát a Workflow motort a második a Workflow példányt képviseli. A **System.data.OleDb** az adatbázis műveletek végrehajtásához, az **OPWorkflow** pedig az **OPWorkflow** projektben lévő interfész elérése miatt kell.

A **MainForm** osztályt a **Form** osztályon kívül az **IOrderingService** –ből is származtatni kell, hogy az interfész hozzáférhető legyen. Ezután meghatározom a főbb mezőket:

itemsList, itemsQuantity

Ezek az alapméretezett beviteli mezők nevei, amik azért vannak halványan, mert a Form designéren vannak létrehozva, olyan primitív beviteli eszközökkel, mint a *ComboBox* és *NumericUpDownControl*.

tbState

Ez szimbolizálja a **MainForm** felületén lévő többsoros *textboxot*, ami a rendelési információk megjelenítésére szolgál.

```
public OleDbConnection connection
```

Ez tartalmazza az adatbázishoz csatlakozáshoz szükséges kapcsolati stringet. Az adatbázis fájl esetünkben a projekt futtatási könyvtárában található, tehát a következő értéket adjuk neki: ("Provider=Microsoft.JET.OLEDB.4.0;data source=db.mdb")

```
private ExternalDataExchangeService exchangeService
```

Workflow és Host közti futásidejű kommunikációhoz szükséges

```
private delegate void WorkflowToHostDelegate(string message);
```

A Workflow felől érkező string típusú üzeneteket veszi át és továbbítja a WorkflowToHost metódushoz.

```
FormDB NewWindow = new FormDB();
```

Létrehoz egy új **FormDB** ablakot, amit később tetszés szerint megjeleníthetünk vagy elrejtethetünk. Ha befejeződik a workflow automatikusan meghívódik a **FormDB** frissítést végző függvénye, így nem kell manuálisan frissíteni az adatbázist.

Miután a konstruktor elvégzi az **InitializeComponent()** függvényt szükségünk van egy függvényre, ami feltölti az **itemsList** nevű comboboxot. A **MainForm** osztályon belül csak itt szükséges az adatbázishoz hozzáférni, hogy kiolvassuk a termékneveket és feltöltsük vele a tárolót. Mivel egy I/O műveletről van szó érdemes használni a **try\catch\finally** szerkezetet.

- try{...}: csatlakozás az adatbázishoz és az itemsList combobox feltöltése
- catch{...}: hibaüzenet csatlakozási vagy egyéb probléma esetén
- finally{..}: kapcsolat lezárása

A **try** blokkon belül használt módszert a programtervezés során többször is fel fogom használni adatbázis olvasási műveletekhez:

```
connection.Open();
OleDbCommand cmd = new OleDbCommand("SELECT terméknév FROM készlet",
connection);
OleDbDataReader rdr = cmd.ExecuteReader();
while (rdr.Read())
{
    this.itemsList.Items.Add(rdr[0].ToString());
}
this.itemsList.SelectedIndex = 0;
```

A combobox feltöltése után még mindig a konstruktoron belül létrehozzuk a **workflow runtime**-ot. Ez egy fontos lépés, mivel itt nem csak a workflow futási környezetét hozom létre, hanem hozzáadom az **exchangeService**-t ezzel vélik képessé a workflow a kommunikációra.

```
this.workflowRuntime = new WorkflowRuntime();
this.exchangeService = new ExternalDataExchangeService();
this.workflowRuntime.AddService(this.exchangeService);
this.exchangeService.AddService(this);
this.workflowRuntime.StartRuntime();
```

A konstruktorban létrehozok egy **workflowRuntime** eseményt, ami akkor hajtódik végre, ha a Workflow futása befejeződik:

```
workflowRuntime.WorkflowCompleted +=  
new EventHandler<WorkflowCompletedEventArgs>  
(workflowRuntime_WorkflowCompleted);
```

3.1.1. Az osztály tagfüggvényei és eseményei

```
// Megjeleníti a FormDB ablakot  
private void btnDB_Click(object sender, EventArgs e)  
{  
    NewWindow.Show();  
}  
  
// Megjeleníti a FormHistory ablakot  
private void btnHistory_Click(object sender, EventArgs e)  
{  
    FormHistory ujablak = new FormHistory();  
    ujablak.ShowDialog();  
}  
  
// ez a metódus fut le, ha a rendelés gombra kattintunk  
private void btnOrder_Click(object sender, EventArgs e)  
{  
    if (this.itemsQuantity.Value != 0)  
    {  
        btnOrder.Enabled = false;  
        this.szerkesztőToolStripMenuItem.Enabled = false;  
        this.korábbiRendelésekToolStripMenuItem.Enabled = false;  
        this.tbState.Clear();  
  
        Type type = typeof(OPWorkflow.OrderProcessingWorkflow);  
        Dictionary<string, object> properties = new Dictionary<string,  
        object>();  
        properties.Add("Termék", itemsList.SelectedItem.ToString());  
        properties.Add("Mennyiség", (int)itemsQuantity.Value);  
  
        workflowInstance = workflowRuntime.CreateWorkflow(type, properties);  
  
        workflowInstance.Start();  
    }  
    else  
    {  
        this.tbState.Text="Jelöld ki mennyit szeretnél rendelni!";  
    }  
}
```

A fenti metódus némi magyarázatra szorul, hiszen itt egy lényeges dolog történik a Workflow szempontjából. Az if részen belül a következő műveletek zajlanak le:

- Átmenetileg kikapcsol pár gombot, hogy rendelés közben ne tudjunk beavatkozni a Workflow működésébe.
- Létrehoz egy típust, ami megegyezik a használt Workflow-al
- Létrehoz egy dictionary tárolót, amit a felhasználó által választott termék és termék-mennyiség alapján feltölt. Majd ezt átadja a Workflow példánynak
- Létrehozza a Workflow egy példányát
- Elindítja a Workflow példányt!

```
// Ezt a függvényt írja ki a rendelés állapotát a tbState üzenetablakra
public void WorkflowToHost(string message)
{
    if (this.tbState.InvokeRequired)
    {
        this.tbState.Invoke(new
            WorkflowToHostDelegate(this.WorkflowToHost), message);
    }
    else
    {
        this.tbState.Text += message + "\r\n";
    }
}
```

A fenti **WorkflowToHost** függvény szerkezete nem a megszokott, mivel az **interfészen** keresztül történik a meghívása. Először fel kell élesztenünk a tbState üzenetablakot, hogy fogadja a másik szálról érkező üzenetet. A tényleges funkció az **else** ágon belül található.

```
//Workflow befejezésekor hajtódik végre
void workflowRuntime_WorkflowCompleted(object sender,
    WorkflowCompletedEventArgs e)
{
    if (this.tbState.InvokeRequired)
    {
        this.tbState.Invoke(new EventHandler<WorkflowCompletedEventArgs>
            (this.workflowRuntime_WorkflowCompleted), sender, e);
    }
    else
    {
        if(e.OutputParameters["Result"].ToString().Length!=0)
        {
            this.tbState.Text += e.OutputParameters["Result"].ToString();
        }
        this.tbState.Text += "\r\nMunkafolyamat befejezve";

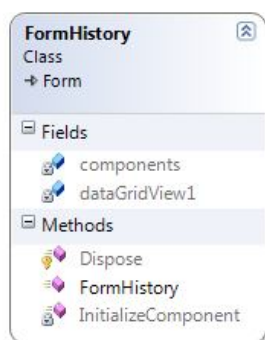
        NewWindow.UpdateTables();
        btnOrder.Enabled = true;
        this.szerkesztőToolStripMenuItem.Enabled = true;
        this.korábbiRendelésekToolStripMenuItem.Enabled = true;
    }
}
```

A fenti kód a **WorkflowCompleted** esemény hatására hajtódik végre a workflow befejezésekor. Az **if** ágban a Workflow visszaadja a vezérlést a Form-nak és végrehajtódik az else ágban lévő kód:

- Megvizsgálja, hogy a Workflow kimeneti „**Result**” paramétere üres-e, és ha nem akkor kiírja a képernyőre. Ez a paraméter egyébként arra használható, ha a Workflow nem tudja a rendelési mennyiséget kiszámítani, akkor rögzíti, hogy mennyi a maximálisan rendelhető mennyiség és ezt a mennyiséget szövegkörítéssel együtt beírja a **Result**-ba. Ezután tájékoztat a munkafolyamat befejezéséről.
- Meghívja a **FormDB UpdateTables()** függvényét, ami automatikusan frissíti majd a táblát
- Aktiválja a rendelés gomb megnyomásakor kikapcsolt gombokat, hogy újra rendelhessünk.

3.2. FormHistory osztály

Feladata: Korábbi rendelések megjelenítése



12. ábra. FormHistory osztálydiagram

Mivel ez az osztály az **Orders** tábla megjelenítésre szolgál, csak a **System.Data.OleDb** névtérrel kell használnunk a form elemeken kívül. A konstruktorban definiálom a megjelenítést végző scriptet:

```
{
...

//létrehozza a data adaptert
OleDbDataAdapter dAdapter = new OleDbDataAdapter("select * from rendelések", con);
//létrehozza az adattáblát a lekérdezési eredmények tárolására
DataTable dTable = new DataTable();
//feltölti az adattáblát
dAdapter.Fill(dTable);
//Kapcsolatot hoz létre az adattábla és a DataGridView szinkronizálásához
BindingSource bSource = new BindingSource();
```

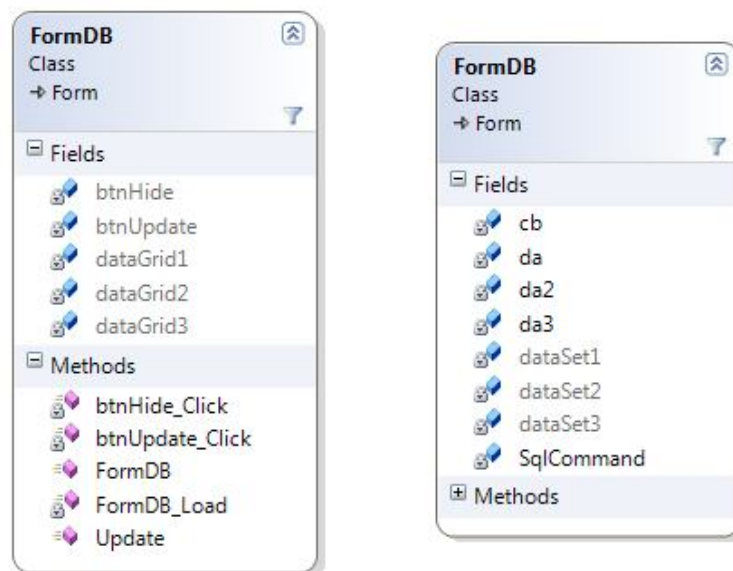
```
//szinkronizálás
bSource.DataSource = dTable;
//hozzárendelés a megjelenítő táblához
dataGridView1.DataSource = bSource;
}
```

A szerkesztési funkciókat érdemes kikapcsolni a **dataGridView properties** ablakában, mert úgyse lesz itt rá szükség, a FormDB osztályban viszont nélkülözhetetlen lesz:

- AllowUserToAddRows: false
- AllowUserToDeleteRows: false
- ReadOnly: true

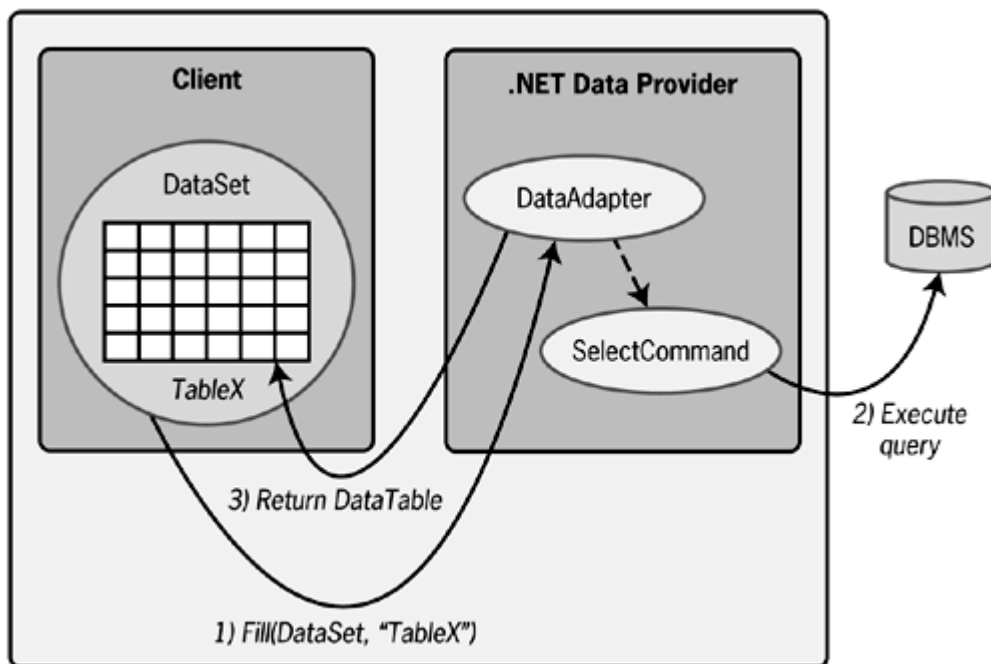
3.3. FormDB osztály

Feladata: Adatbázistáblák megjelenítése és szerkesztése



13. ábra. FormDB osztálydiagramja

A **FormDB** egy bizonyos értelemben vett **adminisztrációs felület** az adatbázishoz. A 13. ábrán ugyanazon **FormDB** osztály attribútumai és metódusai láthatóak, csak külön vannak szedve az ADO.NET-hez használt elemek. A három darab **dataGrid** elem szimbolizálja az adattáblákat, amihez majd a **dataSet**-ek lesznek rendelve. Ebben az osztályban a **FormHistory** osztályban használt módszertől némileg eltérő megoldással valósítom meg az adatbázis kapcsolatot, de a működés lényege a 14. ábrán jól látható.



14. ábra. Adatbázis elérése az ADO.NET segítségével⁵

Legelőször definiálni kell a használt adatbázis osztályokat:

```

private OleDbCommandBuilder cb;
private OleDbDataAdapter da;
private OleDbDataAdapter da2;
private OleDbDataAdapter da3;
private OleDbCommand SqlCommand;
  
```

Szükség van még három **DataSet** tárolóra, ezután jöhet a **DataGrid**-ek feltöltése. A táblák feltöltése a **FormDB** ablak **Load** eseményében történik meg a következőképp:

```

private void FormDB_Load(object sender, EventArgs e)
{
    ...

    string sql = "SELECT * FROM rendelések";
    SqlCommand = new OleDbCommand(sql, this.Connection);
    da = new OleDbDataAdapter();
    da.SelectCommand = SqlCommand;
    cb = new OleDbCommandBuilder(da);
    da.Fill(dataSet1, "rendelések");
    dataGrid1.SetDataBinding(dataSet1, "rendelések");

    ...
}
  
```

Ez annyiban különbözik a **FormHistory**-nál használt megoldástól, hogy **DataGrid**eken keresztül jeleníti meg a táblázatokat. Az **OleDbCommandBuilder**re a táblaszerkesztések helyes működése miatt van szükség.

⁵ Forrás: <http://flylib.com/books/en/2.78.1.50/1/>

A **btnUpdate** gomb eseménykezelője frissíti a táblák tartalmát az adott táblához rendelt **da** data adapteren keresztül.

```
private void btnUpdate_Click(object sender, EventArgs e)
{
    da.Update(dataSet1, "rendelések");
    da2.Update(dataSet2, "készlet");
    da3.Update(dataSet3, "központiraktár");
}
```

Az **UpdateTables** függvény a **MainForm**-ból hívódik meg a Workflow befejezésekor. Annyiban tér el a **FormDB** ablak **Load** eseményében használt kódtól hogy a táblák feltöltése előtt törli a **dataSet**eket: **dataSet1.Clear()**

3.4. *IOrderingService* Interfész

Feladata: Kommunikációhoz használt elemek definiálása

```
namespace OPWorkflow
{
    [ExternalDataExchange]
    public interface IOrderingService
    {
        void WorkflowToHost(string message);
    }
}
```

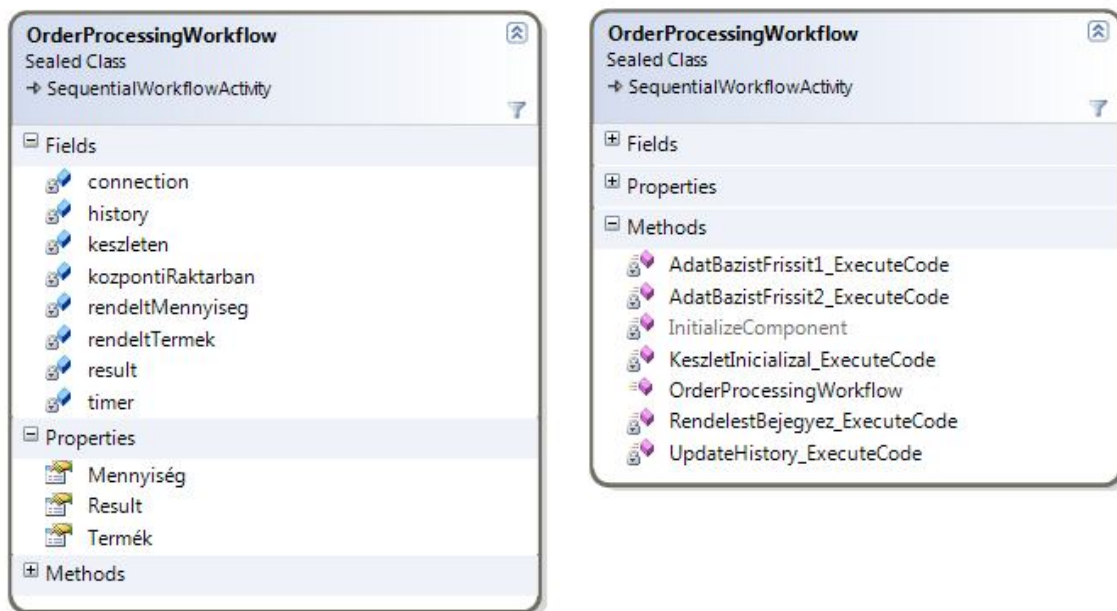
Az interfészen belül használt függvénynek az lesz a feladata, hogy a Workflow tervezés során használt **CallExternalMethod** activityk által generált üzeneteket továbbítsa a **MainForm** felé futásidőben.

3.5. *OrderProcessingWorkflow* osztály

Feladata: Rendelési folyamatok végrehajtása

Az **OPWorkflow** projekt létrehozása a **Sequential Workflow Library** sablonjával történik. A **MainForm**-ban elindított rendelés hatására a háttérben elindul a Workflow példány futása. Az **OrderProcessingWorkflow** a **SequentialWorkflowActivity** osztályból öröklődik, ami meghatározza a végrehajtási szerkezetet. A **StateMachine** szerkezetű végrehajtás azért nem indokolt, mivel a folyamat rövid lefutású és nem igényel felhasználói beavatkozást.

Az osztálydiagramon (15. ábra) nem tüntettem fel az **activityket**, abból kifolyólag, hogy ezek a program design felületén való elhelyezéskor automatikusan generálódnak és nem szeretném összekeverni őket a saját magam által létrehozott attribútumokkal.



15. ábra. OrderProcessingWorkflow osztálydiagramja
az Activityk ábrázolása nélkül

Az első dolog, miután elindult a workflow, hogy át kell venni a **dictionaryban** lévő értékeket és elmenteni őket az **rendeltTermek** és **rendeltMenniseg** változóba. Ez a művelet tulajdonságok segítségével történik. A tulajdonságok nevének pontosan meg kell egyeznie a **host**-ból átadott **dictionary**-ban lévő nevekkal:

```
public string Termék
{
    set
    {
        this.rendeltTermek = value;
    }
}
public int Mennyiség
{
    set
    {
        this.rendeltMennyiseg = value;
    }
}
```

Szükség van egy kimeneti **Result** tulajdonságra is, amit átadunk a **host**-nak a workflow befejezésekor:

```

public string Result
{
    get
    {
        return this.result;
    }
}

```

Az `OleDbConnection` `con` itt is a kapcsolati stringet jelképezi. Szükség lesz még a **készleten** és a **kozpontiRaktarban** változókra, amik a készleten és központi raktárban lévő termék mennyiségét fogják tárolni. A **timer** változó a Workflow inicializálásának idejét fogja tárolni ezred másodperces pontossággal, ami alapján az Inicializálási szakaszban lévő **UpdateHistory** számolni fogja a pontos végrehajtási időt. Végül egy **string history[]** tömb is kell majd, amiből majd a Workflow futása végén beírjuk a megfelelő adatokat az **rendelések** táblába. A tömb használata nélkül minden Workflow tevékenységnél újra meg kellene nyitni az adatbázist, és ez nem csak hatékonysági szempontból kedvezőtlen, hanem felesleges kód sorokat is generál.

A konstruktorba az inicializálás után megadom a **history[]** első elemét:

```
history[0] = DateTime.Now.ToLongTimeString();
```

Mivel a folyamat rövid lesz, elegendőnek tartom az aktuális idő beírását, ez kerül majd az adatbázis **Megkapva** mezőbe, ami a rendelés beérkezési idejét jelképezi.

A munkafolyamat megvalósítását több szakaszban fogom végrehajtani:

- 1. Rész: Inicializálás
- 2. Rész: Termék kiszolgálása készletről
- 3. Rész: Termék kiszolgálása raktárról és lezárás

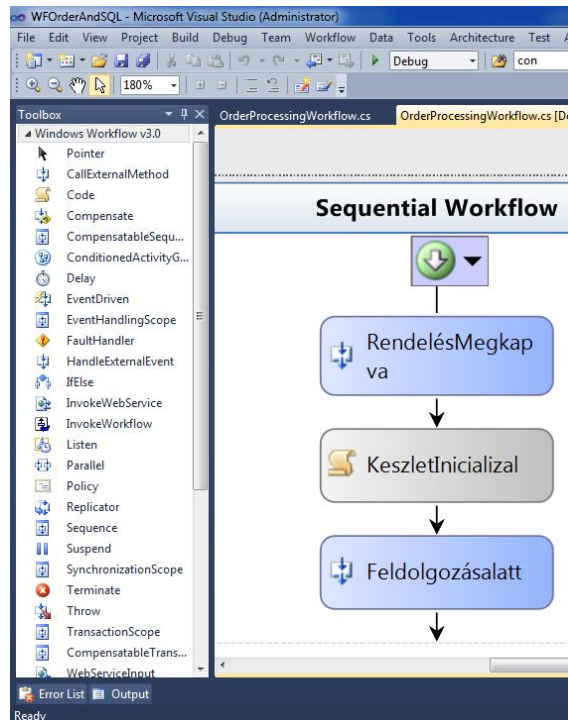
Logikusnak láttam a Workflow designtervezésnél magyar kifejezéseket használni, mivel ez nem csak a programozónak készül, tehát egy külső szemlélőknek is képesnek kell lenni a megértésére.

3.5.1. Rendelés inicializálási szakasz implementálása

Az inicializálási részben (16. ábra), miután elindult a Workflow, rögtön szükség lesz egy **CallExternalMethod** elemre, ami üzenetet küld a **MainForm**-nak a rendelési igény megérkezéséről. Ennek az elemnek 4 tulajdonságát kell módosítani a **Tulajdonságok** ablakban, amik név szerint:

- **Name:** activity neve, ami utal a küldött üzenetre
- **InterfaceType:** itt kell megadni az előre definiált és mindkét projekt alól elérhetővé tett **IOrderingService** interfészt
- **Message:** az interfészen keresztül küldött üzenet
- **MethodName:** interfészben definiált függvény: **WorkflowToHost()**

Ezzel elküldtük a „Rendelés Megkapva” üzenetet a felhasználónak, így az tudatában van annak, hogy megérkezett a rendelése a Workflow-hoz.



16. ábra. Inicializálási részben használt Activityk végrehajtási sorrendben és a workflow 3.0 toolbox elemei

Ezután jön a **KészletInicializal** kód, ami feltölti a **készleten** és a **kozpontiRaktarban** változókat az adott készletből. A kód szerkezete a **Host**-nál használt adatbázis olvasással megegyező, csak itt szükség van változók beágyazására is az SQL parancson belül:

```
try
{
    conection.Open();
    string query = @"SELECT mennyiség FROM készlet
    WHERE terméknev = '" + this.rendeltTermek + "'";
    ...
}
```

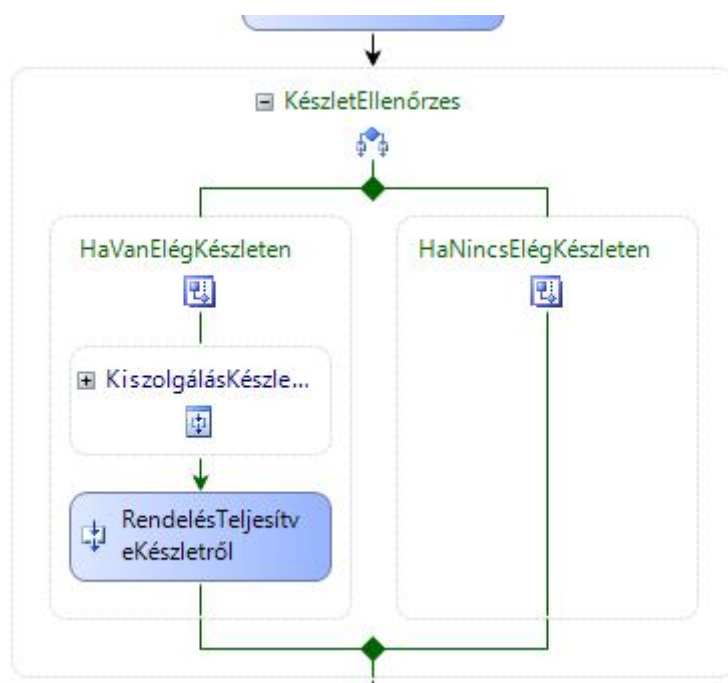
A fenti kódsor gépelésekor nem várt hibák történhetnek, például ha nem jó típusú aposztrófot használok, és hiába a **catch**-ba integrált hibakezelő, nem mindig egyszerű rájönni a

hiba forrására. Ilyenkor újra át kell olvasni a szöveget és kijavítani az esetleges elírásokat. A központi raktár lekérdezése is hasonló módon történik:

```
@"SELECT mennyiség FROM központiraktár  
WHERE terméknév = '" + this.rendeltTermek + "'";
```

A szakasz utolsó elemeként újabb státuszüzenetet küldünk, amiben informáljuk a felhasználót a feldolgozás megkezdéséről.

3.5.2. Rendelés kiszolgálása készletről szakasz implementálása



17. ábra. Kiszolgálás készletről szakasz

Ebben a szakaszban eljutunk a Workflow első elágazáshoz, ahol a **KészletEllenőrzés** nevű **if/else activity** segítségével eldöntjük, van-e elég darab termék a készleten a megrendelt termékből. A bal oldali ág, azaz a **HaVanElegKészleten branch activity** tulajdonságainál kell megadni a logikai feltételt, ami ha igaz, sorban végrehajtódnak a benne lévő Activityk. Ha hamis, akkor a jobb ágon lévő tevékenységek hajtódnak végre.

A feltételeket **Declarative Rule Condition** formában adom meg, mivel ehhez nem kell külön függvényt létrehozni az **OrderProcessingWorkflow** felületén, hanem a kifejezések egy külön **rules** kiterjesztésű állományban tárolódnak. (18. ábra)

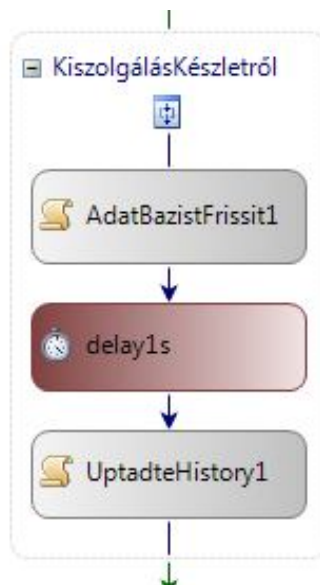
(Name)	HaVanElegRaktaron
Condition	Declarative Rule Condition
ConditionName	VanElegRaktaron
Expression	this.orderedQuantity <= this.realQuantity
Description	
Enabled	True

18. ábra. HaVanElegRaktaron Activity tulajdonságai

A feltétel összehasonlítja a rendelt mennyiséget a készleten lévő mennyiséggel, és ha van elég termék raktáron, végrehajtja a bal oldali ágban lévő Activityket.

Tegyük fel, hogy teljesült a feltételünk és a vezérlés átkerült a baloldali ágra. Nincs más dolgunk, mint elvégezni a termék **KiszolgálásKészletről** Sekvential Activityt (19. ábra) és informálni a felhasználót a rendelés kiszolgálásának befejezéséről. Magyarul:

- 1) frissíteni kell az adatbázist,
- 2) késleltetni kell a végrehajtást (hogy valósnak tűnjön a munkafolyamat),
- 3) a végrehajtás végeztével rögzíteni kell a **history[]**-ba a végrehajtási időt (adatbázis frissítés + időzítés)
- 4) és informálni a **Host**-ot a befejezésről.



19. ábra. KiszolgálásKészletről Sequence részletesebben

Ez a rész a WF eszköztár **Delay**, **CallExternalMethod** és az **AdatbazisFrissit** és **UpdateHistory** nevű **code activity**-vel történik. Az SQL parancsban megadjuk, hogy mi-

lyen új értékre frissítse a készleten lévő termékmennyiséget. Ezután átadjuk az OleDbCommand tagnak, aki végrehajtja a műveletet.

```
private void AdatBazistFrissit1_ExecuteCode(object sender, EventArgs e)
{
    try
    {
        connection.Open();

        string updateString =
            @"UPDATE készlet SET mennyiség = "
            + (this.készleten - this.rendeltMennyiség)
            + " WHERE terméknev = '" + this.rendeltTermek + "'";

        OleDbCommand cmd = new OleDbCommand(updateString);
        cmd.Connection = connection;
        cmd.ExecuteNonQuery();
    }
    ....
}
```

Az **UpdateHistory** kódját úgy írtam meg, hogy a Workflow tervezése során többször is fel lehessen használni. Attól függően, hogy miből és mennyit rendelt a felhasználó, három fajta érték kerülhet be a **history[]**-ba. Feltételnek ugyanazt adtam meg, mint a Workflow baloldali ágához. Az if ág csak annyit csinál, hogy beírja hány milisec telt el a **timer** változó inicializálása óta. Ehhez biztosan van kevésbé nyakatekert megoldás is, csak jobban kéne ismernem a c# dátum kezelési függvényeit vagy a Timer eszközt.

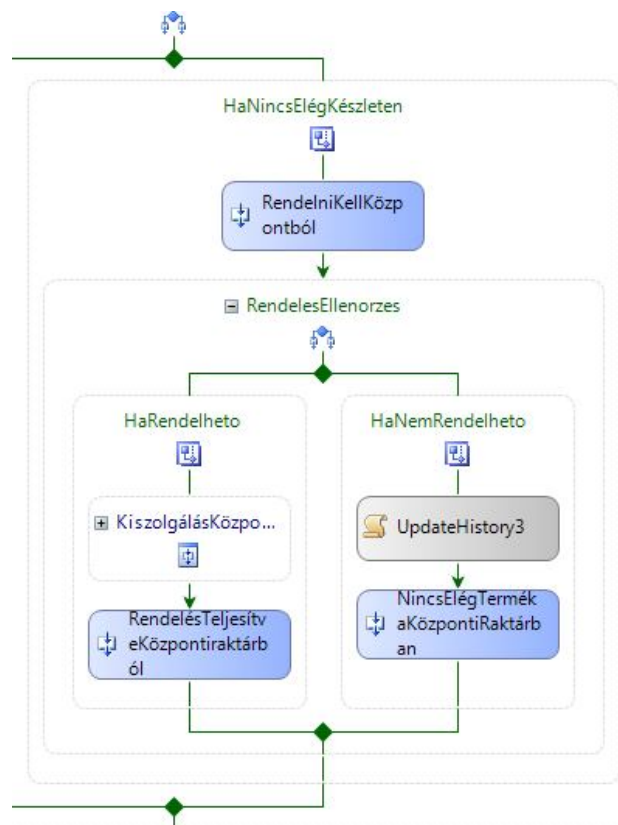
```
private void UpdateHistory_ExecuteCode(object sender, EventArgs e)
{
    if (this.rendeltMennyiség <= this.készleten)
    {
        // WF inicializálás-tól (timer) eltelt időt adja vissza másodpercben
        history[1] = (DateTime.Now.Minute * 60000 + DateTime.Now.Second *
            1000 + DateTime.Now.Millisecond - timer) / 1000 + " s készletről";
    }
    else if ()
    {
        ...
    }
    else
    {
        ...
    }
}
```

A valóságban itt több időnek kéne eltelnie, ezért az **AdatbazistFrissit** és az **UpdateHistory** közé berakok egy 1 másodperces időzítést. (19. ábra)

3.5.3. Rendelés kiszolgálása központból és lezáró szakasz implementálása

Az utolsó szakaszban nem csak az **if/else** jobb oldali ágát valósítom meg (20. ábra), hanem a munkafolyamat lezáró activityket is (21. ábra). Az első dolgunk, hogy egy **CallExternalEvent** activityvel tájékoztassuk a felhasználót arról, hogy nincs elég termék a készleten, ezért a központi raktárból kell teljesíteni a rendelést. Ezután jöhet az ellenőrzés, ami kideríti van-e elég termék a raktárakban:

keszleten + kozpontiRaktarban >= rendeltMennyiseg



20. ábra. KészletEllenőrzés hamis ága

Ha van elég termék a raktárakban, akkor végrehajtjuk a rendelést, azaz az **KiszolgálásKözpontból SequentialActivity**-t, ami az előző szakaszban használt háromlépéses módszerből áll:

```

private void AdatBazistFrissit2_ExecuteCode(object sender, EventArgs e)
{
    connection.Open();
    string updateString1 =
        @"UPDATE készlet SET mennyiség = 0
        WHERE terméknév = '" + this.rendeltTermek + "'";
    . . .
    string updateString2 =
        @"UPDATE központiraktár SET mennyiség = "
        + (this.kozpontiRaktarban - (this.rendeltMennyiseg - this.keszleten))
        + " WHERE terméknév = '" + this.rendeltTermek + "'";
    . . .
}

```

A rendelés szerkezetileg összetettebb, mivel mind a készletről, mind a központi raktár-ból le kell vonni az adott termékmennyiséget. A kérdés csak az, hogy honnan szolgáljuk ki a terméket elsősorban. A fenti példa szerint először a készletről, aztán a többi a központi raktárból. Ideális megoldás lenne, ha egy külön Workflow segítségével gondoskodnánk az újrendelésekről, amennyiben a készlet egy bizonyos határérték alá kerül. Ez azonban már inkább logisztikai, mint programozói kérdés.

Az időzítés, ami tulajdonképp a Workflow szál fagyasztása, ezúttal három másodperc, hiszen a központból való kiszolgáláshoz több idő kell. A **history[]** pedig a kiszolgálási idő mellett tájékoztat, hogy a kiszolgálás a központból történt.

```

private void UpdateHistory_ExecuteCode(object sender, EventArgs e)
{
    if (this.rendeltMennyiseg <= this.keszleten)
    {
        . . .
    }
    else if (this.keszleten + this.kozpontiRaktarban >= this.rendeltMennyiseg)
    {
        history[1] = (DateTime.Now.Minute * 60000 + DateTime.Now.Second *
            1000 + DateTime.Now.Millisecond - timer) / 1000 + " s központból";
    }
    else
    {
        . . .
    }
}

```

A **HaNemRendelhető** ágban lévő activityk futására akkor kerül sor, ha a rendelési mennyiség nagyobb, mint amennyit a készletről és a raktárból ki tudnánk szolgálni. Ekkor nincs szükség semmilyen adatbázis műveletre, vagy késleltetésre, csupán a felhasználó

tájékoztatására a **history[]** feltöltésére valamint a maximálisan rendelhető termékek számának beírásába a **result** változóba:

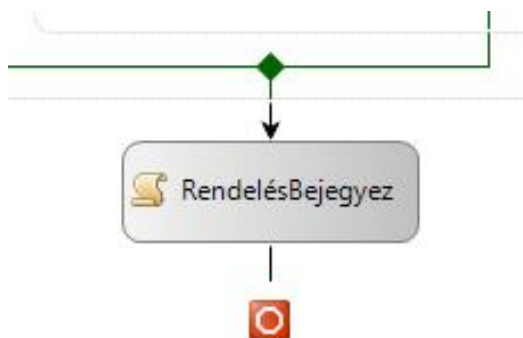
```
history[1] = "készlethiány";
this.result = "Maximum " + (keszleten + kozpontiRaktarban) + " db rendelhető!";
```

Az befejező szakasz végrehajtására (21. ábra), mindenképpen sor kerül, attól függetlenül, hogy melyik végrehajtási útvonalat választjuk. A **RendelestBejegyez** beírja a **history[]**-ba a munkafolyamat befejezésének idejét, ezután betölti a tömböt az adatbázisba.

```
{
    connection.Open();

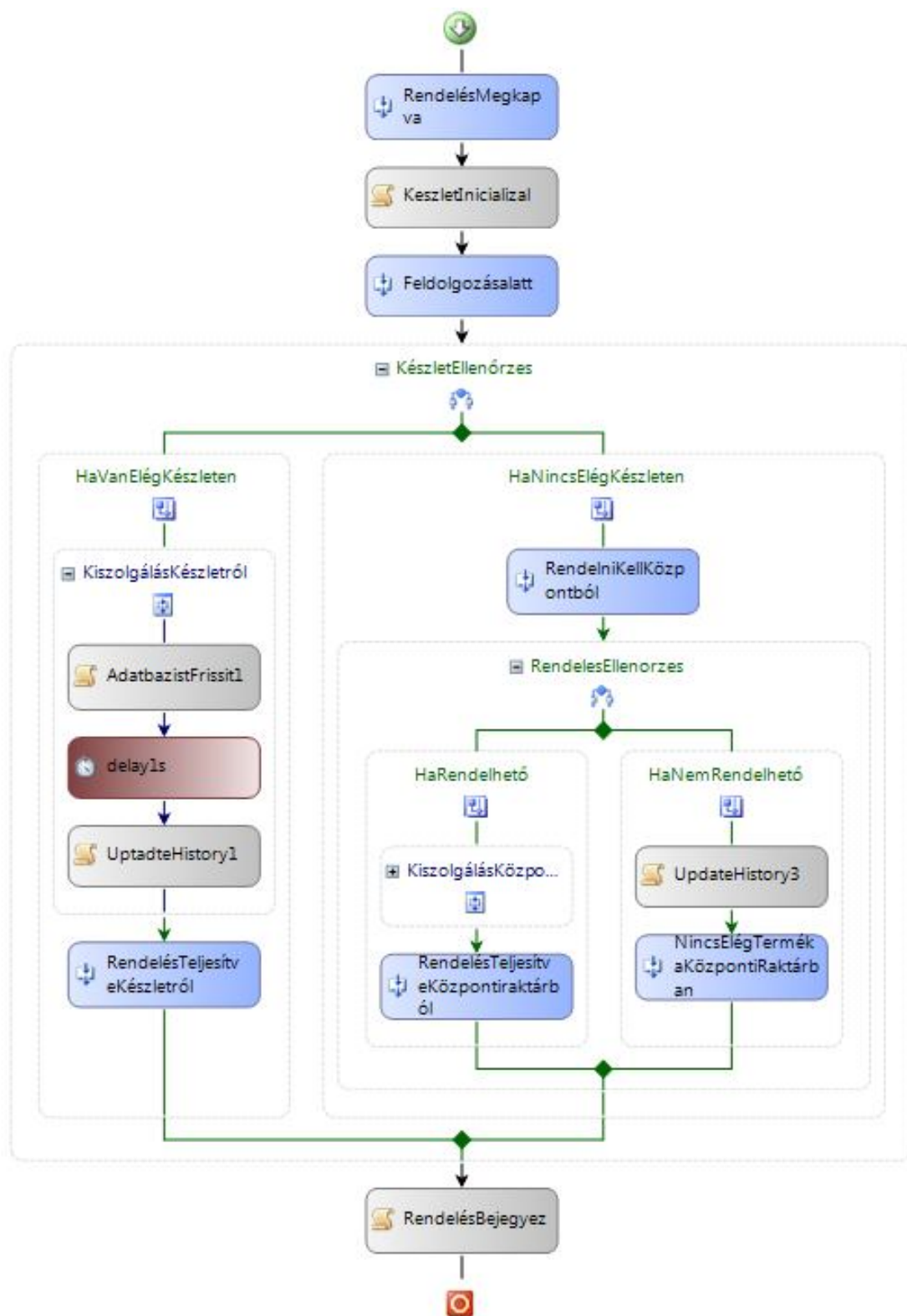
    string insertString = @"
INSERT INTO rendelések
(terméknév, mennyiség, megkapva, Feldolgozási_idő, WF_befejezve)
values ('" + this.rendeltTermek + "', '" + this.rendeltMennyiség + "', '"
+ history[0] + "', '" + history[1] + "', '" + DateTime.Now.ToLongTimeString() +
"')";

    OleDbCommand cmd = new OleDbCommand(insertString);
    cmd.Connection = connection;
    cmd.ExecuteNonQuery();
}
```



21. ábra. Befejező Szakasz

Végül a felhasználó értesítést kap a feladat befejezéséről és befejeződik a munkafolyamat. Továbbá a **MainForm** értesül a feladat befejezéséről, frissíti a **FormDB** ablakot és elérhetővé válik a rendelés gomb. A kész munkafolyamat a 21. ábrán látható.



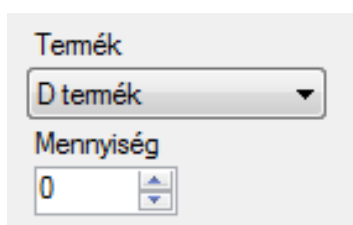
22. ábra. Az elkészült Workflow

4. Felhasználói dokumentáció és tesztelés

4.1. Felhasználói dokumentáció

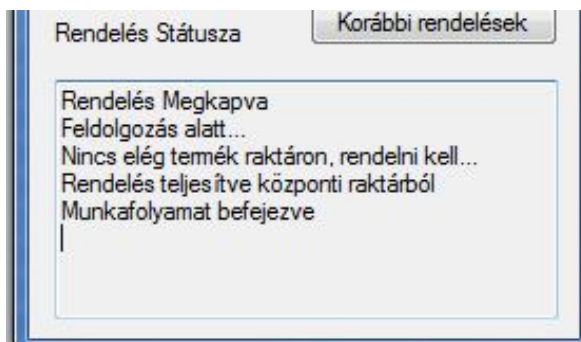
A programot, Windows platformon az **host.exe**-vel lehet elindítani, a **db.mdb** az adatbázist az **OPWorkflow.dll** pedig a Workflow szerelvényt szimbolizálja. Külön telepítést nem igényel. A továbbiakban bemutatásra kerül, a program használata.

A program indítása után megjelenik a főablak. Itt kell kiválasztani a terméket, termék-mennyiséget. A példában 4 féle termék közül választhatunk.



23. ábra. Beviteli mezők

Ha nem adunk meg rendelési mennyiséget rendelés gomb megnyomása előtt, a program nem enged tovább. A rendelés kiszolgálása alatt értesítéseket kapunk a feldolgozási fázisokról.



24. ábra. Státuszüzenetek

A Korábbi rendelések megjelenítésére kattintva megtekinthetjük az **Rendelések** táblát, ahol tájékozódhatunk, hogy melyik termékből, mikor és mennyit rendeltek, valamint hogy sikerült-e a termék kiszolgálása, és ha igen honnan.

frmHistory

	orderid	Terméknév	Mennyiség	Megkapva	Feldolgozási_idő	WF_befejezve
▶	133	Playstation 3	1	12:51:43	1,271 s készletről	12:51:45
	134	Playstation 3	15	12:52:12	3,163 s központból	12:52:15
	135	Playstation 3	15	12:52:35	készlethiány	12:52:36
	136	XBOX 360	1	16:00:47	2,498 s készletről	16:00:49
	137	Nintendo Wii	3	18:45:55	1,388 s készletről	18:45:56

25. ábra. Korábbi rendeléseket megjelenítő ablak

A főablakból, megnyitható az Adatbázis szerkesztő ablak (26. ábra), ami teszteléskor hasznos, mivel lehetőségünk van:

- felmérni a készlet és a központi raktár állapotát
- mezőket módosítani
- mezőket törölni
- új rekordokat felvenni
- rekordokat törölni

FormDB

Rendelések Bezár

	orderid	Terméknév	Mennyiség	Megkapva	Feldolgozási_idő	WF_befejez
	133	Playstation 3	1	12:51:43	1,271 s készletről	12:51:45
▶	134	Playstation 3	15	12:52:12	3,163 s központ	12:52:15
	135	Playstation 3	15	12:52:35	készlethiány	12:52:36
	136	XBOX 360	1	16:00:47	2,498 s készletről	16:00:49
	137	Nintendo Wii	3	18:45:55	1,388 s készletről	18:45:56

Készlet

	id	terméknév	mennyiség
▶	1	XBOX 360	9
	2	Playstation 3	0
	3	Sony PSP	10
	4	Nintendo Wii	7
*			

Központi raktár

	id	terméknév	mennyiség
▶	1	XBOX 360	10
	2	Playstation 3	4
	3	Sony PSP	10
	4	Nintendo Wii	10
*			

Változtatások mentése

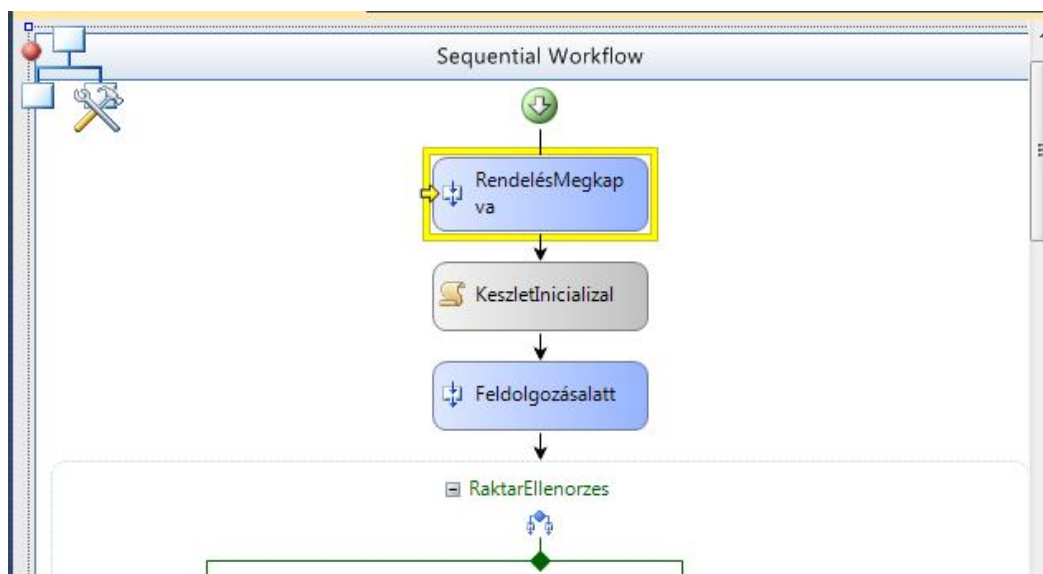
26. ábra. Adatbázis szerkesztő ablak

Az elvégzett módosításokat a változtatások mentése gombbal lehet véglegesíteni. A Bezár gombbal pedig eltüntethetjük az ablakot. Mivel ez egy különálló ablak, nincs szükség a bezárására, ahhoz hogy a **MainForm** ablakot használhassuk. A Workflow befejezése után az ablak automatikusan frissül.

4.2. Workflow tesztelése és futási forgatókönyvek

Ahhoz, hogy a Workflow-ot tesztelni tudjuk, el kell helyezni egy töréspontot a design felületén. Ezen kívül az **OPWorkflow** projektet kell beállítani kezdőprojektnek. Ezáltal lehetőség nyílik nem csak, a forráskód debuggolására, hanem a design felületen keresztül végigkövethetjük a munkafolyamatot is.

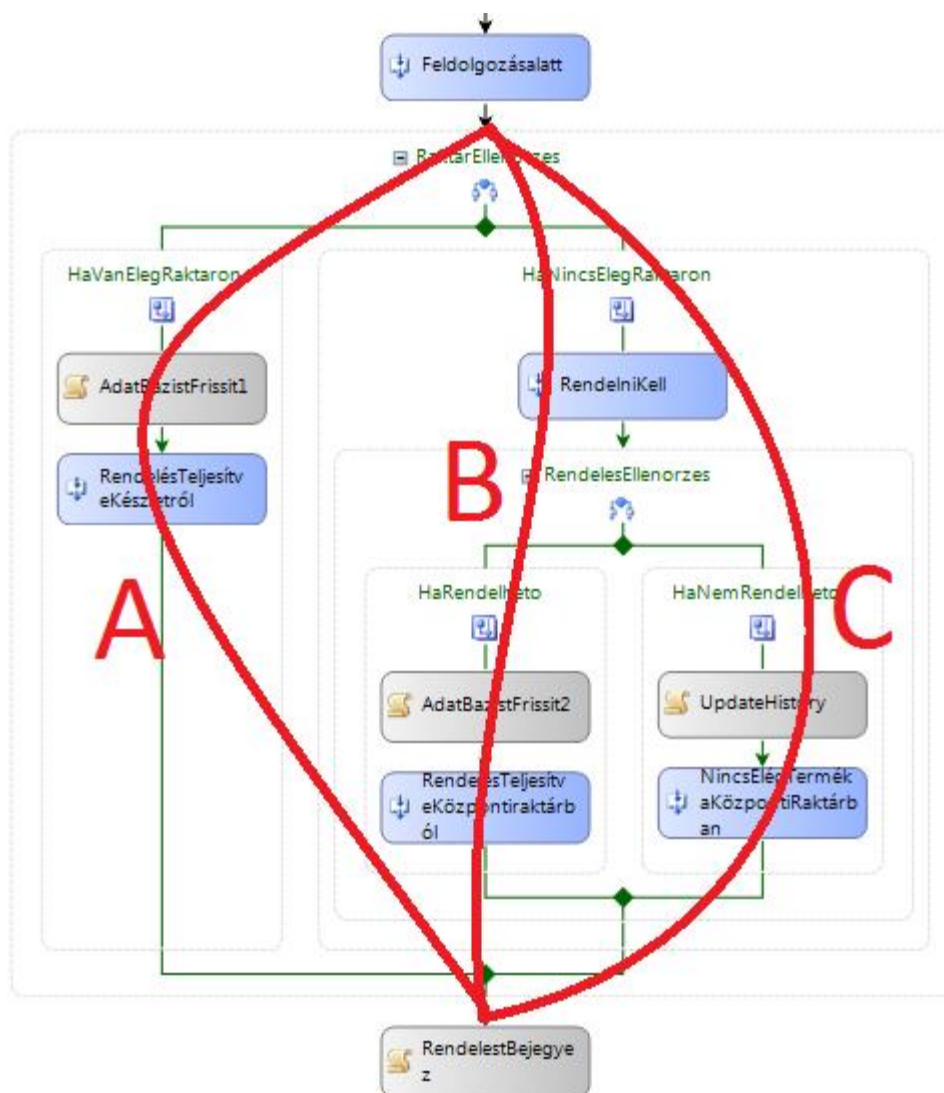
A Workflow példány elindításakor megjelenik a debugger ablak:



27. ábra. Workflow debugger

Ha egyszerre nézzük a **Debugger** és a **MainForm** ablakot részletesen kielemezhetjük a program futását. A **CallExternalEvent** activityk működése közvetlenül a **MainForm** ablakon látható, a **Code activityk** futása pedig végigkövethető.

Tekintve, hogy a Workflow két darab elágazást tartalmaz, a futás három lehetséges útvonalon történik. Ezeket elnevezem A,B,C forgatókönyveknek. (28. ábra)



28. ábra. WF végrehajtási útvonalak

Az „A” forgatókönyv szerint van elég termék a készleten, a „B” szerint nincs elég a készleten, de a készletről és központi raktárból még teljesíthető a rendelés. A „C” forgatókönyvre akkor kerül sor, ha az előző feltételek nem teljesülnek. Ha végrehajtom mindhárom rendelési esetet, a következő rendelési adatokat kapom:

frmHistory						
	orderid	Terméknév	Mennyiség	Megkapva	Feldolgozási_idő	WF_befejezve
▶	138	XBOX 360	1	18:56:09	1,232 s készletről	18:56:11
	139	XBOX 360	9	18:56:19	3,198 s központból	18:56:23
	140	XBOX 360	50	18:56:34	készlethiány	18:56:34

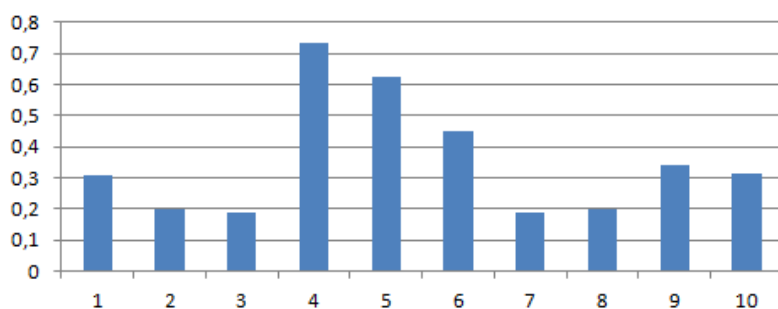
29. ábra. Az Orders tábla tartalma az A,B,C esetekkel

A program egy felhasználóra lett tervezve, de képes egy időben több rendelést is teljesíteni egy adatbázisból. Erről úgy bizonyosodhatunk meg, ha megnyitunk még egy ablakot és elindítunk egy hosszabb lefutású rendelést először az egyik, majd közben a másik rendelés gomb megnyomásával. Ez azért lehetséges, mivel a háttérben külön dolgozik a két Workflow példány. Arra azonban oda kell figyelni, hogy azonos termékek rendelésénél ne ütközzenek az adatbázis lekérdezések. A két Workflow nem látja egymást így előfordulhatnak adatbázis hibák, amik csökkenthetőek, ha az adatbázis inicializálást összekötjük a logikai vizsgálatokkal, tehát minden if/else Activity logikai vizsgálatának helyére **RuleCondition** helyett **CodeConditiont** használunk.

4.3. Eredmények kiértékelése

A 29. ábráról leolvashatjuk, hogy a 138-es ID-jú rendelés az egy másodperces késleltetéssel több, mint 1.232 másodpercig tartott. A 139-es a 3 másodperces késleltetéssel már 3,198 másodpercbe került. A 139-es pedig kevesebb, mint 0 másodpercbe tellett. Ebből látszik, hogy a Workflow első futásához több idő szükséges. Hosszú lefutású folyamatok esetén ennek nincs jelentősége. A feldolgozási idő vége egybeesik a Workflow befejezésének idejével, ez azért van így, mivel az adatbázis frissítési folyamat után nincs több nagyobb munkát végző összetettebb munkafolyamat, amit végre kéne hajtani.

Egy kisebb terhelési teszttel megállapítottam, hogy az adatbázis elérési ideje készletről való rendelés esetén késleltetés nélkül [186 -731] ms közötti skálában mozog:



30. ábra. Adatbázis elérési ideje

Ez az eredmény függ a számítógép teljesítményétől is. A tesztet a következő hardver és szoftver konfigurációval végeztem:

- windows7 64 bit
- intel i3 380M processzor 2.53Ghz
- 4 Gb DDR3 memória.

Ez az eredmény a Workflow-ról nem sokat mond, viszont látható, hogy lokális adatbázis esetén hány másodperc az adatelérési idő. Ha SQL szerveret használnánk, az elérési időt tovább csökkenthetnénk, távoli adatbázis esetén pedig ennek az időnek a többszörösére lenne szükség.

4.4. Programfejlesztési lehetőségek

A működő alapprogram tesztelése után, felmerül a kérdés, hogy miképpen lehetne fejleszteni a programot és milyen új funkciókat lenne érdemes beépíteni.

A `CallExternalMethod` mellett alkalmazhatnánk a `HandleExternalEvent` activity-t, ezáltal a felhasználó be tudna avatkozni a Workflow végrehajtásába. Például a felhasználó visszavonhatná a megrendelést mielőtt azt kiszolgáltatná a Workflow.

Nagyszabású fejlesztés lenne, ha a program felhasználói felületét implementálnánk egy ASP.NET-es weboldalba, és az adatbázis egy adatbázisszerveren lenne. Ide azonban komolyabb szinkronizációs problémákat kéne megoldani, hogy egy időben több felhasználó is rendelhessen. Olyan WF eszközökkel kéne dolgozni, mint az `SqlWorkflowPersistenceService` vagy `ManualWorkflowSchedulerService`.

A Workflow Tervezőfelületén elférne még pár komponens, amik használhatóbbá tennék a programot. Párhuzamosan, a WF futásával egyidejűleg, vagy akár külön Workflowban megvalósíthatnánk az újrendelési algoritmust, hogy mindig legyen elegendő termék a készleten. Érdemes lenne felhasználni a Windows Communication Foundation elemeit is.

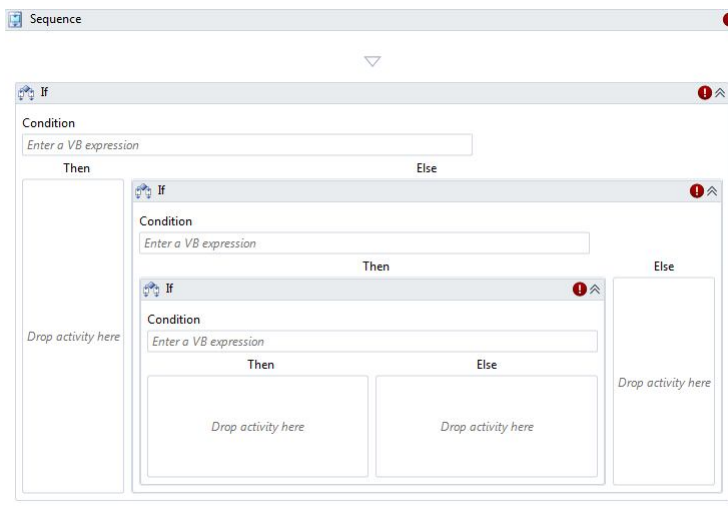
Ha már elkészítettük a programot NET 3.5-ben felvetődik bennünk a kérdés, hogyan nézne ki a program OPWorkflow része 4.0-ás Framework –al. Kisebb kerülőutakkal megvalósítható a feladat. Erről fog szólni a következő fejezet, amit kiegészítésképp írtam a szakdolgozathoz.

4.5. A Workflow megvalósítása a .NET 4.0-val

A feladat kivitelezése több szempontból is komplikált. A megvalósítás közben a következő problémákkal kellett szembenéznem:

- *A .NET 4.0 teljesen különböző szerelvényekkel dolgozik.* Nincs semmi kompatibilitás a két verzió között. Míg a NET 3.5-ben a **System.Workflow.Activities** és **System.Workflow.Runtime** volt az alapsztály, itt csak a **System.Activities**-re van szükség.
- *A változók és ki/bemeneti tulajdonságok megadása csak xml felületen keresztül lehetséges:* Tehát vagy a tervezőfelületen gépeljük be egyesével a változókat, vagy a xaml fájlba dolgozunk. Nincs lehetőség C# osztályon belüli inicializálásra.
- *Visual Basic:* Korábban említettem, hogy a grafikus tervezőfelületen való munka során, ha egy Activity-nél logikai feltételeket vagy kifejezéseket akarunk megadni azt csak Visual Basic szintaktikával tehetjük. Egy online VB -> C# fordító^[6] segítségével azonban sikerült megbirkóznom a nyelvvel.
- *Code Activity és CallExternalMethod Activity hiánya:* A tervezőfelület eszköztárán lévő elemek között nem található meg a leggyakrabban használt két Activity. A **Code Activity** helyett saját Activityket kellett létrehoznom, amik külön Workflow-ként működtek, tehát nem látták a fő Workflow változóit és tulajdonságait. Ebből kifolyólag egyesével kellett átadni neki a változókat kimeneti és bemeneti tulajdonságok formájában. A **CallExternalMethod**ot pedig az **InvokeMethod**-al lehetne helyettesíteni, ez az elem viszont nem az interfész kommunikációra van kitalálva, úgyhogy nem raktam bele a programba. Helyette **Tracking Service**-t használok, ami jelzi, ha termékkiszolgálás történik.
- *A Sequential Activity Design nem alkalmas az összetettebb folyamat ábrázolásra:* Az elemek egymásba ágyazása szekvenciális tervezésnél átláthatatlan. Egy többszörös if/else ág például így néz ki:

⁶ Forrás: <http://www.carlosag.net/tools/codetranslator/>



31. ábra. Többszörös elágazás megvalósítása WF4-ben

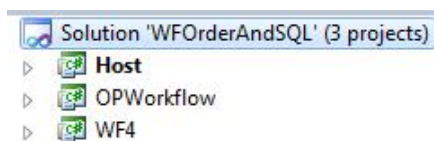
A képen (31. ábra) nem láthatóak a végrehajtási útvonalak, egy nem hozzáértő szemlélő szemszögéből nézve szinte értelmezhetetlen a munkafolyamat. Ebből kifolyólag a programot a **Flowchart** tervezési eszközeivel kell megvalósítani. A **Flowchart** egy bizonyos átmenetet képez a szekvenciális és az állapotgép tervezés között. A Workflow végrehajtása egy kezdőpontból történik, és nincs megadva végpont. Nagy előnye a .NET 3.5-ös szekvenciális tervezéssel szemben, hogy lehetőség van ugrálni az Activity-k között.

A felsorolt hiányosságok mellett, a Workflow 4 rengeteg olyan újítást tartalmaz, amiért érdemes áttérni a használatára. Ha az alkalmazásunkból szeretnénk indítani egy Workflow példányt egy egysoros rövid kóddal megtehetjük ezt:

```
WorkflowInvoker.Invoke(new Workflow1());
```

Nincs szükség a Runtime létrehozására, és az új osztályok tartalmazznak olyan függvényeket, amik segítségével még jobban támogatott a Host és Workflow közötti interakció.

A megvalósításhoz első lépésben létre kellett hozni, egy új **WF4** nevű projektet, mivel két Workflow verzió egy projekten belül nem lehetséges. Az új Workflow 4 nem különálló programként fog működni, hanem bele lesz építve a Solution-be. A Form-on keresztül tudunk választani, hogy melyik Workflow-val történjen a végrehajtás.



32. ábra. Az új Workflow példányt külön projektként kell kezelni

Szükség van a Code Activity helyett új Activityk létrehozására a beágyazott Workflow 4-es **CodeActivity** sablon segítségével:

```
public sealed class CodeActivity1 : CodeActivity
{
    public InArgument<string> Text { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        string text = context.GetValue(this.Text);
    }
}
```

Az **InArgument** olyan tulajdonságokat jellemez, amiket átveszünk a Workflowunk-ból, az **OutArgument**-hez pedig az Activity kimenete társul, tehát amit visszaadunk a Workflownak. Az Execute függvényen belül történik a tényleges végrehajtás. A ki és bemeneti tulajdonságok a **context.GetValue()** és a **context.SetValue()** segítségével érhetők el. Az **AdatbázistFrissit1** függvény például így néz ki lekódolva:

```
public sealed class AdatbázistFrissit1 : CodeActivity
{
    public InArgument<OleDbConnection> Con { get; set; }
    public InArgument<String> RendeltTermék { get; set; }
    public InArgument<Int32> Készleten { get; set; }
    public InArgument<Int32> RendeltMennyiség { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        Con.Get(context).Open();
        string updateString =
            @"UPDATE készlet SET mennyiség = "
            + (Készleten.Get(context) - RendeltMennyiség.Get(context))
            + " WHERE terméknev = '" + RendeltTermék.Get(context) + "'";

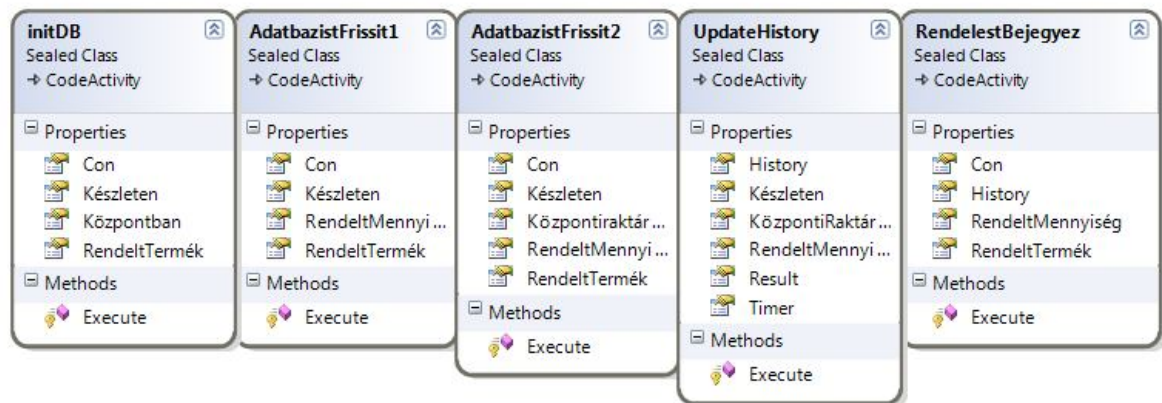
        OleDbCommand cmd = new OleDbCommand(updateString);
        cmd.Connection = Con.Get(context);
        cmd.ExecuteNonQuery();
        if (Con.Get(context) != null) { Con.Get(context).Close(); }
    }
}
```

Miután rámentünk a build solution-ra megjelenik az elkészített AdatbázistFrissit1 Activity az eszköztáron.

Misc		
Con	connection	...
DisplayName	AdatbázistFrissit1	
Készleten	keszleten	...
RendeltMennyiség	rendeltMenny	...
RendeltTermék	rendeltTermek	...

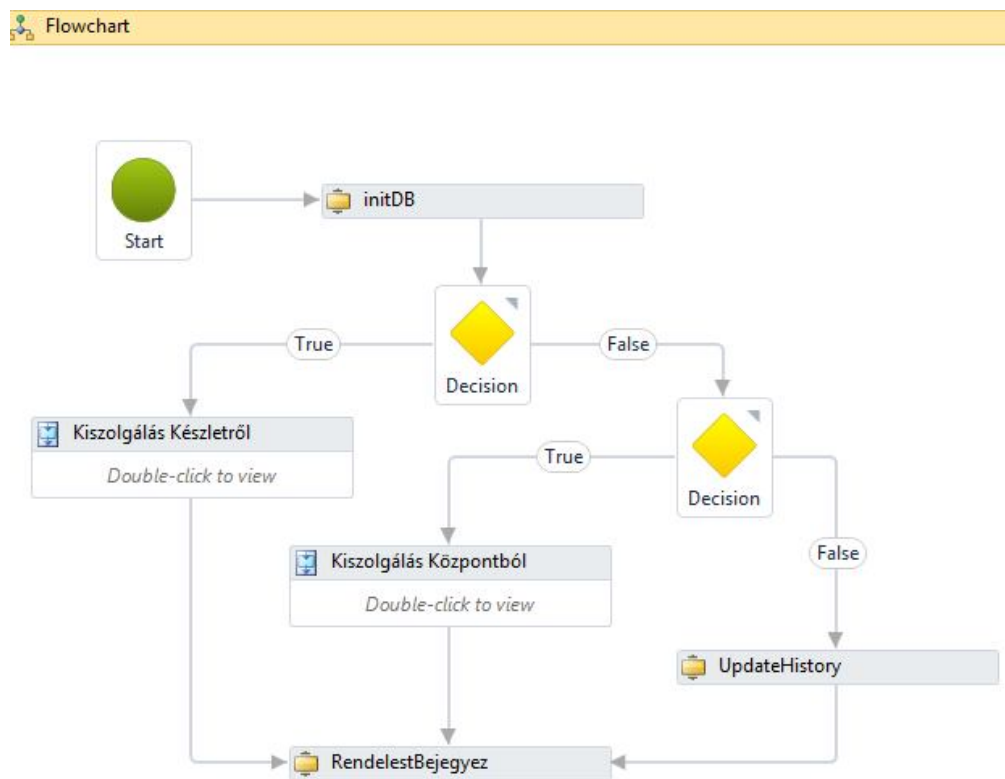
33. ábra. AdatbázistFrissit Activity tulajdonságai

Hasonlóképp el kell készíteni a többi **CodeActivity**-t:



34. ábra. WF4 Projektben használt saját Code Activity-k

Miután megadtuk a Workflow változóit és tulajdonságait és elkészítettük az Activityket, maga a Workflow elkészítése már nem nagy feladat:



35. ábra. Az elkészült .NET 4-es Workflow diagram

A 35. ábrán látható kész Workflow .NET 4-es diagram működése majdnem megegyezik a 22. ábrán lévő Workflow 3.5-ös verziójával, itt azonban nincsenek státuszüzenetek vég-

rehajtás közben. Az **if/else** elágazások helyett **Decision Activity**-ket használtam, ami ábrázolási szempontból előnyösebb.

Mivel a Workflow 4-nek nincs konstruktora, ezért létrehozáskor adtam kezdőértéket a változóknak, ebből kifolyólag a kiszolgálási idő megnövekedett:

Name	Variable type	Scope	Default
rendeltTermek	String	Flowchart	Termék
rendeltMennyiség	Int32	Flowchart	Mennyiség
keszleten	Int32	Flowchart	Enter a VB expression
kozpontiRaktarban	Int32	Flowchart	Enter a VB expression
history	String[]	Flowchart	New String(2) {DateTime.Now.ToLongTimeString(), "", ""}
timer	Double	Flowchart	DateTime.Now.Minute * 60000 + DateTime.Now.Second * 1000 + DateTime.Now.Millisecond
connection	OleDbConnection	Flowchart	New OleDbConnection("Provider=Microsoft.JET.OLEDB.4.0;data source=db.mdb")

36. ábra. Workflowban használt változók definiálása

Az elkészült Workflow-ot már csak meg kell hívni a Host alkalmazásból átadva neki a ugyanazt a **Dictionary**-t amit a korábbi 3.5-ös verziónál használtunk. Annak eldöntésére, hogy melyik Workflow-val történjen a rendelés kiszolgálása, két **radioButton**-t helyeztem el a Form felületén:

37. ábra. Az új Form design, Workflow választó rádiógombokkal

A Rendel gomb megnyomására a következő esemény játszódik le:

```
if (this.radioButton1.Checked == true)
{
    #region Start Workflow .NET 3.5
    ...
}
else
{
    #region Start Workflow .NET 4.0
    ...
}
```

Ezen belül az else ág a lényeges, hiszen ezzel indítjuk ez a 4.0-ás Workflow-ot. Mivel itt nem használok Interfészt ezért a Workflow példányt szinkronizálni kell. A Workflow elindítása a következőképp történik:

```
var workflow = new workflow2();
var app = new WorkflowApplication(workflow, properties);
app.SynchronizationContext = SynchronizationContext.Current;
app.Run();
```

Még ugyanitt létrehozok egy eseménykezelőt a Workflow befejezéséhez, és egyet a tétlen állapot jelzéséhez:

```
app.Completed = delegate(WorkflowApplicationCompletedEventArgs e2)
{
    tbState.Text += Convert.ToString(e2.Outputs["Eredmény"]);
    Thread.Sleep(500);
    this.tbState.Text += "\r\n\nWorkflow Befejezve";

    ...
};

app.Idle = delegate(WorkflowApplicationIdleEventArgs e2)
{
    tbState.Text += "\r\n . . .";
};
```

És ezzel elkészült az alkalmazás .NET 4.0-ás megfelelője. Működési szempontból nem sokban tér el az előző Workflow -tól. Jól látszik, hogy a tervezés során használható C# programozási eszközök száma kevesebb, mint a .NET 3.5-ben, a grafikus motor viszont jóval letisztultabb elődjénél. A termékrendelési folyamat mostantól mindkét keretrendszerrel megvalósítható a programban.

Összefoglalás

A szakdolgozat célja egy Windows Workflow Foundation alapú alkalmazás készítése volt, amely képes kiszolgálni termékrendelési folyamatokat. A kitűzött cél volt még, hogy bemutassam a munkafolyamat grafikus szemléletű tervezésének előnyeit a Workflow tervezője segítségével. Véleményem szerint ezt sikerült megvalósítani. A program tartalmazza az olyan alapvetőbb funkciókat, mint: Rendelés, Adatbázis szerkesztés, Workflow, Kommunikáció.

A program elkészítésénél törekedtem arra, hogy a későbbiekben továbbfejleszthető legyen. A következő szolgáltatásokat mindenképp szeretném még megvalósítani: felhasználó oldali kommunikáció beépítése, több felhasználó egyidejű kiszolgálása. A tovább fejlesztésnél a fentebb említett szolgáltatásokon kívül, a grafikai fejlesztésekre is nagy hangsúlyt kell helyezni.

A 4.5.-ös fejezetben megmutattam, hogy milyen kihívásokkal jár átültetni egy hagyományos Workflow alkalmazást 4.0-ás környezetbe, hogy ezzel megfeleljen korunk Workflow elvárásainak. A munkafolyamat alapú alkalmazások népszerűsége folyamatosan növekszik. A közeljövőben talán közelebb hozza egymáshoz az informatikai és az üzleti szektort.

Irodalomjegyzék

- [1] Andrew Troelsen: A C# 2008 és a .NET 3.5. Szak kiadó, Budapest, 2009., 342-360. oldal
- [2] Bruce Bukovics: Pro WF Windows Workflow in .NET 3.5, Apress kiadó, 2008., 1-129. oldal, 217-236. oldal
- [3] Robert Eisenberg Teach Yourself Windows Workflow Foundation, Sams kiadó, Indiana, 2009 10-33. oldal
- [4] Andrew Troelsen: Pro C# and the .Net 3.5 Platform, Apress kiadó, New York, 2007., 189. oldal, 269-308. oldal
- [5] Carlos Aguilar Mares: basic scriptek fordítása c# ra
<http://www.carlosag.net/tools/codetranslator/>, 2011
- [6] Balássy György: Szakmai Blog
<http://balassygyorgy.wordpress.com/tag/workflow/>, 2008
- [7] MSDN Lybrary: Windows Workflow Foundation tutorials
<http://msdn.microsoft.com/en-us/library/ms735927%28v=VS.90%29.aspx>, 2011
- [8] Andrew Troelsen: Pro C# 2010 and the .NET 4 Platform 1077-1114. oldal