Szoftverfejlesztést támogató eszközök fejlesztése .NET ill. Java környezetben

Varga Tamás

E7BN3F

Dátum:2020. 12. 16.

Bevezetés

Témalabor során Java nyelven, a *SonarQube* által használt szabályok bővítésével foglalkoztam. Addig számomra a platform ismeretlen volt. Már az első használat során rettentő hasznosnak találtam. Ezek után kezdett el jobban érdekelni a téma, hogy az egyes fejlesztést támogató rendszereknek, segédleteknek jobban utánanézzek. Korábbi tanulmányaimból visszaemlékezve az objektumorientált fejlesztéshez számomra a nagy segítséget az UML diagramok megismerése jelentette. Az egyes diagramok megjelenítéséhez, szerkesztéséhez használt szoftverek, mint a *WhiteStarUML* jól működtek, de volt velük néhány probléma. Jól lehetett szerkeszteni, azonban gyengébb hardverű gépeken, mint akkoriban az enyém, nagyobb diagramoknál rendszerint akadozott a megjelenítés. Ez mellett hiányzott belőle az átláthatóság szabályozása. Az ember maga adta meg a diagram felépítését, és kevésbé precíz munka során egészen átláthatatlan lett a kapott kép.

A téma címéből olvasható, hogy egy fejlesztést támogató eszköz fejlesztése volt a feladat. Részletesebben ez azt jelentette, hogy egy WhiteStarUML programban készült osztálydiagrammot az általam készített „eszköz” beolvasson, majd megjelenítse, később pedig a szerkeszthető is legyen.

UML

A feladat megoldásához frissíteni kellett az ismereteimet az UML szabvánnyal kapcsolatban. Az UML (*Unified Modelling Language*) egy szabvány szoftverrendszerek modellezésére. A modell a valóság egy egyszerűsített változata. Ha jól van elkészítve, akkor hangsúlyozza a fontos részleteket és elhanyagolja az irrelevánsokat. Az UML szabvány többféle diagramtípust definiál. Ezek különböző nézeteket adnak meg a rendszerről. A szabvány két dolgot ír le

- **szintaxis**: hogyan néznek ki az egyes diagramok (azaz a nézetek)
- **szemantika**: mit jelentenek a diagramok egyes elemei

A különböző nézetekből, azaz a szintaxisból kiolvasható szemantikának egymással konzisztensnek kell lenniük. Tehát a különböző nézeteken, diagramokon megjelenő elemek, adatok, összefüggések, nézettül függetlenül ugyan azt jelentik

A szemantikának, hogy az egyes elemek a diagramokon mit jelentenek, két fontos aspektusa van:

- **Strukturális szemantika**: a modellezett rendszer egyes elemeinek jelentését definiálja egy időpillanatban.
- **Viselkedési szemantika**: a modellezett rendszer elemei jelentésének időbeli változását definiálja.

A szemantikának a két fontos aspektusa szerint két csoportba lehet osztani az UML szabvány által definiált diagramokat

Strukturális UML diagramok	**Viselkedési UML diagramok**
Osztály diagram	Use case diagram
Component diagram	Szekvencia diagram

A két UML diagram csoport további diagram típusokat tartalmaznak, de a feladat szempontjából most nem lényegesek.

Use Case diagramok

Ahhoz, hogy megfelelően tudjam megjeleníteni a Use Case diagramokat, szükséges tudni, hogy milyen elemek, illetve kapcsolatok szerepelnek ezen a diagramon:

- **Aktor**: a felhasználó szerepköre, ember, vagy külső rendszer
- **Use case**: a felhasználó és a rendszer közötti interakciót reprezentálja
- **Asszociáció**: Aktor-t és Use Case-t kapcsol össze.

A kapcsolatokat három csoportba lehet csoportosítani:

- Kapcsolat Aktor és Use Case között : **Asszociáció**
- Kapcsolat Aktorok között: **Öröklődés** (generalization)
- Kapcsolat Use Case-k között**: <<extend>>, <<include>>**

Osztály diagramok

Az osztálydiagram az objektumorientált modellek leírásának szabványa. Szoftver architektúrának a dokumentálására használják szoftverfejlesztők. Maga az osztálydiagram fontos része egy fejlesztésnek, ezért is lényeges, hogy jól kezelhető, felhasználóbarát eszközzel tegyük meg.

Az osztálydiagram elemei:

- **Osztály**: objektumok közös viselkedését, szemantikáját írja le
    - **Operáció**: közös viselkedés
    - **Property**: field-ek, vagy attribútumok
- **Interfész**: publikus operációk halmaza.
- **Classifier**: az osztályok és az interfészek közös neve

Kapcsolatok az elemek között:

- **Megvalósítás** (realization)
- **Öröklődés** (generalization)
- **Asszociáció**
- **Függőség**(dependecy)

Az UML szabvány még definiál láthatóságot. Az operációknak, illetve property-knek vannak láthatóságai, de a szabványban definiált eltérhet az adott programozási nyelvben használttól.

Az UML egy gyakorlati, objektum orientált modellező, nagyméretű rendszerek vizuális dokumentálására alkalmas. Az osztálydiagramok közvetlenül programkóddá alakíthatóak. Egy széles körben elfogadott és használt szabvány, azonban mégis több kritika éri:

A szabványban a fent felsorolt diagramokon kívül még elég sokat definiál, azonban ezek egy részét alig használják, egy része pedig redundáns, tehát új információt nem adó, ismétlődő. Másrészt a szemantikát is éri kritika. A szemantika definiálásakor hiányzik a formális nyelveknél megszokott szigor. A szabvány folyamatosan fejlődik, és még nem teljesen kiforrott, kisebb méretű szoftverek fejlesztésénél, ahol nem kritérium a megfelelő dokumentálás, illetve a szoftver bonyolultsága sem kívánja meg, ott fontolóra érdemes venni a használatát.

Fejlesztés első lépései

Az elkészített alkalmazás egy WPF alapú szoftver. A WPF (*Windows Presentation Foundation*) alapja egy felbontásfüggetlen, vektor alapú render motor. A fejlesztési munkát a segíti a XAML, az adatkötés, a 2D, 3D grafika bevezetése. A WPF a .NET része, a szoftverek készülhetnek .NET programozási nyelveken, mint a C#, vagy Visual Basic.

Kiindulásként egy projektet kaptam, amiből kellett felépíteni a szoftvert. Korábbról csak kevés UWP tapasztalatom volt, ezért egyből felvetődött a kérdés, hogy átdolgozom a másik platformra a kiinduló projektet. Windows asztali alkalmazás fejlesztésére mind az UWP (*Universal Windows Platform*), mind a WPF egy jó kiinduló alap lehet, eltérő erősségekkel, gyengeségekkel. A szükséges fejlesztői képességek hasonlóak, tehát a migráció a kettő között nem lehetetlen.
UWP a vezető platform Windows 10 alkalmazások és játékok fejlesztésében. Mellette rendkívül jó támogatást élvez különböző tervezési minták felé NuGet csomagok formájában.

Egy hosszabb átgondolás után végül arra az állásra jutottam, hogy nem érdemes a kiinduló projektet élőről újra fejleszteni, hiszen az UWP nem ad annyi előnyt, ami megtérítené a szükséges fejlesztési időt. Az előnyökről, illetve hátrányokról a Microsoft hivatalos oldalán tudtam tájékozódni.

Fontolóra vettem, hogy tervezési mintát használok a fejlesztés során. MVVM tervezési minta jött szóba, mint lehetőség. A minta alkalmazásának az egyértelmű előnye, hogy tisztán elkülöníti a megjelenítést a szoftver logikai, modell részétől. Az MVVM mintát azonban ritkán implementálják le kézi módszerekkel, helyette valamilyen előre elkészített osztály könyvtárat használnak, amelyek segítenek általános alkalmazáséletciklussal kapcsolatos feladatokat megoldani. UWP esetén többször implementáltam már a Template10 könyvtárat, de ez WPF alatt nem elérhető. Prism NuGet csomaggal próbálkoztam a meglévő projektben az implementálással, de inkább kevesebb, mint több sikerrel. Az alkalmazás fő funkciójában egy nézetet fog megjeleníteni, így annyira nem is tartozzam fontosnak az implementálást, hiszen a nézetek közötti váltás, adatcsere nem számottevő.

A feladatot két részre bontottam a kezdetekkor. Először foglalkoztam az UML beolvasásával, illetve a megfelelő modell felépítésével, majd a modell megjelenítésével.

Beolvasás

Ahhoz, hogy jobban átlátható legyenek a lépések, vázlat szerűen a következők történnek:

- MetaDslx framework segítségével a beolvasott fájlból egy modell készül. Ez tartalmazza a különböző diagramok elemeit, kapcsolataikat egymással, illetve a diagramok között.
- Az ábrázoláshoz egy gráfot kell létre hozni. A gráf csomópontjait meg kell feleltetni az egyes diagramon megjelenő elemeknek, a gráf éleit pedig az elemek közötti kapcsolatnak.
- A megfeleltetés után a felhasználónak prezentálható formában meg kell jeleníteni a gráfot. A gráf csomópontjait megfelelően el kell helyezni, éleit pedig szintén egy jól értelmezhető módon meg kell jeleníteni.

A beolvasandó fájl egy WhiteStarUML-ben elkészített UML, osztálydiagrammal, Use Case diagrammal, Szekvenciadiagrammal, illetve egyéb diagrammokkal. A feladat során az osztály, illetve Use-Case diagramra koncentráltam. A szekvenciadiagram megjelenítése sajnos a gráfos elgondolással nem, vagy nagyon nehezen megvalósítható. Ebből kellett egy számunkra alkalmas modellt készíteni, amiből könnyedén kinyerhetőek akár az osztálydiagram, akár Use Case diagramon megjelenő elemek, kapcsolatok. Ehhez a feladathoz a MetaDslx Framework volt segítségül. A MetaDslx egy .NET Standard 2.0 alapú metamodellező framework. A framework-nek az egyik tulajdonsága, hogy teljes implementációt ad az OMG metamodel szabványoknak, tehát a MOF, UML-nek.

Lehetőséget biztosít tehát egy UML fájl szerializálására. Megadott minta alapján a beolvasás gyakorlatilag három sor kódból megoldható. Ebben az esetben a *model* változóban tároljuk a beolvasott adatokat.

```
2 references
protected void loadModel(string fileName)
{
    UmlDescriptor.Initialize();
    var umlSerializer = new WhiteStarUmlSerializer();
    var model = umlSerializer.ReadModelFromFile(fileName, out var diagnostics);
```

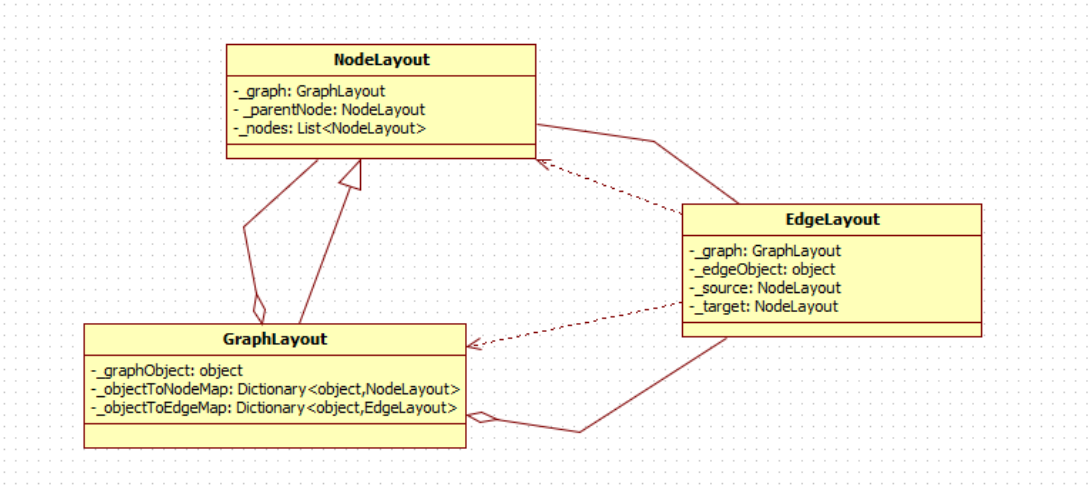
A beolvasás megtörtént, a model-ben az UML szabvány által meghatározott, felhasználó által WhiteStarUML-ben elkészített nézetek (diagramok), a bennük megjelenő elemek, kapcsolatok megtalálhatóak.

Ez után a következő lépés egy olyan művelet elkészítése, amely elvégzi a megfeleltetést gráf, és a beolvasott modell között.

Szerencsére egy gráf rajzolásához vannak open-source megoldások. Ilyen például a *Graphviz* is (https://graphviz.org/)
Ezt a megoldást használja a MetaDslx.Graphviz. A Graphviz segítségével egy gráf csomópontjainak, illetve éleinek az elhelyezkedését meg tudja határozni. Ezekhez úgynevezett Layout-k vezettek be.
A projektben szereplő osztályok közöl a használat során a következőek a legfontosabbak:

- NodeLayout
- GraphLayout
- EdgeLayout.

A GraphLayout reprezentálja a gráfot, a NodeLayout a csomópontokat, de maga a GraphLayout a NodeLayout-ból származik le. A GraphLayout példányok egy-egy Dictionary-ban tárolják a gráfhoz tartozó csomópontokat, illetve éleket. Az EdgeLayout reprezentálja az éleket. A csomópontok, élek, mint Layout-k tartalmazzák a megjelenítéshez szükséges pozíciókat, méreteket, és tárolják egy field-ben a modellből, ahhoz a csomóponthoz, tehát valamilyen elemhez (például osztály), vagy élhez (valamilyen kapcsolat, például örökölés) tartozó információkat.



A megfeleltetéshez készültek a következő osztályok:

- GraphLayoutLoader: ősosztály, definiálja azokat a metódusokat, amelyek elvégzik a megfelelő megfeleltetést a Layout-k és a modell között.

- LogicalViewmodelLoader: az osztály diagram Layout-nak betöltéséért felelős, a GraphLayoutLoader leszármazottja. Tudja, hogy milyen elemek, kapcsolatok fordulnak elő egy osztály diagramban, és az ott megjelenőket megfelelteti a Layout-oknak.
- UseCaseViewLoader. a Use-case diagram Layout-nak a betöltéséért felelős, a GraphLayoutLoader leszármazottja. Tudja, hogy milyen elemek, kapcsolatok fordulnak elő egy use-case diagramban, és az ott megjelenőket megfelelteti a Layout-oknak.

Az ősosztályban vannak definiálva azok a metódusok, amelyek a gráfhoz a modell alapján csomópontokat, éleket képesek rendelni, tehát a megfelelő Layout-t meghatározzák.
Ez összesen 11 metódust jelent, és a leszármazottak, attól függően, hogy milyen diagramot is szeretnénk megjeleníteni, használják fel.

Az egyes Loader osztályok rendelkeznek egy-egy GraphLayout field-el, ebben tárolják a betöltött csomópontokat, éleket. A megfelelő nézet is tartalmaz egy GraphLayout példányt, és egy értékadással töltjük a nézetbe az értéket. így a nézet, és a modell között nagyon kis kapcsolat van, amely segíti akár a tesztelést, akár a továbbfejlesztést is.

Egy konkrét metódus, a GraphLayoutLoader-ből:

```
protected void AddGeneralizations(IEnumerable<Generalization> generalizations)
{
    foreach (var gen in generalizations)
    {

        var allnodes = Layout.AllNodes.ToList();
        NodeLayout specNL = null;
        NodeLayout genNL = null;
        foreach (var n in allnodes)
        {
            var namedNode = (MetaDslx.Languages.Uml.Model.NamedElement)n.NodeObject;
            if (string.Equals(namedNode.Name, gen.Specific.Name)) specNL = Layout.FindNodeLayout(namedNode);
            else if (string.Equals(namedNode.Name, gen.General.Name)) genNL = Layout.FindNodeLayout(namedNode);

        }
        //if (specObject == null) { Layout.AddNode(gen.Specific.Name); specObject = Layout.FindNodeLayout(gen.Specific.Name); }
        //if (genObject == null) { Layout.AddNode(gen.General.Name); genObject = Layout.FindNodeLayout(gen.General.Name); }
        if (specNL != null && genNL != null)
        {
            var e = Layout.AddEdge(specNL.NodeObject, genNL.NodeObject, specNL.NodeObject.ToString() + "-|>" + genNL.NodeObject.ToString());

        }
    }

}
```

Az *AddGeneralizations(IEnumerable<Generalization> generalizations)* metódus a modellben található öröklés kapcsolatokat adja Layout-hoz. Fontos, hogy egy-egy kapcsolat, vagy elem ne kerüljön többször a gráfba. Azért is fontos, hiszen egy kapcsolatnak két tulajdonosa van, így elkerülhetjük, hogy duplikáljuk ezeket.
A Layout, ami egy GraphLayout Property, számon tartja a már hozzáadott csomópontokat, így megvizsgálhatjuk, hogy az öröklés, mint kapcsolat két fogadó fele hozzá lett-e adva. A kapcsolatok hozzáadása előtt az adott diagramnak megfelelő elemek, tehát osztályok, interface-k, stb. is hozzáadásra kerül, így ha esetleg egy olyan kapcsolat kerül a modellbe, aminek nem definiáltak a tulajdonosai, nem lesz a Layout-hoz adva.
Ezután, ha megvannak a kapcsolat tulajdonosai, akkor a Layouthoz kerül a kapcsolat, az él.

```
public void LoadLayout(string fileName)
{


    this.loadModel(fileName);

    this.AddClasses(model.Objects.OfType<Class>());

    this.AddInterfaces(model.Objects.OfType<Interface>());

    this.AddEnumerations(model.Objects.OfType<Enumeration>());

    this.AddGeneralizations(model.Objects.OfType<Generalization>());

    this.AddInterfaceRealizations(model.Objects.OfType<InterfaceRealization>());

    this.AddDependecies(model.Objects.OfType<Dependency>());

    this.AddAssociations(model.Objects.OfType<Association>());

    Layout.NodeSeparation = 10;
    Layout.RankSeparation = 50;
    Layout.EdgeLength = 30;
    Layout.NodeMargin = 20;
    Layout.ComputeLayout();
}
```

A LogicalViewmodelLoader leszármazottban pedig felhasználásra kerülnek ezek a metódusok. A hozzáadások után a Layout egyéb paraméterei is beállításra kerülnek, majd meghívásra kerül a ComputeLayout() metódus. Ez a GraphLayout osztály metódusa, ez a metódus felelős a csomópontok helyeinek, méreteinek a meghatározásáért.

Megjelenítés, adatkötés

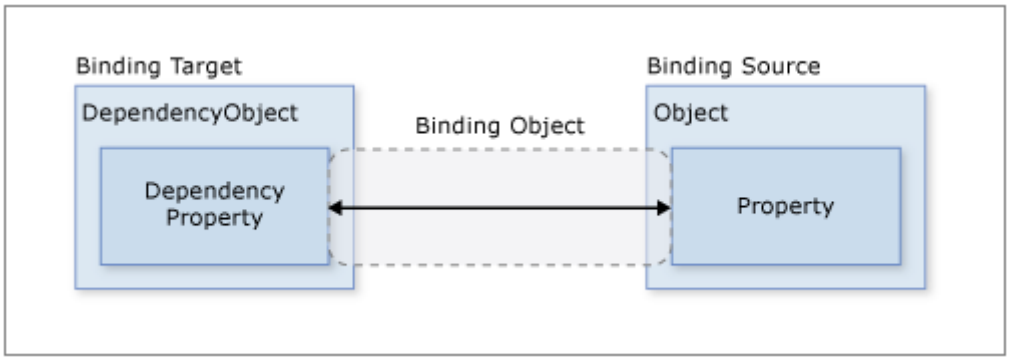
Az úgynevezett back-end résszel miután készen voltam, foglalkozhattam a megjelenítéssel.

Az adatkötés (*data binding*) egyszerű, és konzisztens módja, hogy egy alkalmazásban adatokat jelenítsünk meg, és ezekkel az adatokat módosítsuk. Az adatkötés funkció rengeteg előnnyel jár a hagyományos modellekkel szemben. Magával hoz egy jó adag property-t, rugalmas UI megjelenítést az adatoknak, és talán a legnagyobb előnye, hogy a szoftver üzleti logikája elszeparált a megjelenítéstől.

Az adatkötés tehát egy kapcsolat az adat, és annak megjelenítése között. Ha a kötés jól van beállítva, és az adat rendelkezik megfelelő notification-okkal, akkor amikor az adat változik, akkor a hozzá kapcsolt (általában valamilyen UI) elem automatikusan változik. Érdemes itt megemlíteni az Observer tervezési mintát. Ennek a rövid magyarázata, a következő: az adatokat, rajtuk végzett műveleteket emeljük ki egy osztályba, ez lesz a dokumentum. A dokumentumhoz különböző nézeteket lehet beregisztrálni, ezek az observerek. Ha egy nézet megváltoztatta valamilyen adatját, akkor a dokumentum értesíti az összes beregisztrált nézetet. Az értesítés hatására pedig minden nézet lekérdezi az adatokat, és frissíti magát. Mivel egyszerre csupán egyfajta nézetet, a gráf nézetét szeretném megjeleníteni, így ez a architektúrális tervezési mintát nem vezettem be, de kiinduló projekt megjelenítése is hasonló elveket követett.

Adatkötést általában a modell (vagy nézetmodell) és a nézet közötti objektumok között létesítünk. Ezt a WPF a DependencyProperty segítségével oldja meg, ami mindig a nézet felől van definiálva. WPF specifikus a DependencyObject alaposztály, neki a property leírója a DependencyProperty.

A Property önmagában még nem elég, hiszen valahogy a kötést is biztosítani kell. Ezt a Binding osztály segítségével érhetjük el.



Két nézetet tartalmaz az alkalmazás. Egy MainWindow-t, illetve egy DiagramView nézetet. Ez egy nézet a nézetben eset, amikor a fő nézet nagy részét a DiagramView tölti ki, ami felelős közvetlenül a megfelelő diagram kirajzolásáért.
A korábban említett Loader osztályok a MainWindow-ban vannak implementálva, és itt történik a DiagramView nézet GraphLayout property-nek a beállítása is. A nézethez az adat kapcsolása (Binding) templatek alapján megtörtént, a megfelelő Dependecy Propertyk a DiagramView nézet felől vannak definiálva.

A megjelenítéshez segít DrawingVisual. Ez egy WPF megjelenítési rétegben használandó osztály. Lightweight rajzoló osztály, alakzatok, képek, szövegek renderelésére. Lightweight abból ered, hogy például nem tartalmaz beépített eseménykezelést, ami nagymértékben növeli a teljesítményt.

Ahhoz, hogy DrawingVisual objektumokat használhassunk, egy Host csomagolót kell készíteni. Ennek a Host csomagoló osztálynak kötelezően a FrameworkElement osztályból kell leszármaznia,

ami biztosítja például az eseménykezelést. A Host-nak kifelé nincs publikus Propertyje, hiszen a fő feladata gyermek objektumok tárolása.
Amikor a Host készül, akkor a DrawingVisual objektumok referenciáját kollekcióba érdemes eltárolni. A szoftver esetén csupán egy diagram megjelenítése lesz, ezért nem szükséges kollekciót használni, hiszen nincs több fajta alakzat rajzolása.

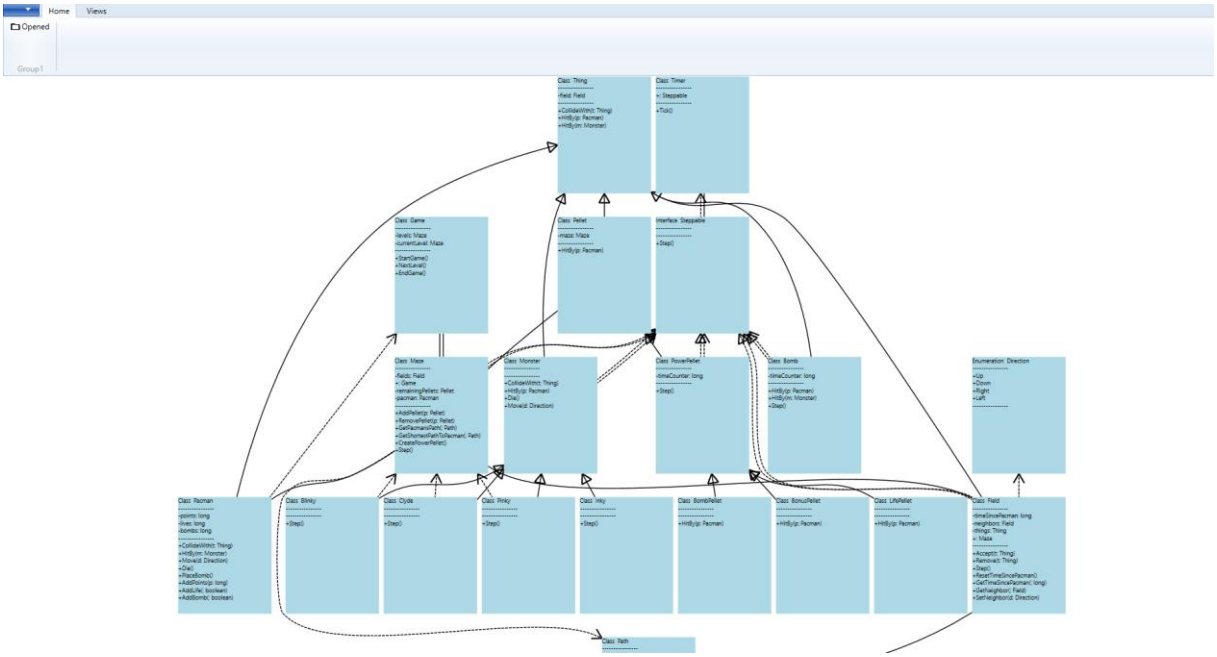
A fő feladata tehát a Host-nak DrawingVisual tárolása. Ez a tárolt, esetünkben 1 darab Visual határozza meg a megjelenített képet. A BindGraphLayout(GraphLayout layout) metódusa, a paraméterként kapott Layout alapján alakítja a Visual-t.

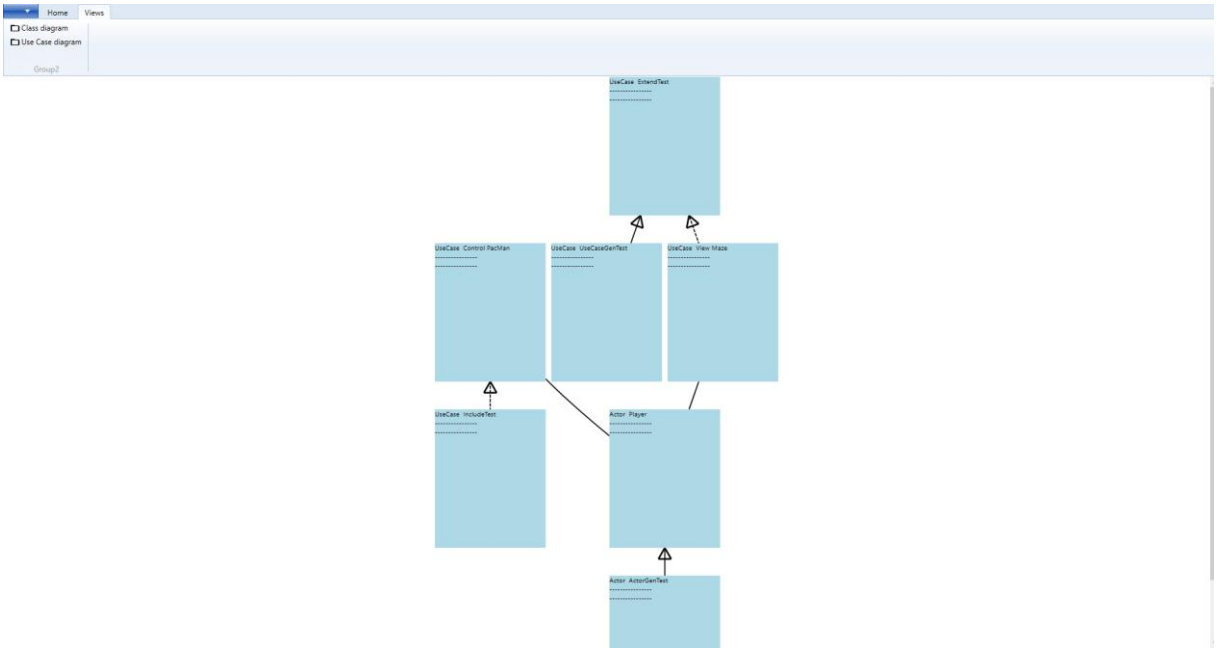
A metódus a Layout-hoz tartozó összes csomóponton meghívja a Draw() metódust. Ez a metódus csomópont esetén egy téglalapot, és benne a megfelelő attribútumokat, metódusokat rajzolja ki. Él esetén pedig a kapcsolat fajtájától függően szaggatott vagy folytonos töröttvonalat rajzol, valamint, ha szükséges, akkor nyilakat is a végére. A nyilak egy egyszerű módszer szerint számítódnak. Az él végpontja adott, oda kell a háromszög felső csúcsát tolni, és a csúccsal szemközti alaphoz tartozó magasságot pedig az él irányába forgatni, feltételezve, hogy az él végén levő „nyíl" egy szabályos háromszöget alkot.

Összegezve, a megjelenítés folyamatai nagyvonalakban:

- A MainWindow code behind-ban megtörténik az UML fájlból a modell beolvasása, valamint a gráfhoz tartozó GraphLayout megfeleltetése
- A DiagramView nézet (ami a diagram megjelenítéséért felelős) GraphLayout Propertiét átállítjuk az előbb beolvasottra.
- A beregisztrált Dependecy Property megváltozott, ami elsüti a regisztrációkor megadott metódust. További metódushívásokon keresztül eljutunk a Host (*DiagramVisualHost osztály*)BindGraphLayout metódusához.
- A Host tárolja a Visual objektumot. A BindGraphLayout metódus paraméterében megadott Layout alapján kirajzolódnak a téglalapok, TextBox-k, illetve az éleknek megfelelő vonalak.

A mintaként használt Pacman osztálydiagram a következőképpen néz ki az elkészített szoftveren:





Tapasztalatok, nehézségek

A félév elején egy kiinduló projektet kaptam. Ez volt talán az eddigi legnagyobb projekt, amiben dolgoznom kellett. Nem volt teljesen ismeretlen a technológia, maga a WPF, hiszen UWP tapasztalataim már voltak, és az átjárás a kettő technológia között nem lehetetlen. A kiinduló projekt megértése nem volt túl egyszerű, hiszen tartalmazott üzleti logikát, illetve megjelenítést is. A projektnél használt MetaDslx framework-nél egyszerűbben eligazodtam, példa alapján egyértelmű volt, hogy melyik műveleteket kell használni a beolvasás során. A megjelenítés során rengeteget kellett debug-nom. Kezdetben a MetaDslx-t NuGet package-ként adtam a projekthoz, és bátran támaszkodtam rá, hogy megfelelően működik. Azonban a csomópontok, és az élek rendre pontatlanul jelentek meg. Végül a MetaDslx.GraphLayout ComputeLayout() metódusában megtaláltam a hibát. Sikerült felfrissíteni az UML-l kapcsolatos ismereteimet, illetve sokat tanultam a WPF adatkötésekről.

Vannak további elvégzendő feladatok az alkalmazással. Jelenleg az elkészített szoftver az osztálydiagramokat, Use-case diagramokat gyorsan, és átláthatóan jeleníti meg. Vírus miatti több hetes betegség miatt sajnos nem jutott idő a szerkesztésre, de a továbbiakban szeretném megvalósítani. A nehézséget nem a beolvasott modell változtatása jelenti, hanem a változtatott modell “deszerializálása”, amihez a megfelelő kódot a MetaDslx-ben szükséges megírni. Ez mellett érdemes lehet a gráfok éleinek változtatására is. Abban az esetben, ha például egy osztálynak több leszármazott osztálya van, további osztályokkal különböző a kapcsolata, előfordulhat, hogy rengeteg él fut be. Ezek az élek mind 1 pontban kapcsolódnak a csomóponthoz, ami a megjelenítés átláthatóságát zavarja.

Felhasznált források:

- Kezdetben kapott kiinduló projekt
- MetaDslx framework dokumentációa
- Microsoft hivatalos oldalán a segédanyaok
- Szoftvertechnológia tárgy segédanyagai