Tapasztalatok a programozási feladatok visszavezetéssel történő megoldását támogató osztály-sablon könyvtár használatáról

Gregorics Tibor, Nagy András

gt@inf.elte.hu, nagyandras95@inf.elte.hu
ELTE IK

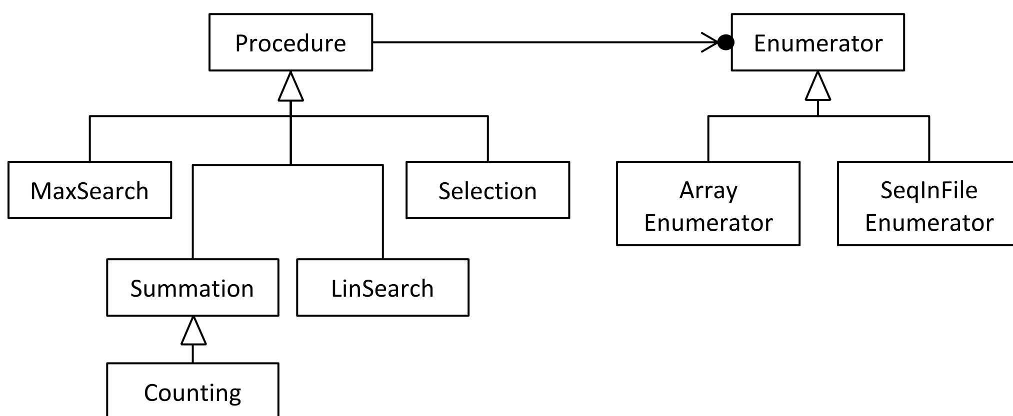
Absztrakt. Eltelt néhány év azóta, hogy az ELTE IK PTI BSc szakon a programozás oktatás része lett annak az újrahasznosítható osztály-sablon könyvtárnak az ismertetése és felhasználása, amely a programozási tételekre történő visszavezetéssel tervezett programok C++ nyelvű megvalósítását támogatja. Ebben a cikkben az oktatásba történő bevezetéssel kapcsolatos tapasztalatokról számolunk be, és bemutatjuk az ezek hatására módosított osztály-sablon könyvtárban található újításokat.

Kulcsszavak és kifejezések: visszavezetés, programozási tétel, felsoroló, objektum orientált programozás

1. Bevezetés

2009-ben került bevezetésre az ELTE IK programozó informatikus szak objektum-elvű alkalmazások fejlesztése kurzusának tananyagába egy olyan osztály-sablon könyvtár, amelynek segítségével a programozási tételekre visszavezethető feladatok C++ nyelvű megoldását lehet elkészíteni. A könyvtár és annak használata elsősorban objektum-orientált technológiára épült [1], de sablon-paramétereket is használt.

A könyvtárban három féle osztályt találunk: a felsorolót használó programozási tételeket [2] általánosan leíró osztály-sablonokat és ezek ősosztályát (*Procedure*); néhány nevezetes felsorolást [4] definiáló osztályt és ezek ősosztályát (*Enumerator*), valamint két segédosztály a maximum keresés tételének maximumot, illetve minimumot kereső változatainak előállításához. [3]



1. ábra. Programozási tételek osztály-sablon kód-könyvtára

Nemcsak a könyvtár felhasználása épül objektum-orientált technológiára, de maga a könyvtár is ennek szellemében készült. A *Procedure* osztály-sablon az összes programozási tétel őse, amely egy általános feldolgozási stratégiát fogalmaz meg: nevezetesen, végig megy egy felsoroló (erre mutat az *enor* pointer) által előállított elemeken, hogy azokat feldolgozza.

```
init();
for (enor->first(); !enor->end(); enor->next()) {
    body(enor->current());
}
```

Ezt a feldolgozást a *Procedure* ősosztály *run()* metódusa tartalmazza, és végsősoron ezt hajtja végre minden programozási tételre visszavezetett megoldás is. A végrehajtás attól válik egyedivé, hogy az egyes programozási tételeket leíró osztályok egyedi módon implementálják (felülírják) az *init()* és a *body()* virtuális metódusokat. Mindeközben újabb, a konkrét tételre jellemző virtuális metódusokat vezetnek be, amelyeket a konkrét alkalmazásoknál kell/lehet felüldefiniálni.

Egy programozási tételre visszavezethető feladat megoldása ezen könyvtár megfelelő programozási tételt leíró osztály-sablonjának felhasználásával készül. Előbb megadjuk ennek sablon-paramétereit helyettesítő értékeket, majd egy egyedi osztályt származtatunk belőle, és az így nyert osztály objektumpéldányára meghívott (öröklött) *run()* metódus oldja meg a feladatot. Ennek a tevékenység-objektumnak az egyedi viselkedését egyrészt (fordítási időben) a származtatás során felülírt virtuális metódusok biztosítják, másrészt annak a speciális felsoroló objektumnak hozzáadása (futási időben) az (öröklött) *addEnumerator()* metódus segítségével (függőség befecskendezés), amelyik a megoldás során feldolgozandó elemeket sorolja fel.

A felsorolókat [2] [3] leíró osztályok őse egy interfészsablon (*Enumerator*), egy teljesen absztrakt osztály, amely bevezeti, de nem definiálja a felsoroló műveleteket. Ebből származnak a konkrét felsorolók osztályai, amelyek már reprezentációt és implementációt is tartalmaznak.

Ebben a cikkben ennek az osztály-sablon könyvtárnak a bevezetésével kapcsolatos tapasztalatokat gyűjtöttük össze, valamint a negatív kritikák nyomán végrehajtott módosításokat, amelyekkel sikerült megújítani a könyvtárat.

2. Korábbi verzióval kapcsolatos tapasztalatok

Az osztály-sablon könyvtárnak a bevezetése a programozás egyetemi tanításába az eredetileg kitűzött célon túl, miszerint az objektum-orientált programozást gyakoroltató újrafelhasználható könyvtárra volt szükség, több előnnyel is járt.

1. Nagyszámú olyan feladatot tudunk megfogalmazni, amelyek a könyvtár segítségével megoldhatók. Hosszú listája született a gyakorló feladatoknak, házi feladatoknak és zárthelyi feladatoknak. A könyvtár igen robosztus lett. Még a nem kifejezetten programozási tételekre történő visszavezetéssel megoldható feladatok is megoldhatók a segítségével (pl. láncolt lista felépítése a dinamikus memóriában, láncolt lista elemeinek feldolgozása).
2. A könyvtár használata bemutatja az objektum-orientált technológiákat, mint az egységbe zárást, az elrejtést, az öröklődést, a futási idejű polimorfizmust, függőség befecskendezést, de jó példákat szolgáltat a tervezési mintákra is. Ezen kívül a sablonok (*template*) has**ználatával is megismertet. Mivel maga a könyvtár is osztály-hierarchiára épül, jól** megérthető vele az objektum-orientált újrafelhasználási technika is, valamint a megvalósításhoz szükséges C++ nyelvi elemek (*class*, *virtual*, *protected*, …) is.

3. A könyvtár használata elmélyíti a programozási tételekre történő visszavezetés módszerének megértését és alkalmazását, azaz azt, amikor a programozási tétel nem csak egy laza ajánlás az algoritmikus gondolkodás során létrehozandó kódhoz, hanem egy sablon, amelynek kitöltésre váró paraméterei vannak, és ezáltal a programhelyességet garantáló módszer. Ezzel kapcsolatban mindenképpen pozitív változást hozott, hogy a 2017-es tantervváltozás során közös tantárgyba került a visszavezetéses programozási módszer és az objektum elvű programozás tanítása, és ezekkel együtt az osztály-sablon könyvtár használata.
4. A könyvtárra támaszkodó programozással sikerült rámutatni az elemzés és tervezés fontosságára már az egyszerű programozási feladatok megoldásánál is. Kikényszeríti, hogy egy feladat megoldásánál ne azon gondolkodjon a hallgató, hogy hol írjon ciklust, hogyan olvassa be a bemenő adatokat, hanem azon, hogy milyen programozási tétellel oldható meg a feladat, ehhez milyen felsorolást kell biztosítani, melyek lesznek a programozási tétel paraméterei, miközben az újrafelhasználható elemek implementálásával és tesztelésével nem kell foglalkoznia.
5. A könyvtárra támaszkodva szép megoldásokat tudunk előállítani. Programozói szemmel például nagyon érdekes, hogy az előállított megoldásokban mindössze egyetlen ciklus lesz, mégpedig a programozási tételek ősosztály-sablonjának *run()* metódusában. Ez kerül felhasználásra mindenféle paraméterezés mellett, a saját kódban pedig egyáltalán nem kell ciklust írni.

Természetesen a könyvtár oktatásba történő bevezetése negatív kritikákat is kapott.

1. A hallgatóknak nem tetszett, hogy rájuk kényszerítünk egy számukra idegen kódolási stílust (például, hogy ciklust egyáltalán nem írhatnak), amelyet az alkotói szabadságukba történő beavatkozásként éltek meg.
2. Gyakran hangoztatott kifogás, hogy a könyvtár nem életszerű feladat-megoldást támogat, ipari alkalmazásokban ilyen könyvtárat nem használnak, jobb lenne egy valódi könyvtár használatát tanítani helyette.
3. Az egyszerű feladatok megoldása túl erős technológiákra épül (származtatott osztályok, futási idejű polimorfizmus, függőség befecskendezés, tervminták), és a kapott kód mérete nem lesz kisebb a feladat elemi eszközökkel előállított megoldásánál. Például egy számlást megoldó ciklussal szemben itt egy megoldó objektumot kell egy olyan osztályból példányosítani, amelyet a számlálást leíró osztály-sablonból származtatunk, de felül kell írni a számlálás feltételét adó metódust, majd ehhez a megoldó objektumhoz egy felsoroló objektumot kell kapcsolni, és végül meghívni rá a run() metódusát.
4. A könyvtár helyes használatát csak az arra vonatkozó korlátozások betartatásával lehet kikényszeríteni. A hallgatók igencsak kreatívnak bizonyultak a könyvtár nem szakszerű használatában (a run() metódust felülírták; egy programozási tételből származtatott osztályban új adattagokat vettek fel, és ezeket módosították a felüldefiniált metódusokban, rekurzív hívásokat alkalmaztak stb.). A könyvtár helyes használatát egy évről évre bővülő tiltásokat tartalmazó szabály kódexszel próbáltunk kivédeni.
5. Nem sikerült jól az összegzés programozási tételének osztálya. Ugyanis a tétel igen robosztus, nagyon sokféleképpen lehet paraméterezni, és ezt a szabadságot olyan osztállyal sikerült biztosítani, amely különösen sok lehetőséget adott a hallgatóknak ahhoz, hogy a konvenciókat felrúgva, ne programozási tételekre épülő megoldásokat készítsenek.

3. Új elemek az osztály-sablon könyvtárban

A hallgatói visszajelzések, az oktatás és számonkérés alapján szerzett tapasztalatok után felmerült az igény a könyvtár megújítására. Ehhez felhasználtuk az új C++ nyelvi szabvány elemeit, melyek részben javították a könyvtár kódjának minőségén, részben pedig a könyvtár szakszerű használatát

mozdították elő. Azt a szabályrendszert, melyet a zárthelyiknél eddig ki kellett kötni és külön kellett ellenőrizni, most olyan nyelvi elemekkel biztosítottuk, melyeknek köszönhető a C++ fordító garantálja a szabályok betartását a könyvtár használata során.

3.1. Az összegzés programozási tételre épülő osztály módosításai

Az összegzés programozási tételét leíró osztály korábbi változata túl általánosra sikerült. Például eredetileg a *body()*-t alkotó *add()* művelet nem volt konstans, ezért meglehetősen szabadon lehetett felüldefiniálni. Akár egy tetszőleges algoritmus is leírható volt vele, ha megfelelő adattagokat vettünk fel a *Summation* osztályból származtatott osztályba, és az *add()* függvény felüldefiniálásakor, amely gyakorlatilag az ősosztály ciklusmagját helyettesítette, bármit csinálhattunk ezekkel az adattagokkal.

```
#pragma once

#include "procedure.hpp"
#include <iostream>

template < typename Item, typename Value = Item >
class Summation : public Procedure<Item, Value>
{
private:
    Value _result;
protected:
    void init() final override { _result = neutral(); }
    void body(const Item& e) final override {
        if(cond(e)) _result = add(_result, func(e));
    }
    virtual Value func(const Item& e) const = 0;
    virtual Value neutral() const = 0;
    virtual Value add( const Value& a, const Value& b) const = 0;
    virtual bool  cond(const Item& e) const { return true; }
public:
    Summation(){}
    Summation(const Value &v) : _result(v) {}
    Value result() const { return _result; }
};
```

2. ábra. Összegzés programozási tételt leíró új osztály-sablon

Hiányzott a korábbi *Summation* osztályból az összegzés programozási tételnek [2] azon függvénye, amely az aktuálisan felsorolt elemeket az eredmény típusára képezi le. Ezt az *add()* művelet felülírásának keretében kellett a felhasználónak megadnia.

A korábbi változatban az *init()*-et nem a *Summation* osztály definiálta felül, hanem az abból származtatott konkrét osztályban kellett azt a felhasználónak megadni, pedig az összegzés programozási tétele szerint kezdetben kizárólag az eredményt kell beállítani a neutrális értékre.

Az új változatban (2. ábra) a *Summation* osztály definiálja felül a *Procedure* osztály *init()* és *body()* metódusait, és ennek konkrét leszármazottjaiban kell majd megadni a felsorolt elemhez értéket rendelő *func()* leképezést, a *neutral()* függvényben a neutrális értéket, valamint a két érték összeadását végző *add()* műveletet.

Ettől a módosítástól azt várjuk, hogy a hallgatók a *Summation* használata során kizárólag az általunk elvárt módszert, a megtanult visszavezetési technikát tudják csak alkalmazni.

Az összegzés kódja számos új nyelvi módosításon is átesett, melyek nem kötődnek szorosan az összegzés módosításaihoz, így majd a későbbiekben fogjuk kielemezni őket.

3.2. Új felsoroló típusok bevezetése

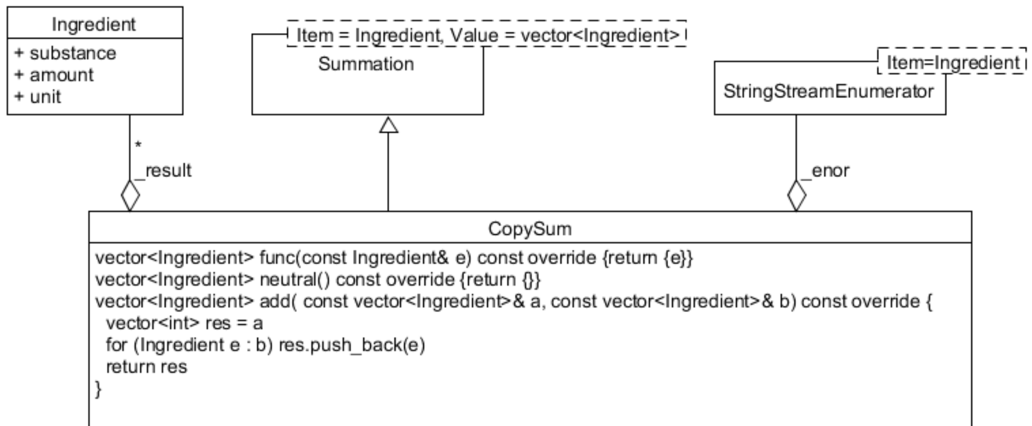
A könyvtár korábbi változata nem támogatta azon feladatok megoldását, amelyekben szöveges állományokat soronként kell feldolgozni (például felfűzni egy-egy sor adatait egy listába), de az egy sorban található adatok száma változó. Az ilyen feladatokat csak ciklus beépítésével lehetett megoldani úgy, hogy a sort szövegfolyammá alakítottuk, és addig olvastunk ebből, míg vége nem lett az adott sornak. Ez a megoldás szembe ment azzal az elképzeléssel, hogy tiltsuk meg a ciklusok használatát, mert ez hatékony kényszerítő eszköz ahhoz, hogy a hallgatók a megoldásaikban a programozási tételekre történő visszavezetésre támaszkodjanak. Ennek a dilemmának a feloldása kézenfekvő: mivel egy sor feldolgozását is valamelyik programozási tételre lehet visszavezetni, csak egy olyan speciális felsorolóra van szükségünk, amely szövegfolyam elemeit képes bejárni. Ezért vezettük be a *StringStreamEnumeratort*. A forráskód [10]-ben megtekinthető.

Nézzünk meg egy példát a *StringStreamEnumerator* használatára: Tekintsük azt a feladatot, amelyben egyetlen sor szövegesen (tehát sztringként) megadott adatait szeretnénk eltárolni egy listában (C++-ban vectorban). Az adatok egy étel receptjében szereplő összetevők. Minden összetevő három részből áll: az összetevő anyagának neve (sztring), a recepthez szükséges mennyiség (int), és annak mértékegysége (sztring).

A megoldáshoz (azaz a listába történő összefűzéshez, ami egy másolás) az összegzés programozási tételét használjuk egy olyan felsorolóval, mely egy szövegből sorolja fel egy recept összetevőit. Először létre kell hoznunk a recept egy összetevőjét leíró *Ingredient* struktúrát, és ehhez definiálni kell egy beolvasó operátort:

```
struct Ingredient {
    std::string substance;
    int quantity;
    std::string unit;
};
std::istream& operator>>(std::istream& in, Ingredient &e)
{
    in >> e.substance >> e.quantity >> e.unit;  return in;
}
```

Ezután egy saját *CopySum* osztályt kell származtatnunk a 3. ábra szerint a Summation osztály-sablonból.

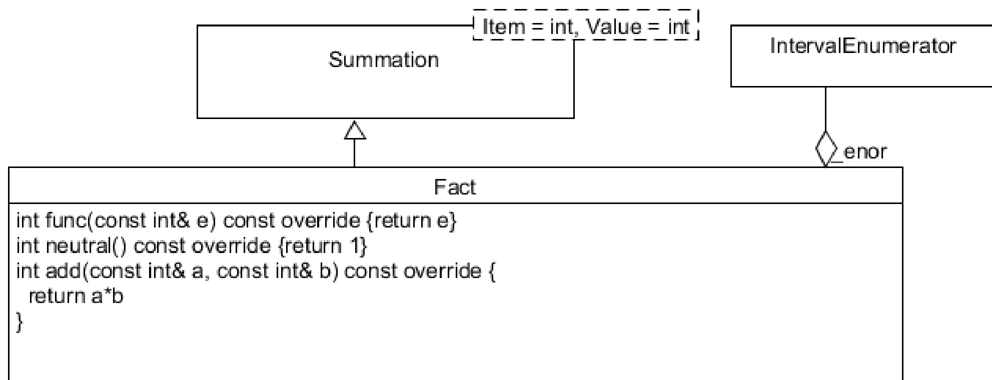


3. ábra. Összetevők összegyűjtésének osztálydiagrammja

Végül az alábbi kódot kell végrehajtani:

```
CopySum sum;
StringStreamEnumerator enum(std::stringstream(
  "tej 1 liter búzadara 13 evőkanál vaj 6 dkg cukor 5 evőkanál"));
sum.addEnumerator(&enum);
sum.run(); // sum.Result() visszaadja az eredmény vectort
```

Egy másik nevezetes felsoroló osztály is beépült az új osztály-sablon könyvtárba. Az *IntervalEnumerator* segítségével, egész számok tetszőleges intervallumának elemeit tudjuk felsorolni [10]. Ennek létrehozásához csak az intervallum elejét és végét kell megadni.



4. ábra. Faktorális kiszámolásának osztálydiagrammja

Erre a felsorolóra van szükségünk például akkor, amikor egy *n* természetes szám faktoriálisát számoljuk ki. Ehhez ugyanis elég az egész számokat kell felsorolni 2-től *n*-ig, és közben a felsorolt számokat összeszorozni. Ez utóbbi visszavezethető az összegzés programozási tételére, ahol az *add()*

művelet speciálisan két egész számot szoroz össze, a neutrális elem pedig az 1 lesz. Jól illusztrálja ez a példa az összegzés robusztusságat [5].

```
Fact sum;
IntervalEnumerator enum(2,n);
sum.addEnumerator(&enum);
sum.run();    // sum.Result() visszaadja a faktoriális értéket
```

3.3. Final kulcsszó bevezetése virtuális metódusoknál.

A korábbi verzióban probléma volt, hogy a hallgatók olyan metódusokat akartak felül definiálni, melylyel felborították a sablon metódus tervminta alapján kialakított könyvtárat, és eltértek a programozási tételekkel történő konvencionális megoldástól. Ezt a problémát tudja kiküszöbölni a *final* kulcsszó, mellyel a fordító biztosítani tudja, hogy mely metódusuk felüldefiniálása nem megengedett.

A *final* kulcsszót virtuális metódusoknál használhatjuk C++11-től kezdve [8], amely azt fejezi ki, hogy az adott metódust a leszármazott osztályokban már nem definiálhatjuk felül. Például a *Procedure* osztály *run()* metódusa, vagy a *Summation* osztály *init()* és *body()* metódusai előtt láthatunk ilyet. Ez biztosítja, hogy a *run()* metódus mindig az ősosztályban megadott algoritmus sémát hajtsa vége, illetve az összegzés mindig az inicializáló értékadással kezdődjék, a ciklusmag pedig csak a *_result* értékét változtathassa meg, és azt is csak a megadott módon.

3.4. Öröklődés korlátozása a barát osztályokra

A korábbi verziókban lehetőség nyílott arra, hogy a hallgatók elkerüljék, hogy programozási tételekkel oldják meg a kitűzött feladatot, mert a *Procedure*-ből közvetlenül lehetett egy saját algoritmust leíró osztályt készíteni. Ezt ugyan tiltottuk, de manuálisan kellett ellenőrizni.

Ha viszont a *Procedure* ősosztályt egyetlen privát konstruktorral látjuk el, akkor ezzel megakadályozhatjuk, hogy új osztályokat lehessen ebből közvetlenül származtatni, hiszen a leszármazott osztályok példányosítása a *Procedure* konstruktorát is igénylik. De hogyan lehet a könyvtárban azokat az osztályokat definiálni (*Summation*, *LinSearch*, *MaxSearch*, stb.), amelyeknek valóban a *Procedure* leszármazottjainak kell lenni? Ezeket az osztályokat barát osztályként jelöljük meg (`friend Summation<Item, Value>`), így számukra hozzáférést biztosítunk a *Procedure* konstruktorához.

Ez a fordítás idejű korlátozás megoldást ad a felvetett problémára, elősegíti a könyvtár szakszerű használatát, rákényszerítve a hallgatókat, hogy valamelyik nevezetes programozási tétellel dolgozzanak, és ne használják közvetlenül a *Procedure* ősosztályt.

3.5. Override kulcsszó bevezetése virtuális metódusoknál

Az *override* kulcsszót a virtuális metódusoknál használhatjuk a C++11-es szabványban [8], mely azt a szándékot fejezi ki, hogy egy virtuális metódust szeretnék felüldefiniálni. Ha elrontjuk a felüldefiniálandó metódus szignatúráját, vagy az adott metódus *final* az ősosztályban, akkor fordítási hibát kapunk. Például az *init*() és a *body()* metódusnál jelezzük a Summation-ban, hogy az *Procedure* osztálybéli megfelelő virtuális metódust felüldefiniáltja.

Ennek didaktikai szempontból két előnye van. Egyrészt láthatóvá válik a könyvtárban, hogy mikor definiálunk felül egy metódust, másrészt, ha rászokunk ennek használatára a saját osztályaikban is, akkor fordítási időben kijönnek az olyan hibák, mikor valaki elront egy szignatúrát a felüldefiniálás során. (pl. kihagy egy *const* kulcsszót a paraméterlistából)

3.6. Template specializáció használata

Az osztály-sablon könyvtár segítségével egyedi felsorolókat is lehet definiálni, de néhány nevezetes felsoroló használatát közvetlenül is támogatja. Ezek közül az egyik a fájlfelsoroló (*SeqInFileEnumerator*). Ez egy szöveges állomány azonos típusú adatait képes felsorolni. Az egyetlen elvásárunk felé az, hogy létezzen az a beolvasó operátor, amely egy adatot olvas be. Ez az olvasás általában figyelembe veszi az elválasztó jeleket (szóköz, tabulátor jel, sorvége jel), de amikor a szöveges állomány tartalmát karakterenként kell olvasni (azaz egy adat egy karakter), akkor többnyire az elválasztó jeleket is be kell olvasnunk. Ezért a karakterek olvasásához a fájlfelsoroló működését specializálnunk kell.

Korábban ez egy futás idejű ellenőrzéssel volt megoldva. Megvizsgáltuk, hogy a beolvasandó adat típusa karakter-e, és ennek megfelelőnek állítottuk be az olvasó operátor működését. Az új verzióban ezt *template* specializációval oldottuk meg. [9]

A *template* specializáció az a mechanizmus, amikor egy sablon osztályt vagy annak egy metódusát bizonyos típusokra teljesen másképp kezelünk, mint ahogy azt az általános sablon definícióban leírtuk. Mivel ez sablon specializáció fordítás idejű ellenőrzés, így ez egy szebb konvenció, ráadásul a hallgatók is találkozhatnak ezzel az új fogalommal a könyvtárral való ismerkedés során.

A konkrét megvalósításban a *SeqInFileEnumerator*-hoz hozzáadtunk egy létrehozó metódust. Ezt a metódust specializáltuk arra az esetre, amikor a felsoroló karaktereket sorol fel, ilyenkor a fájl létrehozó metódus definícióját kiegészítettük az olvasó operátor átállításával arra, hogy az elválasztó jeleket se ugorja át.

A template specializáció hasznos mechanizmusnak bizonyult a *Summation* definíciójánál is.

Az összegzés tételének ugyanis gyakran előforduló speciális alkalmazásai a másolás, ki- vagy szétválogatás, összefuttatás jellegű feladatokat megoldása [5]. Ilyenkor az eredmény egy sorozat, ezért az összeadás műveletét a sorozatok összefűzése helyettesíti, a neutrális elem pedig az üres sorozat. A gyakorlati feladatokban azonban ezt az eredményként keletkező sorozatot a szabványos kimenetre vagy egy szöveges állományba szokták kiírni, azaz a *Value* sablon-paraméter az *std::ostream*. Az ilyen esetekben hatékonyabb, ha átadjuk az összegzésnek a kiírás helyét jelző *ostream* típusú referencia értéket (például *cout*), hogy az összegzés közvetlenül ide helyezze el az eredmény-sorozat elemeit, ne egy ideiglenes sorozatba. Ekkor viszont nincs értelme az eredményt (*ostream* típusú referencia értéket) inicializálni, emiatt az *init()* törzsét üresen kell hagyni, és ennél fogva a *neutral()* metódusra nincs szükség. Az eredményt, azaz a *Summation std::ostream* típusú *_result* adattagjának új értékét közvetlenül a `_result << func(e)` utasítással állíthatjuk elő, és ezt a *body()* törzsében kell elhelyezni az *add()* hívása helyett. Tehát az *add()* metódusra sincs szükség.

Ezt a speciális esetet írja le a *Summation* 5. ábra szerinti változata.

```
template < typename Item >
class Summation<Item, std::ostream>
                      : public Procedure<Item, std::ostream> {
private:
    std::ostream *_result;
protected:
    void init() override final { }
    void body(const Item& e) override final {
        if(cond(e)) *_result << func(e);
    }
    virtual std::string func(const Item& e) const = 0;
    virtual bool cond(const Item& e) const { return true; }
public:
    Summation(std::ostream *o) : _result(o) {}
};
```

5. ábra. Összegzés programozási tétel speciális esetét leíró új osztály-sablon

3.7. Egyéb új nyelvi elemek bevezetése

3.4.1. nullptr

C++11 előtt nem volt nyelvi kulcsszó a semmire nem mutató pointer kifejezésére, helyette a 0 konstanst vagy a NULL makró definíciót használhattuk. A könyvtár korszerűsítésével elkezdtük használni a *nullptr* kulcsszót, mely nem igényel „includot” és a fordító nem keveri össze semmilyen esetben sem a 0 konstans számértékkel. [7]

3.4.2. #pragrma once direktíva

Egy ugyan egy nem szabványos, de széles körben támogatott processzor direktíva. Az „include” őrfeltétel kiváltására szolgál.

4. Összefoglalás

Az osztály-sablon könyvtár előző fejezetben bemutatott kiegészítései és módosításai megnyugtató választ adnak a második fejezetben felvetett 4. és 5. kritikai észrevételekre.

2018-tól új tanterv szerint folyik az ELTE IK-n a programtervező informatikus képzés, és ennek keretében ugyanazon tantárgy tanítja a felsorolókra épített programozási tételek és az azokra visszavezethető (gyűjtemények feldolgozását végző) feladatok megoldását, valamint az objektum elvű programozás. Ezen témák természetes összefonódásaként találta meg a helyét a képzésünkben a felsorolót használó programozási tételek osztály-sablon könyvtára, és annak objektum-orientált felhasználása. Ez talán tompítja az első és harmadik kritikai észrevétel élét is.

Az első három kritikai észrevétellel kapcsolatban az alábbiakat mondhatjuk el.

Az első egy szubjektív álláspont, de nem akarunk vele vitatkozni. Ugyanakkor le kell szögezni, hogy az oktatásnak nem célja, hogy a hallgatókat ne mozdítsa ki a komfortzónájukból. Egy diplomás szakembertől általában is elvárható, hogy megváltozott körülményekre is képes legyen adaptálni a tudását, és ez különösen érvényes a nagyon gyorsan változó informatika területén.

A második kritikának érthető az oka, de egy ipari alkalmazásokhoz használt könyvtár oktatása egyrészt sokkal több időt venne igénybe, másrészt a tartalmi bőség miatt kevésbé lehetne fókuszálni arra a képzési célra, mely szerint az objektum-orientált technikákat (származtatás, függőség befecskendezés, példányosítás, futási idejű polimorfizmus kihasználása stb.) kellene a hallgatóknak megismerni és elsajátítani. Az általunk adott könyvtár a tantárgy anyagára épül, így azt nem kell külön megismertetni a hallgatókkal szemben egy az ipari alkalmazásokhoz hasznos könyvtárral.

A harmadik kritikát is elfogadjuk, de úgy gondoljuk, hogy az összetett technológiákat éppen az egyszerű feladatok megoldásán keresztül kell bemutatni.

Köszönetnyilvánítás

A projekt az Európai Unió támogatásával, az Európiai Szociális Alap társfinanszírozásával valósult meg (EFOP-3.6.3-VEKOP-16-2017-00002).

Irodalom

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. 1st edition, ISBN 0201633612, Pearson Education Inc., Addison Wesley Professionals, 1995.

2. Gregorics, T.: Programming theorems on enumerator, Teaching Mathematics and Computer Science, Debrecen, 8/1 (2010), 89-108

3. Gregorics, T.: Analogous programming with a template class library, Teaching Mathematics and Computer Scinece, Debrecen, 10/1 (2012), 135-152.

4. Gregorics, T.: Programozás – Megvalósítás. ISBN 978-963-312-065-1, Elte Eötvös Kiadó, Budapest, 2013.

5. Gregorics, T.: Force of Summation, Teaching Mathematics and Computer Scinece, Debrecen, 12/2 (2014), 185-199.

6. Stroustrup, B.: A C++ programozási nyelv. ISBN 963-9301-18-3, Kiskapu Kft. 2001.

7. Deitel, P. J., Deitel, H. M.: C++11 for Programmers. 2nd Edition, ISBN-13: 978-0133439854, Pearson Education Inc., 2014.

8. Az override és final használata: https://arne-mertz.de/2015/12/modern-c-features-override-and-final/ (2018.10.31.)

9. A template specializáció: https://en.cppreference.com/w/cpp/language/template_specialization (2018.10.31.)

10. Az osztály-sablon könyvtár: https://people.inf.elte.hu/gt/oep/library.zip (2018.10.31.)