Programozási feladatok megoldása visszavezetéssel egy osztály-sablon könyvtárra támaszkodva[1]

Gregorics Tibor

gt@inf.elte.hu
ELTE IK

Absztrakt. Ebben a cikkben egy osztály-sablon könyvtárat és annak felhasználását ismerhetjük meg. A könyvtár osztályai a programozási tételekre történő visszavezetéssel tervezett programok C++ nyelvű megvalósítását támogatják, de használatuknak elsődleges célja sokkal inkább annak a programozói stílusnak a megtanítása, amely származtatással, a virtuális metódusok felüldefiniálásával, valamint osztály-sablonok példányosításával éri el egy már megírt kód újrahasznosítását.

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

1. Bevezetés

A programozás oktatásának egy tipikus tanmenete, hogy a hallgatók először egyszerű algoritmusokat tanulnak tervezni és kódolni, és ehhez többnyire algoritmus mintákat (úgynevezett programozási tételeket) használnak, majd ezt követően ismerkednek meg a korszerű típus fogalmával, amely átvezet az objektum orientált programozáshoz. Ez utóbbinál az egy-egy osztály segítségével megoldható feladatok általában nem okoznak különösebb nehézséget, de az ennél komplexebb, több osztályból álló, a származtatás és sablon-példányosítás eszközeire építő megoldások már sokak számára nehezen érthető. Ennek az egyik oka szerintem az, hogy alig néhány olyan feladatcsoportot ismerünk, ahol kevés számú osztályra támaszkodva az objektum orientált technikákat lehetne gyakoroltatni. Ráadásul ezek a feladatok teljesen mások, mint amelyeket az egyszerű programok készítésének gyakorlásánál oldatunk meg.

De miért ne lehetne ugyanazon feladatokkal, amelyeket a programozási tételekre történő visszavezetéssel [2][3] [5] oldhatunk meg, az objektum orientált programozást is illusztrálni? Ha elkészítjük a programozási tételek osztály-sablon könyvtárát (ez néhány osztályból áll csak, hiszen a programozási tételek száma nem túl nagy), akkor erre támaszkodva, ennek a kódját újrahasználva nagyon sok olyan feladat oldható meg, amelyeket már korábbi tanulmányaikból ismerhetnek a hallgatók, így most csak a megoldás előállításának technikája lesz nekik új.

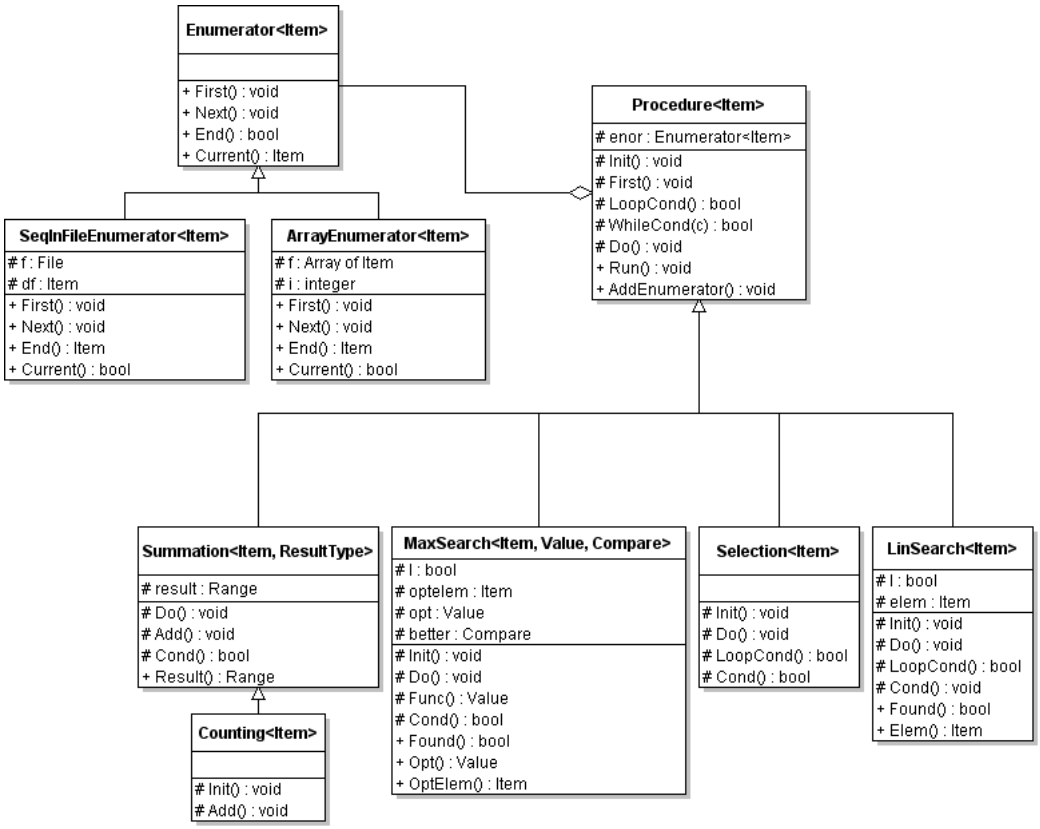
Ebben a technikában [6][7][8] a programozási tételekre visszavezetett programokat egy-egy tevékenység-objektum (illetve annak speciális metódusa) hajtja végre. Egy ilyen objektumnak az osztályát egy olyan osztály-sablonból származtatjuk, amelyik a megfelelő programozási tételt a legáltalánosabb formában, azaz felsorolókra megfogalmazott alakjában [4] tartalmazza. A származtatás során úgy hangoljuk rá az öröklött kódot a konkrét feladatra, hogy a programozási tétel

[1] A cikk a TÁMOP 4.2.1/B-09/1/KMR-2010-0003 pályázat támogatásával készült.

speciális tulajdonságait egyrészt a virtuális metódusok felüldefiniálásával adjuk meg (például egy számlálás feltételét vizsgáló logikai függvényt), másrészt a sablon-paramétereket kitöltjük (például a programozási tétellel bejárt elemeknek a típusát), harmadrészt futás közben további komponenseket kapcsolunk hozzá (a tevékenység-objektumhoz hozzákötjük azt a felsoroló-objektumot, amely a programozási tétel számára adagolja a feldolgozandó elemeket).

2. Az osztály-sablon könyvtár elemei

Az osztály-sablon könyvtár elemeit két nagy csoportba soroljuk. Az egyikben a programozási tételek osztályai, a másikban a nevezetes felsoroló objektumok osztályai lesznek. Mindkét csoport egy-egy ősosztályból származik.[1]



2.1. Általános felsoroló osztály

A felsorolók általános tulajdonságait [4] az *Enumerator* absztrakt osztály-sablon rögzíti. Minden olyan objektum, amelynek osztálya ebből származik, rendelkezni fog a bejárás négy alapműve-

letével: *First(), Next(), Current(), End()*. Ezek a műveletek ezen a szinten nincsenek definiálva (absztraktak), a bejárt elemek típusa sem ismert, azt az *Item* sablon-paraméter jelzi. Az osztály-sablon kódja C++ nyelven [8]:

```
template <typename Item>
class Enumerator {
public:
    virtual void First() = 0;
    virtual void Next()  = 0;
    virtual bool End()     const = 0;
    virtual Item Current() const = 0;

    virtual ~Enumerator(){}
};
```

Az ilyen absztrakt osztályt szokták interfésznek is hívni, amelyet a belőle származtatott osztályoknak implementálni kell. A továbbiakban minden felsoroló objektumnak az osztályát ebből az ősosztályból származtatjuk majd.

2.2. Nevezetes felsoroló osztályok

Leggyakrabban egy tömb vagy egy szekvenciális input file elemeit kell egy programozási tételben felsorolni. A szekvenciális input fájl sokszor egy szöveges állományra épül.

Egy egy-dimenziós tömb elemeinek felsorolóját egy olyan *Enumerator*-ból származtatott osztály-sablon definiálja, ahol a tömb elemeinek típusát még mindig sablon-paraméter jelzi, de az osztálynak implementálnia kell a felsorolás műveleteit. Ezért a tömbön kívül fontos adattagja lesz az osztálynak azon index, amelyikkel a tömb elemeit bejárjuk. Ezt az indexet állítja rá a tömb első elemére a *First()* művelet, ezt növeli meg eggyel a *Next()*, ezt vizsgálja meg az *End()*, hogy elérte-e már a tömb végét, és a tömbnek ennyiedik elemét adja vissza a *Current()*.

Egy szekvenciális inputfájl elemeinek felsorolóját is az *Enumerator* implementálásával készítjük el. Ennek fontos adattagja maga a fájl. A *First()* és *Next()* művelet az *f* fájl soron következő elemét olvassák be egy *df* adattagba, amelynek az értékét a *Current()* segítségével kérdezhetjük le, az *End()* pedig akkor ad igazat, ha a legutóbbi olvasás már sikertelen volt.

Nézzük meg ennek egy olyan C++ megvalósítását, ahol egy szöveges állomány elemeit kell szekvenciális inputfájlként kiolvasni (felsorolni). Ennek a konstruktora kapja meg a szöveges állomány nevét, és egy input adatfolyamot (*ifstream*) nyit az állományra. Nem létező állománynév esetén kivételt dob. A sablon-paraméter a beolvasandó értékek típusát adja meg. A szöveges állományban az értékeket szóközök, tabulátorjelek és sorvége-jelek választhatják el egymástól, kivéve, ha a szöveges állomány karaktereit akarjuk egyesével beolvasni. Ez utóbbi esetben a konstruktor kapcsolja ki azt az automatizmust, amely az olvasás során az elválasztó jeleket átugorja. A szöveges állományból való olvasás a >> operátorral történik, ezt használja a *First()* és a *Next()* művelet. Az osztálynak két adattagja van: a szöveges állományra nyitott *f* adatfolyam és a legutoljára beolvasott *df* érték. Ez utóbbi kérdezhető le a *Current()* metódussal, az *End()* metódus pedig az adatfolyam olvasási hibájakor (ezek között van a fájlvége bekövetkezése is) ad vissza igaz értéket.

```
template <typename Item>
class SeqInFileEnumerator : public Enumerator<Item>{
protected:
    std::ifstream f;
    Item          df;
public:
    enum Exceptions { OPEN_ERROR };
    SeqInFileEnumerator(const std::string& str){
        f.open(str.c_str());
        if(f.fail()) throw OPEN_ERROR;
        if(typeid(Item)==typeid(char))
            f.unsetf(std::ios::skipws);
    }
    void First()         { Next();}
    void Next()          { f >> df;}
    bool End()     const { return f.fail();}
    Item Current() const { return df; }
};
```

2.3. Általános programozási tétel osztálya

A *Procedure* osztály-sablon az összes programozási tétel őséül szolgál. Ha észrevesszük azt, hogy a nevezetes programozási tételeink valójában ugyanolyan ciklusból állnak: végig kell menniük egy felsoroló (legyen a neve mondjuk *enor*) által előállított elemeken és azokat kell feldolgozniuk, akkor ezt feldolgozást általánosan leírhatjuk, és ezt helyezzük el a *Procedure Run()* metódusában, amelyet a származtatás révén használhatnak majd az egyes tételek.

```
Init();
for (enor.First(); !enor.End(); enor.Next()){
    Do(enor.Current());
}
```

A *Run()* metódus végleges, nem-felülírandó, de két olyan metódust (*Init()* , *Do()*) hív meg, amelyeket ezen az absztrakciós szinten még nincs értelme megadni, csak majd származtatás során az egyes programozási tételeknek megfelelően kell felüldefiniálni.

A *Procedure* osztály-sablonnak biztosítania kell egy olyan metódust (*AddEnumerator()*) is, amellyel egy konkrét felsorolót lehet a feldolgozáshoz kapcsolni. Erre majd az *enor* adattag hivatkozik, amely a C++ megvalósításban egy pointer lesz. Annak érdekében, hogy a feldolgozás ne kezdődhessen el anélkül, hogy rendelkezünk felsorolóval, a *Run()* metódusban megvizsgáljuk a felsoroló állapotát: ha nem létezik (*enor==NULL*), dobunk egy kivételt. A *Procedure* osztálynak sablon-paramétere a felsorolt elemeknek ezen a szinten még ismeretlen típusa (*Item*).

```
template <typename Item>
class Procedure {
protected:
    Enumerator<Item> *enor;

    Procedure()enor(NULL){}
    virtual void Init()= 0;
    virtual void Do(const Item& current) = 0;
    virtual void First() {enor->First();}
    virtual bool WhileCond(const Item& current) const {return true;}
    virtual bool LoopCond() const
        { return !enor->End() && WhileCond(enor->Current()); }
public:
    enum Exceptions {ExpectedEnumerator};

    void Run();
    void AddEnumerator(Enumerator<Item>* en) { enor = en;}
    virtual ~Procedure(){}
};
```

Az általánosítás jegyében érdemes néhány módosításokat alkalmazni a *Run()* metóduson.

```
template <typename Item>
void Procedure<Item>::Run(){
    if(enum==NULL) throw ExpectedEnumerator;
    Init();
    for( First(); LoopCond(); enor->Next()){
        Do(enor->Current());
    }
}
```

Egyrészt az *enor->First()* metódust egy olyan *First()* metódussal váltjuk fel, amelynek alapértelmezett definíciója éppen az *enor->First()* lesz, de ez szükség esetén felülbírálható. Ez akkor hasznos, ha egy tevékenységet olyan felsorolóval kell végrehajtani, amelyet már korábban használtunk, de félbe hagytuk, és most folytatni akarjuk a felsorolást. Ilyenkor nincs szükség újra az *enor->First()* végrehajtására, ezért *First()* metódust az üres utasítással definiáljuk felül.

Másrészt az !*enor->End()* ciklusfeltételt kibővítjük (szigorítjuk) egy *WhileCond()* metódus hívásával, amely alapértelmezés szerint igaz értéket ad vissza, de ha kell, felüldefiniálható. Ezzel elérhetjük, hogy ha kell, a feldolgozást még az előtt leállítsuk, mielőtt a felsorolás véget érne. (Erre van szükség például akkor, amikor össze kell adni egy sorozat számait, de csak az első negatív szám előttieket.) A *WhileCond()* metódus értéke a felsorolás aktuális elemétől függ (ez a bemenő paramétere): amíg ez az elem kielégíti a megadott feltételt, addig folytatódhat a feldolgozás.

Az így kibővített ciklusfeltételt a *LoopCond()* metódusba zárjuk, hogy ez által lehetővé tegyük a ciklusfeltétel későbbi módosítását (mint ahogy ezt a lineáris keresés és a kiválasztás megfogalmazásánál meg is tesszük majd).

2.4. Nevezetes programozási tételek osztályai

Az ***összegzés*** tételével többféle feladat-típust is meg lehet oldani. A szigorúan vett összegzés mellett ilyen például az összeszorzás, összefűzés, össze-uniózás, feltételes összegzés, számlálás, másolás, kiválogatás, szétválogatás és összefuttatás. Ezt az általánosságot tükrözi az alábbi *Summation* osztály-sablon.

Az osztály-sablon *Item* paramétere a feldolgozandó elemek típusára, a *ResultType* paraméter az összegzés eredményének típusára utal. A *ResultType* típusú értékre hivatkozó *result* adattag az eredmény elérésére szolgál, amelyet majd a *Result()* metódussal kérdezhetünk le. A megvalósításban ez egy pointer, amely arra a memóriaterületre mutat, amelyet vagy a konstruktorban foglalunk le, vagy a konstruktor kapja meg annak címét

```
template < typename Item, typename ResultType = Item >
class Summation : public Procedure<Item>
{
protected:
    ResultType *result;
    bool inref;

    Summation(){ inref = true; result = new ResultType; }
    Summation(ResultType *r){ inref = false; result = r; }
    void Do(const Item& e){ if(Cond(e)) Add(e);}
    virtual void Add(const Item& e) = 0;
    virtual bool Cond(const Item& e) const { return true;}
public:
    ResultType Result() { return *result; }
    ~Summation(){ if(inref) delete result;}
};
```

Az osztály az *if (Cond(e)) Add(e)* feltételes tevékenységgel implementálja a *Do()* metódust. Ebben az *e* a *Do()* metódusnak paramétereként átadott, éppen felsorolt elem. A *Do()* metódust a *Summation*-ból származtatott osztályokban már nem változtatjuk meg. A *Procedure* osztály *Init()* (ez inizializálja a *result* adattagot) és *Add()*(ez módosítja a *result* adattagot) metódusát a *Summation* még nem definiálhatja, ezeket csak a leszármazott osztályokban tudjuk felüldefiniálni. Ugyanitt lehet majd – ha kell – felüldefiniálni a *Cond()*, a *WhileCond()* és a *First()* metódusokat is.

Mivel a ***számlálás*** programozási tételét, amely egy speciális összegzés, igen gyakran kell használni, ezért érdemes külön elkészíteni az általános osztály-sablonját úgy, mint az összegzés leszármazottja. A származtatásnál a *ResultType* típus az egész számok típusa lesz, ennek következtében a **result* egy egész szám, amelyet az őskonstruktor foglal le, az *Init()* metódusban kell nullára beállítani, és az *Add()* metódusban növelni eggyel az értékét. Az osztály felhasználásakor majd *Cond()* metódust kell felüldefiniálni: ebben kell adni a konkrét számlálás feltételét

```
template < typename Item >
class Counting : public Summation<Item, int>{
public:
    Counting():Summation<Item,int>(){}
protected:
    void Init()             { *Summation<Item,int>::result = 0;}
    void Add(const Item& e) { ++*Summation<Item,int>::result;}
};
```

A *MaxSearch* osztály-sablon egy ***általános maximumkeresést*** definiál, amely mind a közönséges maximum kiválasztást, mind a feltételes maximumkeresést leírja. A *MaxSearch* osztálynak három adattagja van. A keresés sikerét az *l* logikai változó jelzi, az eddig talált legjobb elemet az *optelem* és ennek értékét az *opt* változók őrzik. E három védett adattag értékének lekérdezését az osztály-sablon *Found()*, *Opt()* és *OptItem()* publikus metódusai biztosítják.

A *Func()* absztrakt virtuális metódus a felsorolt elemekhez rendeli azt az értéket, amely alapján összehasonlítjuk az elemeket. Ezen a szinten ezek a típusok még nem ismertek, rájuk sablonparaméterekkel hivatkozunk: *Item* a feldolgozandó elemek típusa, *Value* összehasonlítandó értéknek a típusa. A *Cond()* virtuális metódus pedig a feltételes maximumkeresés feltételét tartalmazza, amely alapértelmezés szerint igazat ad vissza (ettől válik az általános maximumkeresés közönséges maximum kiválasztássá).

Az osztálynak a már említett két sablon-paraméteren kívül van egy harmadik paramétere is: az összehasonlítás típusa (*Compare*). Ennek segítségével egy olyan összehasonlító objektumot (*better* adattag) definiálunk, amelyik két elem közül megmondja, hogy az első jobb-e, mint a második. Ezt úgy teszi, hogy értelmezve van az *operator(,)*, és ennél fogva a *better(left,right)* kifejezés a paraméterként megadott két értéket képes összehasonlítani. Ha a *Compare* helyébe az alábbi *Greater* osztály-sablont helyettesítjük, amelyik „>” reláció alapján értelmezi az *operator(,)*-t, akkor a *better(left,right)* kifejezés valójában a *left>right* kifejezést fogja helyettesíteni. A legnagyobb értékű elem (a maximum) kereséséhez pedig éppen egy ilyen kifejezésre lesz szükségünk.

```
template <typename Value>
class Greater{
public:
    bool operator()(const Value& left, const Value& right){
        return left > right;
    }
};
```

Ehhez hasonlóan készíthető el a „<” relációt magába foglaló *Less* osztály-sablon is, amely a legkisebb értékű elem (a minimum) kereséséhez kell, de ezen kívül bármilyen más „jobbsági” kritériumot is definiálhatunk.

Tekintsük meg a *MaxSerach* osztály-sablont.

```
template < typename Item, typename Value = Item,
      typename Compare = Greater<Value> >
class MaxSearch : public Procedure<Item>{
protected:
    bool    l;
    Item    optelem;
    Value   opt;
    Compare better;

    void Init(){ l = false;}
    void Do(const Item& current);
    virtual Value Func(const Item& e) const = 0;
    virtual bool  Cond(const Item& e) const { return true;}
public:
    bool  Found()   const { return l;}
    Value Opt()        const { return opt;}
    Item  OptElem() const { return optelem;}
};
```

Mivel a *MaxSearch* osztály-sablont a *Procedure* osztály-sablonból származtatjuk, ezért elsődleges feladata, hogy implementálja annak absztrakt metódusait. Az *Init()* metódus az *l* logikai változót állítja be hamisra. A *Do()* metódus a feltételes maximumkeresés programozási tételéből ismert hármas elágazásból áll. Egy konkrét maximum keresést ebből a MaxSerach-ből kell származtatnunk, de az *Init()* és a *Do()* metódusokat nem szabad felüldefiniálni, ellenben a *Func()* metódust mindenképpen meg kell adni, a *Cond()*, a *WhileCond()* és a *First()* metódusokat pedig felül lehet definiálni.

```
template < typename Item, typename Value, typename Compare >
void MaxSearch<Item,Value,Compare>::Do(const Item& current)
{
    Value val = Func(current);
    if ( !Cond(current) ) return;
    if (l){
        if (better(val,opt)){
            opt = val;
            optelem = current;
        }
    }else {
        l = true;
        opt = val;
        optelem = current;
    }
}
```

A kiválasztás osztály-sablonja (Selection) úgy implementálja az Init(), a LoopCond() és a Do() metódusokat, hogy a Run() metódus egy felsorolásnak az első adott tulajdonságú, de bizto-

san bekövetkező elemét keresse meg. A konkrét tulajdonságot Selection-ből származtatott osztályban a Cond() absztrakt metódus felüldefiniálásával lehet majd megadni, és ezért a LoopCond() metódust úgy kellett felüldefiniálni, hogy éppen a Cond() tagadottja legyen. Az Init() és a Do() metódus az üres utasítás lesz.

```
template < typename Item >
class Selection : public Procedure<Item> {
protected:
    void Init(){}
    void Do(const Item& e) {}
    bool LoopCond() const {
        return !Cond(Procedure<Item>::enor->Current());
    }
    virtual bool Cond(const Item& e) const = 0;
};
```

A ***lineáris keresés*** osztály-sablonja biztosítja, hogy akár egy pesszimista (normális), akár egy optimista (tagadott) lineáris keresést elő lehessen belőle állítani. A pesszimista lineáris keresés egy felsorolásnak az első adott tulajdonságú elemét keresi meg, és azt is megmondja, hogy egyáltalán talált-e ilyet. Az optimista lineáris keresés azt dönti el, hogy vajon a felsorolás minden eleme rendelkezik-e a megadott tulajdonsággal, és ha nem, akkor az első nem ilyen tulajdonságút adja meg. Az elnevezés onnan ered, hogy a normális keresésben a keresés sikerét jelző logikai változót kezdetben *hamis*-ra állítjuk, mintegy azt sugallva, hogy „úgysem fogunk találni megfelelő elemet”. Ezzel ellentétben az optimista keresésnél a logikai változó kezdőértéke az *igaz* lesz. Azt, hogy a keresés pesszimista vagy optimista legyen egy külön sablon-paraméterrel (*optimist*) állíthatjuk be. Ez alapértelmezés szerint *hamis*, amely a pesszimista lineáris keresést definiálja.

```
template < typename Item, bool optimist = false >
class LinSearch : public Procedure<Item> {
protected:
    bool l;
    Item elem;

    void Init() {l = optimist; }
    void Do(const Item& e) {l = Cond(elem = e);}
    bool LoopCond() const {
        return (optimist?l:!l) && Procedure<Item>::LoopCond(); }
    virtual bool Cond(const Item& e) const = 0;
public:
    bool Found()  const { return l;}
    Item Elem()   const { return elem;}
};
```

A keresés a sablon-paraméter értékétől függően inicializálja logikai adattagot (*Init()*) és a ciklusfeltétele (*LoopCond()*) is ennek megfelelően változik. A keresett tulajdonságot a *Cond()* absztrakt metódus felüldefiniálásával lehet majd megadni a *LinSearch*-ből származtatott konkrét osztályokban. A *Do()* metódus a keresés *l* logika változóját akkor állítja *igaz*-ra, ha az aktuális

elem kielégíti a keresett tulajdonságot. Az *elem* adattag a legutoljára megvizsgált elemet tartalmazza. A keresés eredményeit a *Found()* és *Elem()* metódusokkal kérdezhetjük le.

3. Feladat megoldások

Nézzünk meg most két olyan feladatot, amelynek a megoldását a fenti osztály-sablon könyvtárra támaszkodva állítjuk elő. Ehhez mindkét feladatnak az absztrakt megoldását egy-egy programozási tételre visszavezetve kell megtervezni. Látni fogjuk, hogy mindkét feladatnál egy egyedi felsoroló objektumra lesz szükség, amelynek osztályát az *Enumerator* osztályból származtatva kell elkészítenünk. Ugyanakkor mindkét egyedi felsoroló egy szöveges állomány adatait járja be, ezért az egyedi felsorolók definiálásához fel kell használnunk a szöveges állomány adatait kiolvasó szekvenciális inputfájl-felsorolót. A második feladatnál az egyedi felsoroló definiálásához még egy külön programozási tételre is szükség lesz.

4.1. Csupa közös elem

Példa: *Két szöveges állomány egész számokat tartalmaz. A számok mindkét állományban szigorúan növekvően rendezve, elválasztó jelekkel határolva helyezkednek el. Igaz-e, hogy nincs olyan szám bennük, amely ne szerepelne mindkettőben.*

A megoldáshoz a szöveges állományokban található számokat kell felsorolni úgy, hogy minden számról megmondjuk, hogy csak egy vagy mindkettő állományban szerepelt-e. Ha minden számra igaz, hogy mindkettő állományban szerepelt, akkor a feladatra adott válasz igaz lesz, egyébként hamis. Ezt egy optimista lineáris kereséssel dönthetjük el.

A feladatra szabott optimista lineáris keresés osztályában (*MyLinSearch*) a keresési feltételt kell alkalmas módon felüldefiniálni. A kereséshez felsorolt elemek egész szám-logikai érték párok (*Number*), ahol a logikai érték azt mutatja meg, hogy az adott szám közös eleme-e a két szöveges állománynak. A lineáris keresés feltételének (*Cond()*) éppen erre a logikai értékre van szüksége.

```
struct Number{
    int n;
    bool c;
};

class MyLinSearch:public LinSearch<Number,true>{
protected:
    bool Cond(const Number &e) const { return e.c;}
};
```

Szükség van egy speciális felsorolóra (*NumbersEnumerator*), amelyik a két állomány számait sorolja fel úgy, hogy minden számot csak egyszer érint, és megjelöli azokat, amelyek mindkét állományban szerepeltek.

```
class NumbersEnumerator : public Enumerator<Number>{
protected:
     SeqInFileEnumerator<int> *x, *y;
     Number number;
     bool end;
public:
     NumbersEnumerator(const string &str1, const string &str2);
     ~NumbersEnumerator(){ delete x; delete y; }
     void First(){x->First(); y->First(); Next();}
     void Next();
     bool End() const { return end;}
     Number Current() const { return number;}
};
```

A felsoroló megvalósításának ötletét két rendezett sorozat összefuttatásának algoritmusa adja. Ennek nyomát őrzi a felsoroló *Next()* műveletében található több ágú elágazás.

```
void NumbersEnumerator::Next() {
    if(end = x->End() && y->End()) return;
    if(y->End() || (!x->End() && x->Current()<y->Current())){
        number.n = x->Current(); number.c = false;
        x->Next();
    }else if(x->End() || (!y->End() && x->Current()>y->Current())){
         number.n = y->Current(); number.c = false;
         y->Next();
    }else if(!x->End() && !y->End() && x->Current()==y->Current()){
        number.n = x->Current(); number.c = true;
        x->Next(); y->Next();
    }
}
```

Az osztály konstruktora közvetve megnyitja olvasásra a szöveges állományokat és létrehozza azok adatait felsoroló objektumokat.

```
NumbersEnumerator::NumbersEnumerator(const string &str1,
                                     const string &str2){
    try{
        x = new SeqInFileEnumerator<int>(str1);
        y = new SeqInFileEnumerator<int>(str2);
    }catch(SeqInFileEnumerator<int>::Exceptions ex){
        if(ex==SeqInFileEnumerator<int>::OPEN_ERROR){
            cout << "Non-exisiting file" << endl; exit(1);
        }
    }
}
```

A megoldás két kulcsát adó osztály elkészítése után az alkalmazás főprogramja már igen egyszerű. Elég példányosítani a tevékenység objektumot és a felsoroló objektumot, hozzákapcsolni a felsorolót az tevékenységhez, majd végrehajtani a tevékenységet és kiírni az eredményt.

```
   MyLinSearch lin;
   NumbersEnumerator it("input1.txt", "input2.txt");
   lin.AddEnumerator(&it);
   lin.Run();
   if(lin.Found())cout << "All numbers are common" << endl;
   else cout << "There is non-common number" << endl;
```

4.2. Legjobb diák

Példa: *Egy szöveges állományban diákok osztályzatait soroltuk fel. Minden sorban egy diák azonosítóját (6 karakter) és egy osztályzási jegyet találunk szóközzel elválasztva. Az állomány diák-azonosító szerint rendezett. Adjuk meg az egyik legjobb (jegy-átlagú) diáknak a kódját.*

A feladat maximum kiválasztással oldható meg, ahol a diákok átlagát kell összehasonlítani. Ezért a maximum kiválasztás számára fel kell sorolni a diákokat az azonosítójukkal és jegyeiknek az átlagával. Egy ilyen azonosító-átlag pár előállításához egy olyan összegzésre lesz szükségünk, amely az egyazon azonosítóhoz tartozó osztályzatok számát és összegét is megadja.

A diák típusát (*Student*) úgy adjuk meg, hogy az alkalmas legyen egyetlen osztályzat leírására is, de egy diák átlagának leírására is. Annak érdekében, hogy a szöveges állományból egyetlen olvasással hozzájussunk a soron következő osztályzathoz, értelmezzük a *Student* típusra a beolvasás operátort.

```
struct Student{
   string id;
   double result;
   friend ifstream& operator>>(ifstream &in, Student &e)
};

ifstream& operator>>(ifstream &in, Student &e)
{
   in >> e.id >> e.result;
   return in;
}
```

A maximum kiválasztáshoz (*MyMaxSearch*) mindössze a *Func()* függvényt kell definiálni, amely egy azonosító-átlag alakban megadott diákhoz az átlagot rendeli.

```
class MyMaxSearch : public MaxSearch<Student, int>{
protected:
   int Func(const Student &e) const { return e.result;}
};
```

A szöveges állomány osztályzatait egy szekvenciális inputfájl elemeinek tekinthetjük, amelyet fel tud sorolni egy *SeqInFileEnumerator<Student>* típusú felsoroló. (Ehhez kellett a beolvasás operátor definíciója.) A maximum kiválasztáshoz azonban nem az osztályzatokat, hanem az egyes diákok átlagait (azonosító, átlag) kell felsorolni. Ezért egy olyan egyedi felsorolót (*StudentEnumerator*) kell készítenünk, amelyik minden lépésben (*First()* és *Next()*) megszámolja és összegzi az éppen vizsgált azonosítójú diák jegyeit, majd ez alapján kiszámolja az átlagot. Ez a felsoroló az *Enumerator* osztály-sablonból származik, a szöveges állomány osztályzatait bejáró szekvenciális inputfájl felsorolóra támaszkodik. A *student* az aktuális diák adatait tartalmazza, az *end* pedig azt jelzi, hogy befejeződött-e már a felsorolás.

```
class StudentEnumerator : public Enumerator<Student>{
protected:
   SeqInFileEnumerator<Student>* f;
   Student student;
   bool end;
public:
   StudentEnumerator(const string &str);
   ~StudentEnumerator(){ delete f;}
   void First(){f->First(); Next();}
   void Next() ;
   bool End() const { return end;}
   Student Current() const { return student;}
};
```

A konstruktor közvetve megnyitja azt szöveges állományt amelynek a nevét paraméterként megkapta, kivételt dob, ha ez nem sikerül, különben létrehozza az osztályzatok felsorolóját (*f*).

```
StudentEnumerator::StudentEnumerator(const string &str){
    try{ f = new SeqInFileEnumerator<Student>(str); }
    catch(SeqInFileEnumerator<Student>::Exceptions ex){
        if(ex==SeqInFileEnumerator<Student>::OPEN_ERROR){
            cout << "Non-exisiting file" << endl; exit(1);
        }
    }
}
```

A *Next()* művelet ellenőrzi, hogy van-e még feldolgozatlan diák és ha igen, kiszámoljy a következőnek az átlagát.

```
void StudentEnumerator::Next() {
    if(end = f->End()) return;
    student.id = f->Current().id;
    MySummation sum(student.id);
    sum.AddEnumerator(f);
    sum.Run();
    student.result = sum.Result().sum/sum.Result().count;
}
```

A *Next()* művelet definiálásához egy külön programozási tételre van szükség. Sőt tulajdonképpen kettőre: az egyik megszámolja az adott azonosítójú diák jegyeit, a másik összeadja azokat. Ez két azonos alakú összegzés, ezért össze lehet őket vonni (*MySummation*). Ennek a kettős tevékenységnek szüksége van kezdetben az aktuális azonosítóra, az eredménye pedig két szám lesz, amelyet majd a *Pair* típusú struktúrában tárolunk. Az eredmény számait az *Init()*-ben kell inicializálni és az *Add()*-ban módosítani. Lekérdezni majd a *Result()*-tel lehet.

```
struct Pair{
   double sum;
   int count;
};
```

A kettős összegzés felsorolója az osztályzatok *f* felsorolója lesz, amelyet nemcsak egy adott összegzés során az éppen aktuális azonosítójú adatok olvasásához használunk, hanem a teljes feldolgozás során minden azonosítóra. Ez tehát egy már korábban megkezdett felsoroló, amelyet ezért nem szabad újraindítani, ennek megfelelően az üres utasítással definiáljuk felül a *First()* metódust. A kettős összegzés nem az *f* felsorolásának végéig tart, hanem csak addig, amíg a soron következő adat azonosítója megegyezik az aktuálissal. Ezt fogalmazzuk bele a *WhileCond()* metódusba. Az aktuális azonosítót a konstruktor tárolja el az *id* adattagban.

```
class MySummation : public Summation<Student,Pair>{
protected:
    string id;
    void First(){}
    void Init(){result->sum = 0; result->count = 0;}
    void Add(const Student &e)
        {result->sum+=enor->Current().result; ++result->count;}
    bool WhileCond(const Student &e) const { return e.id == id;}
public:
    MySummation(const string &str):Summation<Student,Pair>(),id(str){}
};
```

Végül vizsgáljuk meg a főprogramot. Ez példányosítja a maximum kiválasztás tevékenységének és az egyedi felsorolásnak az objektumait, összekapcsolja őket, és végrehajtja a maximumkeresést és kiírja az eredményt.

```
    MyMaxSearch max;
    StudentEnumerator it("input.txt");
    max.AddEnumerator(&it);
    max.Run();
    if(max.Found())
        cout << "The best student: " << max.OptElem().id << endl;
    else
       cout << "There are no students!" << endl;
```

4. Összefoglalás

Talán a bemutatott két példa is elég annak érzékeltetésére, hogy mennyire sokféle feladat oldható meg az osztály-sablon könyvtár segítségével. Különösebb erőfeszítés nélkül tudunk ehhez hasonló, akár egyszerűbb, akár összetettebb feladatokat kitalálni.

Az osztály-könyvtár alkalmazása még tudatosabbá teszi a visszavezetés technikájának használatát. A megoldások előállításához itt nem használhatjuk az algoritmikus gondolkodást, helyette a feladat elemzésére kell koncentrálni.

A megoldások megvalósítása példát mutat az objektum-orientált programozásra. Rávilágítanak a származtatásnak, a virtuális metódusoknak és a sablon-paramétereknek a szerepére, példát adnak arra, hogy miként lehet a kód-újrafelhasználás folyamatát a szerkesztési, a fordítási és futási időben szétosztani.

A könyvtár használatával igen szép megoldásokat tudunk előállítani. Programozói szemmel például nagyon érdekes, hogy az előállított megoldásokban mindössze egyetlen ciklust találunk, mégpedig a programozási tételek ősosztály-sablonjának *Run()* metódusában. Sem a könyvtár többi osztályában, sem a konkrét alkalmazás hozzáadott kódjában nem kell újabb ciklust írni.

A bemutatott osztály-sablon könyvtár nem az iparszerű alkalmazások számára készült. Kiválóan alkalmas a különféle objektum-orientált implementációs technikák megmutatására, de nem hiszem, hogy a gyakorlatban való alkalmazása egyszerű illetve célszerű lenne. Egy programozási tétel (lényegében egy ciklus) implementálása ugyanis önmagában nem túl nehéz feladat, ezért sokkal könnyebb közvetlenül kódolni, mint egy összetett osztály-sablon könyvtárból származtatni, hiszen ehhez a könyvtár elemeit kell pontosan megismerni és helyesen alkalmazni. Ennél fogva ez a cikk rávilágít arra a határra is, hogy mikor érdemes az objektum-orientált illetve generikus nyelvi eszközöket bevetni egy feladat csoport megoldásánál és ez mikor nem jelent már a gyakorlatban előnyt.

Irodalom

1. Booch, G., Rumbaugh, J., Jakobson, I.: The Unified Modeling Language User Guide, Addison-Wesley Longman Inc., 1999.
2. Fóthi, Á.: Bevezetés a programozáshoz. ELTE Eötvös Kiadó. 2005.
3. Csepregi, Sz., Dezső, A., Gregorics, T., Sike, S.: Automatic Implementation of Service Required by Components, PROVECS'2007 Workshop, ETH Technical Report 567. 2007.
4. Gregorics, T.: Programming theorems on enumerator. Teaching Mathematics and Computer Science, Debrecen, 8/1 (2010), 89-108
5. Gregorics, T., Sike, S.:Generic algorithm patterns, Proceedings of Formal Methods in Computer Science Education FORMED 2008, Satellite workshop of ETAPS 2008, Budapest March 29, 2008, p. 141-150.
6. Mayer, B.: Object Oriented Software Construction, Prentice Hall PTR, 1997.
7. Rumbaugh,J., et all: Object Oriented Modelling and Design, Prentice Hall International Inc., 1991.
8. Stroustrup, B.: A C++ programozási nyelv, Kiskapu Kft. 2001.