

Programozás alapjai II.

(12. ea) C++

OO tervezési megfontolások, újabb ter. minták

Goldschmidt Balázs

Szeberényi Imre

BME IIT

<balage@iit.bme.hu>



MŰEGYETEM 1782

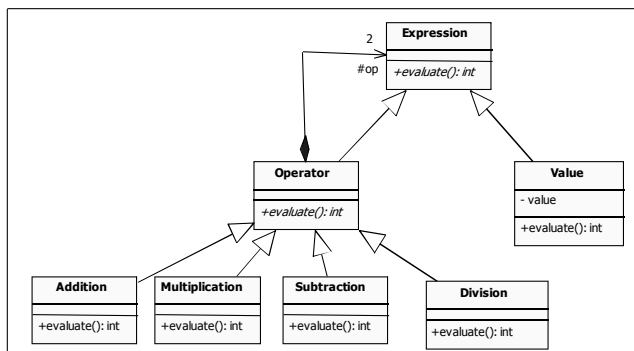
Tervezési feladat

- Készítsünk számológépet!
 - matematikai kifejezéseket kell modellezni
 - pl. $3+4*(5-2)$
 - alpműveletek
 - + - * /
 - egész számok
 - pl 1, 2, 3, 13
 - a beolvasás most mellékes

Kifejezések modellezése

- Kifejezés: számok és műveletek
- Számok felelőssége
 - megmondja, mennyit tárol
- Műveletek felelőssége
 - van operandusa (bal, jobb)
 - kiszámolja a művelet eredményét
 - mi lehet operandus?
 - művelet *vagy* szám
 - heterogén kollekció → öröklés
 - hogyan kapunk eredményt? Metódussal!

Kifejezés osztály és leszármazottai



C++ megvalósítás

- Expression

```
class Expression {
public:
    virtual int evaluate() const = 0; // absztrakt metódus
    virtual ~Expression() {}
};
```

C++ megvalósítás

- Value

```
class Value : public Expression {
    int value;
public:
    Value(int v = 0) : value(v) {}
    int evaluate() const;
};

int Value::evaluate() const { return value; }
```

C++ megvalósítás

- Operator

- hogyan hivatkozunk az operandusokra?
 - referencia vs pointer
- referencia
 - nem kell memóriakezeléssel foglalkozni
 - csak egyszer állítható (konstruktor)
- pointer
 - memóriakezelés kérdéses
 - másolás, destrúálás, stb...
 - konstruálás után is beállítható, módosítható

C++ megvalósítás

- Operator

- használjunk pointert
 - memóriakezelést később

```
class Operator : public Expression {
protected:
    Expression * op[2];
public:
    Operator(Expression * e1, Expression * e2) {
        op[0] = e1;
        op[1] = e2;
    }
    void setOperand(Expression * e, int n) { op[n] = e; }
};
```

C++ megvalósítás

- Addition

- *Operator* leszármazottja
 - operandusok, evaluate

```
class Addition : public Operator{
public:
    Addition(Expression * e1, Expression * e2) :
        Operator(e1,e2) {}
    int evaluate() const;
};

int Addition::evaluate() const {
    return op[0]->evaluate() + op[1]->evaluate();
}
```

C++ megvalósítás

- Multiplication, Subtraction, Division
– Mint *Addition*, csak *evaluate* más

```
....  
int Multiplication::evaluate() const {  
    return op[0]->evaluate() * op[1]->evaluate();  
}  
int Subtraction::evaluate() const {  
    return op[0]->evaluate() - op[1]->evaluate();  
}  
int Division::evaluate() const {  
    return op[0]->evaluate() / op[1]->evaluate();  
}
```

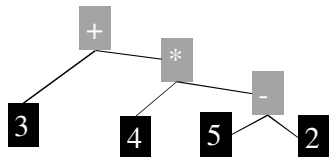
C++ megvalósítás

- Egyszerű példa a használatra
3 + 4

```
Value v3(3);  
Value v4(4);  
  
Addition a(&v3, &v4);  
  
cout << a.evaluate() << endl;
```

C++ megvalósítás

- Összetett példa a használatra
3+(4*(5-2))
3+(4*(5-2))



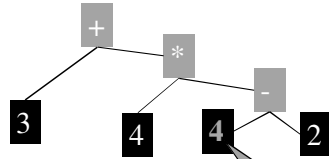
```
Value v2(2), v3(3), v4(4), v5(5);  
Subtraction s(&v5, &v2);  
Multiplication m(&v4, &s);  
Addition a(&v3, &m);  
  
cout << a.evaluate() << endl; // 15
```

C++ megvalósítás

- Összetett példa a használatra

$3+(4*(4-2))$

$3+(4*(4-2))$



```
Value v2(2), v3(3), v4(4), v5(4);  
Subtraction s(&v5, &v2);  
Multiplication m(&v4, &s);  
Addition a(&v3, &m);
```

```
cout << a.evaluate() << endl; // 11
```

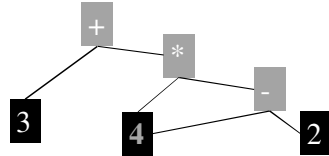
Ilyen már van!

C++ megvalósítás

- Összetett példa a használatra

$3+(4*(4-2))$

$3+(4*(4-2))$



```
Value v2(2), v3(3), v4(4), v5(4);  
Subtraction s(&v4, &v2);  
Multiplication m(&v4, &s);  
Addition a(&v3, &m);  
cout << a.evaluate() << endl; // 11
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_13 → számologep

Memóriakezelés

- Pointereket tárolunk
 - a példában lokális változók → automatikusan megszűnnek
 - mi van, ha dinamikus? Ki szabadítja fel?
- Felszabadítás beépítve (kompozíció)
 - mindig dinamikus a foglalás
 - egy példány csak egy operátornál
- Felszabadítás külön kezelve
 - ki csinálja?

Factory objektum

- Felelősség
 - új objektum létrehozása
 - munka végén rendet rak
- Létrehozza az objektumokat
 - *add*(Expression* e1 = null, Expression* e2 = null)
 - *div*(...), *mult*(...), *sub*(...)
 - *val*(int value)
 - ha korábban már létrehoztuk, jó a régi
 - a többszörözést is el tudjuk kerülni

Factory tervezési
minta

Factory használata

- Összetett példa a *factory* használatára

3+(4*(4-2))

```
{ Factory f;  
Subtraction *s = f.sub(f.val(4), f.val(2));  
Multiplication *m = f.mult(f.val(4), s);  
Addition *a = f.add(f.val(3), m);
```

```
cout << a->evaluate() << endl; // 11
```

```
}
```

Létrehozza a műveletet és
egy pointer-t ad vissza az
objektumra.

f destruktora felszabadít minden
létrehozott objektumot

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_13 → számológép

Factory megvalósítás

```
class Factory {  
    list<Expression*> created; // vector?  
    map<int, Value*> values;  
public:  
    Value* val(int i);  
    Addition* add(Expression * e1, Expression * e2);  
    ...  
    ~Factory() {  
        while (!created.empty()) {  
            delete created.front();  
            created.pop_front();  
        }  
    }  
};
```

felszabadít minden
létrehozott objektumot

Factory megvalósítás

```
Value* Factory::val(int i) {
    if (values[i] == NULL) {
        values[i] = new Value(i);
        created.push_back(values[i]);
    }
    return values[i];
}
Addition* Factory::add(Expression * e1, Expression * e2) {
    Addition* a = new Addition(e1, e2);
    created.push_back(a);
    return a;
}
```

Kihasználjuk hogy a <value> def. konstruktra fut le, ha nem volt megfelelő <key> a map-ben.

Változók

- Legyenek változóink!

$x^3 + z \cdot y$

- az értékeket lehessen központilag állítani
- minden kiértékelésnél az aktuális értékkel számoljanak

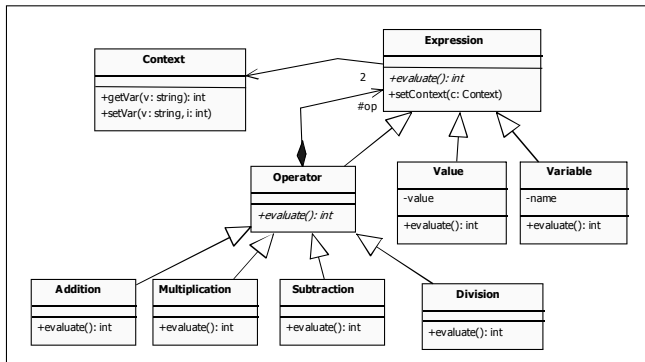
```
Variable vx("x"), vy("y"), vz("z");
Value v(3);
```

```
Expression *e = f.add(f.mult(&vx, &v), f.mult(&vy, &vz));
e->evaluate(); // Hogy kapnak értéket a változók?
```

Változók értéke hol?

- Be van drótozva *Variable*-be
 - akkor miért változó?
- Globális tárban (kontextus) tároljuk
 - hogyan tudjuk ugyanazt a kifejezést kiértékelni különböző számokra?
 - pl. $v = s/t \rightarrow \text{div}("s", "t");$
- Kiértékelés előtt állítjuk be (1. változat)
 - `setContext(Context* c)`
 - nem kiértékelés, hanem objektum-specifikus ☹
- A kifejezésnek kiértékeléskor adjuk át (2. változat)
 - `int evaluate(Context* ctx)`
 - módosítani kell a kiértékelő függvényt ☹
 - ez a korrekt megoldás (lokális tudás)

1. változat: Kontextus objektum szinten



Variable megvalósítása (1. változat)

```
class Context {
    map<string, int> values;
public:
    void setVar(string s, int i) { values[s] = i; }
    int getVar(string s) { return values[s]; }
};
class Variable : public Expression {
    string name;
public:
    Variable(string n = "x") : name(n) {}
    int evaluate() const {
        return ctx->getVar(name);
    }
};
```

Válasszon ismerni kell.
Örököljük az expression-ből!

Meglevő osztályok bővítése (1. változat)

```
class Expression {
protected: Context* ctx;
public: virtual void setContext(Context* c) { ctx = c; }
    ... // többi marad
};
class Operator {
public:
    void setContext(Context* c) { Expression::setContext(c);
        op[0]->setContext(c); op[1]->setContext(c);
    }
    ... // többi marad
};
```


Változók használata (1. változat)

- Értékeljük ki: $x*3+z*y$
 $x = 5, y = 6, z = 7$

```
Variable vx("x"), vy("y"), vz("z");  
Value v(3);
```

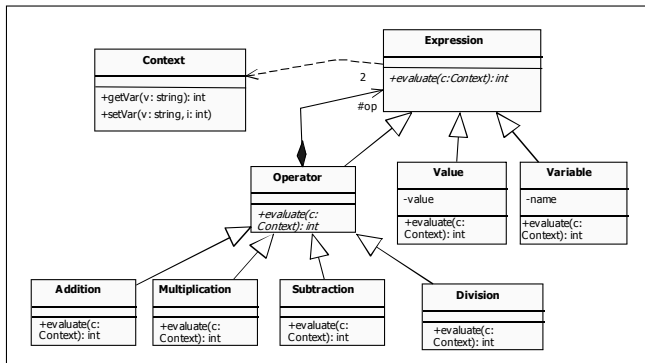
```
Expression *e = f.add(f.mult(&vx, &v), f.mult(&vz, &vy));
```

```
Context ctx;  
ctx.setVar("x", 5); ctx.setVar("y", 6); ctx.setVar("z", 7);  
e->setContext(&ctx);
```

```
cout << e->evaluate() << endl;
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_13 → szamologep2

2. változat: Kontextus metódus szinten



Változók megvalósítása (2. változat)

```
class Context {  
    map<string, int> values;  
public:  
    void setVar(string s, int i) { values[s] = i; }  
    int getVar(string s) { return values[s]; }  
};  
class Variable : public Expression {  
    string name;  
public:  
    Variable(string n = "x") : name(n) {}  
    int evaluate(Context& ctx) const {  
        return ctx.getVar(name);  
    }  
};
```

Metódus paramétereként kapjuk

Meglevő osztályok bővítése (2. változat)

```
class Expression {
public: virtual int evaluate(Context& c) const = 0;
        ... // többi marad
};
```

A többi osztályban is módosul a metódus fejléce. Pl:

```
class Addition : public Operator{
public:
    Addition(Expression * e1, Expression * e2) : Operator(e1,e2) {}
    int evaluate(Context& c) const;
};
```

Változók használata (2. változat)

- Értékeljük ki: $x*3+z*y$
 $x = 5, y = 6, z = 7$

```
Variable vx("x"), vy("y"), vz("z");
Value v(3);

Expression *e = f.add(f.mult(&vx, &v), f.mult(&vz, &vy));

Context ctx;
ctx.setVar("x", 5); ctx.setVar("y", 6); ctx.setVar("z", 7);

cout << e->evaluate(ctx) << endl;
```

Változók Factoryban

- Factory gyártsa a változókat is
 - most mindegyik új, memóriakezelés nincs megoldva
 - lehetne itt is csak újat létrehozni, mint *Value*-nál.

```
// Factory osztályba új metódus deklarációja kerül
// alább pedig a definíció
Variable* Factory::var(string s) {
    Variable* v = new Variable(s);
    created.push_back(v);
    return v;
}
```

Változók használata Factory-val

- Értékeljük ki: $x*3+z*y$

$$x = 5, y = 6, z = 7$$

```
Factory f;
Variable *vx = f.var("x"), *vy = f.var("y"), *vz = f.var("z");
Value* v = f.val(3);

Expression* e = f.add(f.mul(vx, v), f.mul(vz, vy));

Context ctx;
ctx.setVar("x", 5); ctx.setVar("y", 6); ctx.setVar("z", 7);

cout << e->evaluate(ctx) << endl;
```

Mi a kifejezések típusa?

- Most a megoldás *int*-tel dolgozik

- Lehesen bármi ☺

→ Sablon!

– ehhez mindent ki kell bővíteni

- `template <class T>, Expression<T>`, stb
- minden `.h + .cpp` → `.hpp`

– öröklésnél vigyázni

- *using* az örökölt tagváltozók elérése előtt
pl. *using* `Operator<T>::op`;

– innentől minden működik, ami kellhet

- *int*, *double*, *Complex*, stb

```
//pl:
template <typename T>
class Addition : public Operator<T>{
    using Operator<T>::op;
public:
    Addition(Expression<T> * e1, Expression<T> * e2)
        : Operator<T>(e1,e2) {}
    T evaluate(Context<T>& c) const;
};
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_13 → `szamologep3`

Merre tovább?

- Bővítések

– egyoperandusú műveletek (Unary)

- negálás, gyökvonás, reciprokképzés, ...

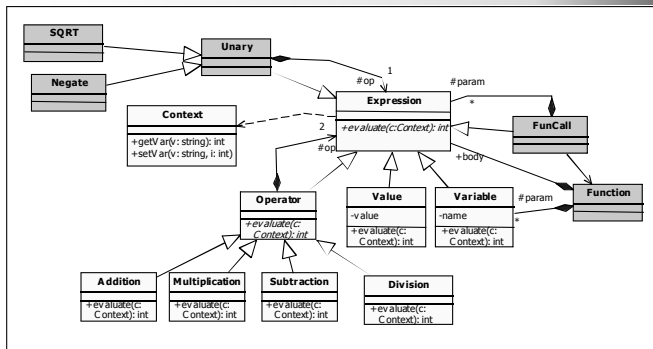
– függvények (Function)

- bemenő formális paraméterek: változók (Variables)
- törzs: kifejezés (Expression)
- kiértékelés:
 - aktuális paraméterek kiértékelése és átadása a formális paramétereknek
 - függvénytörzs lefuttatása

– függvény meghívása (FunCall)

- aktuális paramétert tartalmazó kifejezés és a függvény pointer tárolása
- kiértékelés

Bővített osztálydiagram



Egyváltozós függvény

- pl. $f(x) = x * x$

```

class Function {
    Expression* expr;
    Variable* param;
public:
    Function(Expression* e, Variable* p) : param(p), expr(e) {}
    int evaluate(Context& ctx, Expression* ap) {
        Context c2(ctx);
        c2.setVar(param->getName(), ap->evaluate(ctx));
        return expr->evaluate(c2);
    }
};
    
```

Egyváltozós függvény meghívása

- hivatkozás, pl. $4 + f(5 + 3) * 2 + f(2)$

```

class FunCall : public Expression {
    Function* f; // meghívandó függvény
    Expression* p; // függvény paramétere ez a kifejezés
public:
    FunCall(Function* f, Expression* e) : f(f), p(e) {}

    int evaluate(Context& ctx) const {
        return f->evaluate(ctx, p);
    }
};
    
```

Egyváltozós függvény használata

- használat, pl.: $f(t)=t*t$; $f(a)+f(b)$, ha $a=3$, $b=4$

```
Context ctx;
Variable t("t");
Multiplication m(&t, &t);
Function f(&m, &t);

Variable a("a"), b("b");
FunCall a2(&f, &a), b2(&f, &b);
Addition ad(&a2, &b2);

ctx.setVar("a", 3); ctx.setVar("b", 4);
cout << ad.evaluate(ctx) << endl;
```

Minden együtt

```
// Függvény Faktory-val: ft(t) = t * t, f(a) + f(b), ha a = 3, b = 4;
```

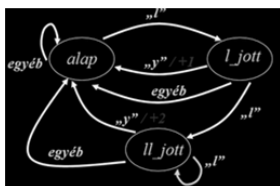
```
Factory<float> f;
Context<float> ctx;
Variable<float> *t = f.var("t");
Multiplication<float>* m = f.mul(t, t);
Function<float> ft(m, t);
Variable<float> *a = f.var("a"), *b = f.var("b");
FunCall<float> *a2 = f.call(&ft, a), *b2 = f.call(&ft, b);
Addition<float> *ad = f.add(a2, b2);

ctx.setVar("a", 3); ctx.setVar("b", 4);
cout << ad->evaluate(ctx) << endl;
```

https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_13 → szamologep3

Állapotgépek (FSM) általánosítása

- Állapotgép állapotai jól modellezhetők osztályokkal
- Állapothoz tartozó műveletek
 - belépés az adott állapotba (Enter)
 - adott állapotban input fogadása → új állapot (Input)
 - állapot elhagyása (Leave)
- Példa: ly-számláló



Állapotgépek általánosítása /2

- általános állapot (sablon az inputra)
 - absztrakt osztály (Enter, Input, Leave)

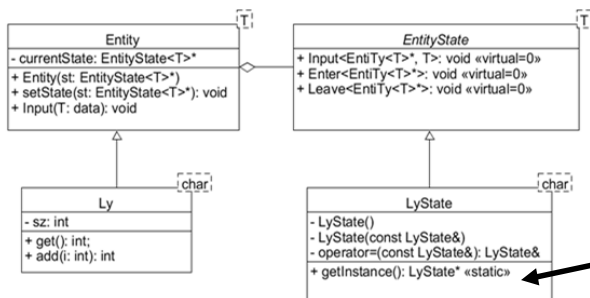
```
template <typename T>
class EntityState {
public:
    virtual void Input(Entity<T>* st) = 0;
    virtual void Enter(Entity<T>* st) = 0;
    virtual void Leave(Entity<T>* st) = 0;
    virtual ~EntityState() {}
};
```

```
template <typename T>
class Entity {
    EntityState<T>* currentState;
public:
    Entity(EntityState<T>* st) : currentState(st) {}
    void setState(EntityState<T>* st) {
        currentState->Leave(this);
        currentState = st;
        currentState->Enter(this);
    }
    void Input(T data) { currentState->Input(this, data); }
    virtual ~Entity() {}
};
```

- általános entitás (sablon az inputra)
 - állapot tárolás (currentState)
 - állapot beállítás (setState)
 - input fogadás és továbbítás az akt. állapotnak.

Ly-számláló FSM terv. mintával

FSM tervezési minta



Singleton tervezési minta

Minden állapotból csak egy példány kell/lehet.

Ly-számláló FSM terv. mintával

```
/// Ly számláló osztály
class Ly : public Entity<char>{
    int sz;
public:
    Ly() : Entity<char>(LyState<S, Alap>::getInstance(), sz(0) {}
    int get() const { return sz; }
    void add(int i) { sz += i; }
};
```

Ly-számláló FSM ter. mintával

```
template <typename T, T c>
class LyState : public EntityState<char> {
    LyState() {}
    LyState(const LyState&);
    LyState& operator=(const LyState&);
public:
    void Enter(Entity<char>*) {}
    void Leave(Entity<char>*) {}
    static LyState* getInstance() {
        static LyState singleton;
        return &singleton;
    }
    void Input(Entity<char>* ly, char ch);
};
```

```
/// Ly számláló állapotai
enum S { Alap, Ljott, LLjott};

/// Alap állapot inputja
template<>
void LyState<S, Alap>::Input(Entity<char>* ly, char ch) {
    if (ch == 'T' || ch == 'L') {
        ly->setState(LyState<S, Ljott>::getInstance());
    }
}
```

Ly-számláló FSM ter. mintával /2

```
/// Ljott állapot inputja
template<>
void LyState<S, Ljott>::Input(Entity<char>* ly, char ch) {
    if (ch == 'T' || ch == 'L') {
        ly->setState(LyState<S, LLjott>::getInstance());
    } else {
        if (ch == 'y' || ch == 'Y') dynamic_cast<Ly*>(ly)->add(1);
        ly->setState(LyState<S, Alap>::getInstance());
    }
}

/// LLjott állapot inputja
template<>
void LyState<S, LLjott>::Input(Entity<char>* ly, char ch) {
    if (ch == 'y' || ch == 'Y')
        dynamic_cast<Ly*>(ly)->add(2);
    if (ch != 'T' && ch != 'L')
        ly->setState(LyState<S, Alap>::getInstance());
}
```

Ly-számláló FSM ter. mintával /3

```
int main() {
    Ly ly;
    char ch;

    std::cout << "Johet a duma:\n";
    while (std::cin >> std::noskipws >> ch)
        ly.Input(ch);
    std::cout << "ly-ok szama:" << ly.get() << std::endl;
}
```

Köszönöm a figyelmet

