HOGYAN HASZNÁLHATJUK FEL A VIZUÁLIS PROGRAMOZÁS (.NET C#) TANÍTÁSÁHOZ AZ UML-ALAPÚ MODELLEZÉST?

Szabolcsi Judit - Johanyák Zsolt Csaba
főiskolai adjunktus, főiskolai adjunktus
Kecskeméti Főiskola, GAMF Kar, Kalmár Sándor Informatikai Intézet

Absztrakt: Tapasztalataink szerint mérnök-informatikus hallgatóink a szoftverfejlesztés során kevés figyelmet fordítanak a tervezésre, modellezésre és módszeres megközelítésre legyen szó egy egyszerű órai feladatról, zárthelyi dolgozatról vagy éppen szakdolgozatról. Cikkünkben az eddig alkalmazott kétfajta gyakorlatvezetési módszer áttekintését és értékelését követően egy olyan új megközelítést mutatunk be, amely a Vizuális programozás néven futó, haladó szintű és komplex objektum-orientált feladatokat felvonultató programozás központú tantárgy és a tervezésre összpontosító Szoftvertechnológia összehangolása által próbál javítani a jelenlegi helyzeten.

BEVEZETÉS

Főiskolánkon a mérnök-informatikus hallgatók a mintatanterv szerint három félév C és C++ programozás után (a kreditrendszer miatt csak elvileg) a negyedik félévben kezdik a Vizuális programozás című tárgyat hallgatni. A tárgy heti két óra előadásból és két óra géptermi gyakorlatból áll. A tantárgyat csak a programozás szigorlat után lehet felvenni, ami lezárja a három félév C/C++-t. Szintén a szigorlat teljesítése után kezdhető el a Szoftvertechnológia nevű tárgy, ami a szoftverfejlesztés (Software Engineering) alapfogalmait és az UML diagramjait mutatja be. Célunk, hogy a két tárgyat összekapcsoljuk, és mind a vizuális programozásban, mind a szoftvertechnológiában elmélyítsük és gyakorlatiassá tegyük az oktatást.
A gyakorlatiasság hiánya főleg a Szoftvertechnológia tárgynál okoz gondot, mivel ez csak előadásból áll. Véleményünk szerint szükség lenne a gyakorlati alkalmazásra, hiszen a szoftverfejlesztést is tartalmazó szakdolgozatok elkészítésénél valamint a későbbi, esetleg programfejlesztési feladatokat is magába foglaló munkahelyi feladatok során sokaknak gondot okoz a módszeres megközelítés hiánya, valamint a felhasználói és fejlesztői dokumentáció elkészítése.
A Vizuális programozás oktatása során a Microsoft Visual Studio 2005-ös fejlesztőkörnyezetét használjuk, így eleve kézenfekvő, hogy az általa a programkódból automatikusan generált osztálydiagramot felhasználjuk. Innen már csak egy lépés, hogy bemutassuk, miért egyszerűbb jól megtervezett modellek alapján kódolni, mint egyből nekiesni a feladatnak.
Terveink között szerepel, hogy a két tárgy felvételét egymáshoz kössük, és amennyiben minden hallgató párhuzamosan teljesíti ezeket (jelenleg is sokan járnak párhuzamosan mindkét tárgyra, de nem mindenki), lehetőség lesz olyan 2-3 fős csoportokban teljesíthető félév végi gyakorlat- és vizsgajegy-szerző feladatok kiadására, amelyek lefedik a használati eset modelltől a kész kódig a szoftverfejlesztés teljes folyamatát.

A C# NYELV

A C# erősen típusos nyelv [1]. Ennek és még több más tulajdonságának (pl. a *System.Object*-ből kiinduló és minden osztályt magában foglaló, logikusan felépített osztály-hierarchiának, az automatikus kezdőértékadásnak, vagy a nem nyilvános adattagok elérésére használt tulajdonságoknak) köszönhetően szerintünk kezdőknek sokkal

praktikusabb programozási nyelv, mint a főiskolánkon jelenleg elsőként oktatott C/C++.
A C/C++ filozófiája azt támogatja, hogy a programozó döntse el mit akar csinálni, és a nyelv ehhez adjon szabad kezet és minél több eszközt. Ez egy profi programozó esetében hasznos is. Viszont egy elsőéves informatikus hallgató, akinek az ismeretei a játékprogramok, a chat és levelezés témakörében merülnek ki, nem tud mit kezdeni egy ilyen jellegű nyelvvel.
A C# filozófiája más, hatékony és mégis egyszerű, és több támogatást nyújt egy logikus nyelvhasználat elsajátításához. Ha megnézzük pl. az érték és a hivatkozási típusok megválasztásának módját, a következőket láthatjuk: a C++-ban minden típus érték típusú, viszont később „keresztbe-kasul" hivatkozhatunk rá, akár mutatókon, akár referenciákon keresztül. Ha érték szerint adjuk át a függvényparamétereket, és kapjuk meg a visszatérési értéket, az hatékony, de ha egy leszármazott típusú objektumot használunk az alaptípus helyett, akkor részleges másolás történik, így elveszítünk minden információt, amelyet az objektum „leszármazott része" hordozott.
Javaban pedig minden hivatkozási típusú, ami következetes, de rettentően lerontja a teljesítményt. Itt minden egyes változóért memóriafoglalással és szemétgyűjtéssel fizetünk. Beláthatjuk, hogy vannak olyan típusok, ahol felesleges a hivatkozási típust erőltetni, de Javaban ezek is hivatkozási típusúak lesznek, mert nincs más választási lehetőségünk.
A C# nyelvben választhatunk, hogy *struct* (vagyis érték típusú) vagy *class* (vagyis hivatkozási típusú) legyen az új típusunk. Ha a típusnak az adattárolás lesz a fő feladata, csak olyan tulajdonságai lesznek, amik az adattagok elérését és módosítását végzik, nem akarunk belőle leszármazottat készíteni és nem akarjuk többalakúként kezelni, akkor legyen értéktípus. Az alkalmazás viselkedésének meghatározására szolgáló típusok pedig legyenek osztályok [1] [2].
A nyelv egy másik kényelmes szolgáltatása a jellemzők (attribútumok) használata. A jellemzőkkel a „kijelentő stílusú" programozást (deklaratív) támogatjuk a „felszólító stílusú" programozás (imperatív) helyett. Nem kell tagfüggvényeket írni, amik meghatározzák a program viselkedését, hanem csak odaírjuk a [WebMethod], [Serializable] vagy a [NonSerialized] attribútumot az adott függvény, osztály vagy adattag elé és ezzel elérjük, hogy a .NET futásidejű környezet hozzátegye a megfelelő viselkedést helyettünk. Ez könnyebben megvalósítható, olvasható és karbantartható kódot eredményez. A létező jellemzők mintájára sajátokat is létrehozhatunk, és ezek valamint a visszatekintés felhasználásával mi magunk is írhatunk kijelentő stílusú programszerkezeteket [1].

A C # ÉS A .NET OKTATÁSA

A Vizuális programozás tantárgy keretén belül a nyelv fentebb említett alapjaitól indulva gyorsan eljutunk a Windows Form-os alkalmazásokhoz, az ADO.NET-hez, a sorosításhoz és a webszolgáltatásokhoz. Ezeknek a témáknak a bemutatásához nyilván összetettebb példaprogramok szükségesek.
Két stratégiát próbáltunk ki ezeknek az programoknak a hallgatók számára érthetővé tételére: az első alapján minden egyes példaprogramhoz egy 8-10 oldalas leírás tartozik, amely lépésről lépésre, sok képernyőképpel illusztrálva leírja, illetve bemutatja az elkészítés menetét (beleértve azt is, hogy a Visual Studio mely részeit és hogyan használjuk,
pl.: „Felület kialakítása
A főablak neve legyen Name=UgraloGombForm, az őt definiáló állomány nevét is változtassuk meg UgraloGombForm.cs-re. Solution Explorerben jobb egérgomb a Form1.cs-n, majd a név átírása.").
Ebben az esetben a hallgatók papíron megkapják ezt a leírást, és egyéni munkában

végigcsinálják a leírtakat.
Ennek a módszernek az az előnye, hogy a hallgató nem marad el a többiektől, a saját tempójában dolgozhat, bármikor vissza tud lapozni, és a későbbiekben bármikor végigcsinálhatja a gyakorlatot. A gyakorlatvezető az óra elején ismerteti a feladatot, a javasolt megoldás fő elemeit, majd a későbbiekben segítséget nyújt az egyéni tempóban haladó hallgatóknak egyenként.
A módszer hátránya az, hogy egyrészt az anyagok elkészítése a tanár igen sok idejét felemészti, másrészt pedig a hallgatók nem igazán fogják fel, hogy mit csinálnak, nem olvassák el az áttekintő bevezető részt, ami a tervezési megfontolásokat taglalja, csak mechanikusan gépelnek vagy másolnak (CTRL+C/CTRL+V), amennyiben elektronikus formában is rendelkezésre áll a dokumentum.
A második stratégia szerint a programkódot a tanár a táblára írja fel, és közben felhívja a figyelmet a hangsúlyosabb elemekre, osztályokra, azok használatának buktatóira.
Ennek előnye, hogy interaktívabb, jobban kiderül, hogy mit nem ért a csoport; hátránya viszont, hogy elsősorban a másolásra koncentrálnak, és a lassabban gépelők elmaradnak, illetve a gyakorlatvezetőnek nem jut elegendő ideje arra, hogy külön-külön foglalkozzon az egyes hallgatókkal.
Az eddigi tapasztalataink szerint egyik fentebb vázolt módszer sem váltotta be teljes mértékben a hozzá fűzött reményeket. Emiatt merült fel egy harmadik megközelítés, amit az utolsó fejezetben fejtünk ki. [3]

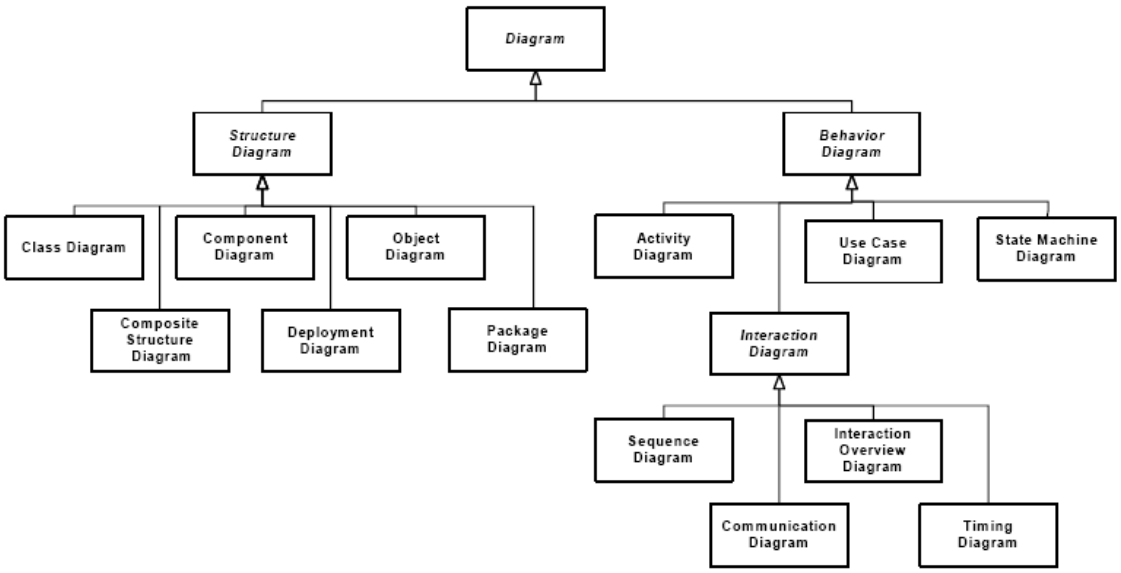
SZOFTVERFEJLESZTÉS ÉS UML

A hazai oktatási rendszerben sokszor elfelejtkezünk arról, hogy a „programozni tudás” nem azonos azzal, hogy „képesnek lenni a kódolásra”. A végzett mérnök-informatikusoknak azzal is tisztában kell lenniük, hogy a szoftver egy termék, tehát mint minden termék előállításához, ehhez is technológiára van szükség. Emellett a terméknek tervezési paraméterei is vannak: mire képes (szolgáltatási funkció), milyen a minősége, mekkora az előállítási költsége és mi az elkészítés határideje.
A technológiának éppen azt kell biztosítania, hogy a kész termék megfeleljen a tervezési paramétereknek [4].
A szoftvertermék dokumentálása szintén egy kardinális kérdés, amire a mostani tantárgystruktúrában nem igazán fordítunk figyelmet. Ez hamar meg is bosszulja magát, rögtön a szakdolgozat-készítésnél, majd később az iparban, amennyiben valaki fejlesztői munkakörben helyezkedik el [5].
Az UML (Unified Modeling Language) elsajátítása mind a tervezést, mind a dokumentálást megkönnyíti, és emellett szabványos és nyílt is [6]. Az UML-re épül az MDA (Model Driven Architecture), amely elkülöníti az üzlet-orientált és a platform-függő döntéseket, mivel egy MDA specifikáció egy platform-független UML modellből (Platform Independent Model – PIM) és egy vagy több platform-specifikus modellből (Platform Specific Model – PSM) áll.
Egy PIM tulajdonképpen teljes alkalmazás specifikáció, amely platform-független. Egy PIM egy PSM-be képződik le, amely a rendszer-architektúra infrastruktúráját biztosítja. Ez a leképezés a PIM implementációját jelenti, azaz a végrehajtható alkalmazás generálását. A PIM lehetővé teszi számunkra a megoldás vizuális modellezését magas absztrakciós szinten. Szükségtelen az alkalmazás újraírása egy új technológia megjelenésekor. Mindössze újra kell generálni az alkalmazást, azaz elvégezni a modell leképezését az új környezetbe. Sok népszerű technológiai platform, mint a CORBA, J2EE vagy .NET számára lehetséges PIM→PSM leképezést definiálni.
Ezzel elérhetjük a régóta óhajtott célt – a modellből automatikusan kód generálható,

emberi közreműködés nélkül!
Térjünk vissza még egy kicsit az UML nyelvhez. Az UML „csak” egy nyelv és nem módszertan, ahogy az a fentiekből már kiderült.
Az UML 2.0 (az UML jelenleg aktuális verziója) a rendszer viselkedését és a szerkezetét egyaránt képes modellezni. A szerkezeti modellt a következő diagramtípusok alkotják:
- osztálydiagram (Class Diagram)
- objektum diagram (Object Diagram)
- telepítési diagram (Deployment Diagram)
- csomag diagram (Package Diagram)
- komponens diagram (Component Diagram)
- Composite Stucture Diagram (új, az UML 1.x-hez képest)

A rendszer viselkedését pedig a következőkkel lehet modellezni:
- szekvencia diagram (Sequence Diagram)
- kommunikáció (együttműködési) diagram (Communication Diagram)
- aktivitás diagram (Activity Diagram)
- interakció áttekintés diagram (Interaction Overview Diagram)
- állapotgép diagram (Statemachine Diagram)
- idődiagram (Timing Diagram)
- használati eset diagram (Use Case Diagram)

A felsorolt diagramtípusok közötti összefüggést is megadhatjuk UML jelöléssel, ezt mutatja az 1. ábra.



1. ábra
Az UML 2.0 diagramfajtái – maga az ábra is egy osztálydiagram

Az UML 2.0-nak is nyilván vannak hiányosságai, gyengeségei, nem ez az „univerzális csodaszer”. A nyelv továbbra sem rendelkezik formális definícióval, még az alapelemek esetében sem, pedig ez volt a régebbi UML verzióban a leginkább kritizált pont. Az UML használhatósága nem nőtt azzal, hogy több új eleme lett és újabb diagramok jelentek meg.
A diagramok között sokszor átfedés figyelhető meg. Az objektum diagram ugyanazt

ábrázolja, mint a kommunikációs diagram, kivéve az üzenetek áramlását. Az interakció áttekintés diagram pedig az aktivitás diagram egy speciális esete.
A modellt alkotó diagramok közötti függőség sincs explicit módon definiálva, a felelősséget a modellezőre vagy az eszközre hagyva, hogy ellenőrizze, valóban konzisztens modell jött-e létre. Így viszont a modell tesztelése okoz gondokat.

A SZOFTVERTECHNOLÓGIA ÉS A VIZUÁLIS PROGRAMOZÁS ÖSSZE-KAPCSOLÁSA

Véleményünk szerint a hallgatók a különböző tárgyak órái alatt sok hasznos és korszerű eszközt megismernek, viszont arra már nincs idő, hogy azt is megmutassuk nekik, ezeket mire tudják használni. Ezért „nem áll össze" bennük a kép, nem tudják, hogy mi mire jó, mikor lenne értelme használni és mikor értelmetlen vele próbálkozni, csak azt az egy konkrét felhasználási lehetőséget képesek elképzelni, amiben ők találkoztak vele. Ez komoly hiányosság.
Ha konkrétan a fentebb említett két nagy témakört nézzük, a következő kérdés merül fel. Mi értelme modellezni? Nyilván nem a modellezésért önmagáért tesszük, ahogy a hallgatók a Szoftvertechnológia tárgy során látni vélik. Viszont ha felhasználnánk egy-egy bonyolultabb (C# nyelvű) program megtervezésére az ott tanult módszereket, akkor a gyakorlatban is látnák a hasznát – onnantól kezdve nem csak programsorok begépelése és egérrel történő kattintgatás lenne a „programfejlesztés".
Ahogy Maksimchuk és Naiburg írja:
„Nézzük más szempontból: van olyan, aki megépítene egy házat tervrajz nélkül? Lehet, hogy nem építjük meg előbb a ház kisméretű változatát, de biztosan a kezünkben (vagy az építész, illetve az építő kezében) lennének a tervrajzok, építészeti tervek és a mérnöki vélemények, mielőtt elkezdenénk az építkezést.
A házak ugyanolyanok, mint a programok. […]
A programtervezés hasonló kihívást jelent. Biztosítanunk kell, hogy a tervek és a rajzok megvalósítása végül megfelelő, felépítésük pedig időtálló legyen, továbbá arról is gondoskodnunk kell, hogy ha változtatásokat kell végeznünk, azok is tartósak legyenek. Azzal is tisztában kell lennünk, hogy milyen módosítások borítják fel a tervet. Ha egy új összetevővel bővítjük a rendszert, ügyelnünk kell, hogy az ne omoljon össze ennek eredményeként." [7]
Az UML alapú modellezésre összpontosító Szoftvertechnológia és a C# nyelvű Vizuális programozás összehangolásával kapcsolatos elképzeléseinket az alábbi pontokban foglatuk össze.

1. Kötelezővé tesszük a két tárgy párhuzamos felvételét hallgatóink számára. Mivel a tanterv kialakításában a fentieken túl még sok más szempont is szerepet játszik, valószínűleg ez nem a legkönnyebben megvalósítható célok közé tartozik.
2. A Szoftvertechnológia előadásokon olyan modellezési példákat mutatunk be, amelyek C# nyelvű és vizuális módszerekkel történő implementációjára a Vizuális programozás gyakorlatokon kerül sor.
3. Néhány Vizuális programozás gyakorlaton a hallgatókkal közösen lépésről-lépésre végigmegyünk a teljes tervezési-implementálási folyamaton.
4. Olyan projekt jellegű féléves nagyfeladatokat adunk ki 2-3 fős hallgatói csoportoknak, amelyben a megadott specifikációból/megrendelői igényekből és megadott módszertant követve kell elkészíteniük és dokumentálniuk a szoftvert. Ezen feladatok sikeres megoldása mindkét tárgyból érdemjegyet eredményezne.

ÖSSZEFOGLALÁS

Mérnök-informatikus hallgatóink programozási gyakorlatának vizsgálata és a programfejlesztést tartalmazó szakdolgozatok konzultációs tapasztalatai ahhoz a felismeréshez vezettek, hogy változtatnunk kell eddigi oktatási módszereinken és gyakorlatunkon. Elképzeléseink kialakítása során az a cél vezérelt bennünket, hogy idejében rávezessük a hallgatókat a tervezés szükségszerűségére, és javítsuk ezen ismeretek elsajátításának hatékonyságát.
Négy pontban összefoglalt elképzeléseink alapgondolata az, hogy a jelenleg túlságosan elvontnak és elméletinek tartott szoftvertechnológiai ismereteket nyújtó tantárgyat egy gyakorlatközpontú, haladó szintű feladatokkal foglalkozó programozási tárggyal hangoljuk össze, és emellett olyan átfogó jellegű projektfeladatokat tűzünk ki hallgatóinknak, amelyek megoldásához mindkét témakör ismerete szükséges.

IRODALOM

[1] WAGNER, B.: **Hatékony C#. Budapest**. Kiskapu Kiadó, 2005. 1-40. old., 135-139. old.
[2] ALBERT, I. (szerk.): **A .NET Framework programozása**. Budapest, Szak Kiadó, 2004. 37-62. old., 312-352. old.
[3] SHARP, J.: **Microsoft Visual C# 2005 lépésről lépésre**. Budapest, Szak Kiadó, 2005.
[4] SIKE, S. – VARGA, L.: **Szoftvertechnológia és UML**. Budapest, ELTE Eötvös Kiadó, második, bővített kiadás, 2003. 13-17. old.
[5] SIKE, S. – VARGA, L.: **Szoftvertechnológia és UML**. Budapest, ELTE Eötvös Kiadó, második, bővített kiadás, 2003. 59-62. old.
[6] http://www.omg.org, **Unified Modeling Language**: Superstructure. Version 2.1.1 2007-02-03
[7] MAKSIMCHUK, R. A. - NAIBURG, E. J.: **UML földi halandóknak**. Budapest, Kiskapu Kiadó, 2006. 7-8. old.