

Curs 3 – Clase și obiecte

- Alocare dinamica - TAD Lista
- Limbajul de programare C++
- Programare orientată obiect

Curs 2 – Programare modulară în C

- Funcții – Test driven development, Code Coverage
- Module – Programare modulară, TAD
- Gestirea memoriei în C/C++

Gestiunea memoriei in C / C++

Memoria ocupată de o aplicație în timpul rulării este împărțita pe segmente de memorie:

Segment	Ce conține	Gestiune in C/C++
Stack (call stack)	Variabile locale, transmitere parametrii, informații despre funcții care au fost apelate dar încă nu sau terminat.	Gestionat automat și eficient. Structura de stiva (LIFO). La fiecare apel de metoda se pune pe stivă un Stack Frame La terminarea metodei se elimină ultimul stack frame.
Heap (free segment, dynamic memory)	Variabile alocate dinamic	Gestionat manual de programator folosind: <code>malloc</code> , <code>calloc</code> , <code>realloc</code> , <code>free</code> .
Data segment (initialized data segment)	Variabile globale și statice inițializate din program	
Bss segment (uninitialized data segment)	Variabile globale și statice neinițializate (memorie inițializată cu 0)	
Code segment (text segment)	Instrucțiuni cod mașina (programul compilat)	Read only in general

Stack vs Heap

Stack avantaje:

- Gestionat automat (domeniu de vizibilitate/ciclu de viață pentru variabile)
- Structura de date tip stiva – LIFO – Last in first out
- Eficient
 - se gestionează doar un pointer la capul stivei (operații simple de aritmetică de pointeri)
 - Zona compactă de memorie, frecvent în memoria cache al procesorului (L1,L2,L3 cache)
 - Fiecare fizică de execuție are stack propriu (nu este nevoie de sincronizare)

Stack dezavantaje:

- Memoria pentru variabilele de pe stiva este eliberată la terminarea funcției (blocului {}) în care am declarat variabila
- Dimensiune limitată (default 1Mb pe windows)
- Memoria aferentă valorilor din stack în general trebuie specificat (cunoscut) la compilare

Heap avantaje:

- Permite alocarea memoriei în timpul rulării programului.
- Putem avea memorie alocată fără a fi dealocată automat la terminarea funcției (blocului {})
- Dimensiunea memoriei este specificată dinamic (valori cunoscute doar în timpul execuției)
- Permite definirea de structuri de date mai complicate (înlănțuire, arbore, graf)
- Dimensiune mult mai mare (limitată doar de sistemul de operare)

Heap dezavantaje:

- Gestiona (alocare/de-alocare) este făcut de programator. Se poate ușor ajunge la memory leak (memorie alocată dar niciodată eliberată).
- Ineficient (comparativ cu stack)
 - folosește o structură internă de gestiune mult mai complicată decât o stiva.
 - Este partajat între fizicele de execuție ale programului
 - Poate conduce la fragmentarea memoriei

Alocare dinamica

Folosind funcțiile **malloc(size)** și **free(pointer)** programatorul poate aloca/elibera memorie pe Heap – zonă de memorie gestionată de programator

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    //allocate memory on the heap for an int
    int *p = malloc(sizeof(int));

    *p = 7;
    printf("%d \n", *p);
    //Deallocate
    free(p);
    //allocate space for 10 ints (array)
    int *t = malloc(10 * sizeof(int));
    t[0] = 0;
    t[1] = 1;
    printf("%d \n", t[1]);
    //deallocate
    free(t);
    return 0;
}

/**
 * Make a copy of str
 * str - string to copy
 * return a new string
 */
char* stringCopy(char* str) {
    int len = strlen(str) + 1; // +1 for the '\0'
    char* newStr = malloc(sizeof(char) * len); // allocate memory
    strcpy(newStr, str); // copy string
    return newStr;
}
```

Programatorul este responsabil cu dealocarea memoriei

OBS: Pentru fiecare **malloc** trebuie să avem exact un **free**

Memory management

void* malloc(int num); - alocă **num** byte de memorie, memoria este neinitializată

void* calloc(int num, int size); - alocă **num*size** memorie, initializează cu 0

void* realloc(void* address, int newsize); - redimensionare memorie alocată

void free(void* address); - eliberează memoria (este disponibilă pentru următoarele alocări)

Memory leak

Programul aloca memorie dar nu dealoca niciodată, memorie irosită

```
int main() {
    int *p;
    int i;
    for (i = 0; i < 10; i++) {
        p = malloc(sizeof(int));
        //allocate memory for an int on the heap
        *p = i * 2;
        printf("%d \n", *p);
    }
    free(p); //deallocate memory
    //leaked memory - we only deallocated the last int
    return 0;
}
```

Memory leak detection

Memory leak – memorie care nu a fost dealocata. Alocat pe heap (cu malloc) dar niciodată dealocat (free).

Memory leak detection – identificarea unui Memory leak.

Chiar dacă zona de memoria ne-deallocata este mica, în timp aceste mici zone se adună și cauzează probleme serioase (Out of memory error).

Cu cat complexitatea aplicației este mai mare, cu atât găsirea acestor probleme este mai dificilă.

Există diferite instrumente care asistă programatorul în găsirea de Memory Leak.

In Visual Studio putem folosi CRT Library:

[https://msdn.microsoft.com/en-us/library/x98tx3cf\(v=vs.140\).aspx](https://msdn.microsoft.com/en-us/library/x98tx3cf(v=vs.140).aspx)

Adăugați în programul vostru, la început în aceasta ordine:

```
#define _CRTDBG_MAP_ALLOC  
#include <stdlib.h>  
#include <crtDBG.h>
```

Unde vreți să vedeați dacă aveți memory leak (în general la sfârșitul funcției main):

```
_CrtDumpMemoryLeaks();
```

Metoda `_CrtDumpMemoryLeaks()` tipărește toate alocările pentru care nu s-a executat `free()`

void*

O funcție care nu returnează nimic

```
void f() {  
}
```

Nu putem avea variabile de tip **void** dar putem folosi pointer la void - **void***

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    void* p;  
    int *i=malloc(sizeof(int));  
    *i = 1;  
    p = i;  
    printf("%d /n", *((int*)p));  
    long j = 100;  
    p = &j;  
    printf("%ld /n", *((long*)p));  
    free(i);  
    return 0;  
}
```

Se pot folos **void*** pentru a crea structuri de date care funcționează cu orice tip de elemente

Probleme: verificare egalitate între elemente de tip **void*** , copiere elemente

TAD – Tip abstract de date (ADT – Abstract data types)

TAD:

- Definește domeniul de valori
- Definește operațiile posibile (interfață)
- Definiția operațiilor este independent de implementare (abstract)
- Ascunde implementarea.

TAD implementat in C

`<adt.h> + <adt.c>`
interfață implementare

Interfața (header) conține declarații de funcții, fiecare funcție este specificată independent de implementare.

Codul client (cel care folosește TAD) nu are acces la detalii de implementare

Putem folosi diferite structuri de date pentru implementare

Este valabil pentru orice concept implementat în cod: separam interfața de detalii de implementare.

Vector dinamic

```

typedef void* Element;

typedef struct {
    Element* elems;
    int lg;
    int capacitate;
} VectorDinamic;

/*
 * Creaza un vector dinamic
 * v - vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic();

/*
 * Initializeaza vectorul
 * v - vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic() {
    VectorDinamic *v =
        malloc(sizeof(VectorDinamic));
    v->elems = malloc(INIT_CAPACITY *
sizeof(Element));
    v->capacitate = INIT_CAPACITY;
    v->lg = 0;
    return v;
}
/*
 * Elibereaza memoria ocupata de vector
 */
void distrugе(VectorDinamic *v) {
    int i;
    for (i = 0; i < v->lg; i++) {
//!!!!functioneaza corect doar daca
//elementele din lista NU refera
//memorie alocata dinamic
        free(v->elems[i]);
    }
    free(v->elems);
    free(v);
}

/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el);

/**
 * Returneaza elementul de pe pozitia data
 * v - vector
 * poz - pozitie, poz>=0
 * returneaza elementul de pe pozitia poz
 */
Element get(VectorDinamic *v, int poz);

/**
 * Alocă memorie aditională pentru vector
 */
void resize(VectorDinamic *v) {
    int nCap = 2*v->capacitate;
    Element* nElems =
        malloc(nCap*sizeof(Element));
    //copiez din vectorul existent
    int i;
    for (i = 0; i < v->lg; i++) {
        nElems[i] = v->elems[i];
    }
    //dealocam memoria ocupata de vector
    free(v->elems);
    v->elems = nElems;
    v->capacitate = nCap;
}

/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el) {
    if (v->lg == v->capacitate) {
        resize(v);
    }
    v->elems[v->lg] = el;
    v->lg++;
}

```

Pointer la funcții

```
void(*funcPtr)(int); // function returns void has an int parameter
int(*funcPtr2)(int,int); // function returns int has two int
parameters
```

```
void f(int a) {
    printf("%d\n", a);
}
```

```
int main() {
    void(*funcPtr)(int);
    funcPtr = f;
    funcPtr(6);
    return 0;
}
```

```
int sum(int a, int b) {
    return a + b;
}
```

```
int main() {
    int(*funcPtr2)(int,int);
    funcPtr2 = sum;
    int c = funcPtr2(6,3);
    printf("%d\n", c);
    return 0;
}
```

```
void func() {
    printf("func() called...");
}
int main() {
    void (*fp)(); // Define a function pointer
    fp = func; // Initialise it
    (*fp)(); // Dereferencing calls the function
    void (*fp2)() = func; // Define and initialize
    (*fp2)(); // call
}
```

Vector dinamic generic – pointer la funcții

Pentru o lista generica (funcționează cu orice fel de elemente) putem avea nevoie diferite funcții generice care oferă diferite operații pe elementele respective

Putem folosi pointer la funcții în structurile de date generice

```
typedef elem (*CopyFctPtr)(elem);
typedef int (*EqualsFctPtr)(elem, elem);
```

```
typedef void* ElemtType;
//function type for dealocating an element
typedef void(*DestroyF)(ElemtType);
```

```
typedef struct {
    ElemtType* elems;
    int lg;
    int capacitate;
} MyList;
```

```
/*
Deallocate list
*/
void destroy(MyList* l, DestroyF deallocate) {
    //free elements
    for (int i = 0; i < l->lg; i++) {
        deallocate(l->elems[i]);
    }
    //free list
    free(l->elems);
    l->lg = 0;
}
```

```
void testCopyList() {
    MyList l = createEmpty();
    add(&l, createPet("a", "b", 10));
    add(&l, createPet("a2", "b2", 20));
    MyList l2 = copyList(&l);
    assert(size(&l2) == 2);
    Pet* p = get(&l2, 0);
    assert(strcmp(p->type, "a") == 0);
    destroy(&l, destroyPet);
    destroy(&l2, destroyPet);
}
```

Urmașul limbajului C apărut în anii 80, dezvoltat de **Bjarne Stroustrup**
Bibliografie:

- B. Stroustrup, The C++ Programming Language
- B. Stroustrup, A Tour of C++

ISO standard din 1998 – isocpp.com

Limbajul C++

- compatibil cu C
- multiparadigmă, suportă paradigma orientat obiect (clase, obiecte, polimorfism, moștenire)
- tipuri noi – bool, referință
- spații de nume (namespace)
- şabloane (templates)
- excepții
- bibliotecă de intrări/ieșiri (IO Streams)
- STL (Standard Template Library)

Evoluție:

C with Classes (1979 Aduce concepte din Simula: clase, clase derivate)

C++ (1983 funcții virtuale, supraîncărcarea operatorilor)

C++ 98 (devine standard ISO – clase abstracte, metode statice/const)

C++ 11 (C++ 0x – auto, lambda, rvalue, move, constexpr, etc)

C++ 14 (C++ 1y – fix/upgrade template, lambda, constexpr, etc)

C++ 17 (C++ 1z – string_view, optional, file system, etc)

C++ 20 – standardul curent (module, coroutine,stl2)

C++ 23 – in procesul de standardizare

Tipuri de date predefinite

int, long, double, char, bool, void, etc.

Conversii intre tipuri

C++ este un limbaj puternic tipizat, in majoritatea cazurilor este nevoie de o conversie explicita type-casting cand dorim sa interpretam o valoare in mod

<code>char c = 23245;</code>	Conversie implicita – de evitat pe cat posibil poate cauza probleme (overflow, trunc) In general compilatorul da warning Se poate intampla la initializare, assignment, la transmiterea de parametrii
<code>int a = static_cast<int>(7.5);</code>	Conversie explicita Verificat la compilare Eroare de compilare daca conversia este imposibila (intre tipuri incompatibile)
<code>char c = (char)2000; <i>//functional notation</i> char c = char(2000);</code>	C-style cast. Elimina warningurile cauzate de conversii periculoase Este de evitat fiindca poate cauza probleme mai ales daca tipurile nu sunt compatibile

Tipul **bool** - domeniu de valori: adevarat (**true**) sau fals (**false**)

```
/** Verifica daca un numar e prim
 * nr numar intreg
 * return true daca nr e prim*/
bool ePrim(int nr) {
    if (nr <= 1)  return false;
    for (int i = 2; i < nr - 1; i++) {
        if (nr % i == 0)
            return false;
    }
    return true;
}
```

Tipul referință

```
data_type &reference_name;
```

```
int y = 7;  
int &x = y; //make x a reference to, or an alias of, y
```

Dacă schimbăm x se schimbă și y și invers, sunt doar două nume pentru același locație de memorie (alias)

Tipul referință este similar cu tipul pointer:

- sunt pointeri care sunt automat dereferențiate când folosim variabile
- nu se poate schimba adresa referită

```
/**  
 * C++ version  
 * Sum of 2 rational number  
 */  
void sum(Rational nr1, Rational nr2, Rational &rez) {  
    rez.a = nr1.a * nr2.b + nr1.b * nr2.a;  
    rez.b = nr1.b * nr2.b;  
    int d = gcd(rez.a, rez.b);  
    rez.a = rez.a / d;  
    rez.b = rez.b / d;  
}  
  
/**  
 * C version  
 * Sum of 2 rational number  
 */  
void sum(Rational nr1, Rational nr2, Rational *rez) {  
    rez->a = nr1.a * nr2.b + nr1.b * nr2.a;  
    rez->b = nr1.b * nr2.b;  
    int d = gcd(rez->a, rez->b);  
    rez->a = rez->a / d;  
    rez->b = rez->b / d;  
}
```

Declarare/Inițializare de variabile

Initializare variabile la declarare

<code>int b { 7 };</code>	Universal form varianta de preferat in modern in C++ Evită problemele legate de conversii prin care se pierde precizie (narrowing)
<code>int a = 7;</code>	Varianta “clasică” moștenită din C
<code>int d; //gresit</code>	Varianta greșita, compilează dar variabila este neinițializată (are o valoare aleatoare)

Nu folosiți variabile neinițializate. Preferați varianta cu {}

auto

Când definim o variabilă putem să nu specificăm explicit tipul (compilatorul deduce tipul din expresia de inițializare).

```
auto a = 7; //a e int
double b{7.4};
double c{1.4};
auto d = b+c; //d e double
```

auto este util pentru:

- a evita scrierea de nume lungi de tipuri
- a evita repetiția
- scriere de cod generic

Const

const semnalează compilatorului ca nu dorim sa schimbam valoarea variabilei

```
const int nr = 100;
```

Daca încercăm sa schimbăm valoarea lui nr rezulta o eroare la compilare

Este util pentru:

- a comunica ce face funcția (descrie mai precis interfața)
 - r1,r2 au fost transmise ca referință (pentru a evita copierea) dar sunt declarate **const** astfel este clar ca aceste valori nu se modifica in interiorul funcției
- compilatorul ajuta la evitarea unor greșeli (compilatorul verifică si dă eroare dacă se încearcă modificarea lui r1 sau r2)
- poate oferi posibilități de optimizare pentru compilator

```
typedef struct{
    int a;
    int b;
} Rationa;

void add(const Rationa& r1, const Rationa& r2, Rationa&
rez){
    ...
}
```

Folosiți **const** pentru a exprima idea de imutabil (nu se modifica)

Folosiți **const peste tot** unde are sens:

- compilatorul v-a ajuta in prinderea de bug-uri
- codul este mai ușor de înțeles de alții (codul exprima mai bine intenția programatorului)
- adăugați **const** de la început (e mai greu sa adaugi apoi)

Const Pointer

const type*

```
int j = 100;  
const int* p2 = &j;
```

Valoarea nu se poate schimba folosind pointerul. Se poate schimba adresa referită

```
const int* p2 = &j;  
cout << *p2 << "\n";  
p2 = &i;//change the memory address (valid)  
cout << *p2 << "\n";  
*p2 = 7;//change the value (compiler error)  
cout << *p2 << "\n";
```

type * const

```
int * const p3 = &j;
```

Valoarea se poate schimba folosind acest pointer dar adresa de memorie referită nu se poate schimba

```
int * const p3 = &j;  
cout << *p2 << "\n";  
//change the memory address (compiler error)  
p3 = &i;  
cout << *p3 << "\n";  
//change the value (valid)  
*p3 = 7;  
cout << *p3 << "\n";
```

const type* const

```
const int * const p4 = &j;
```

Atât adresa cât și valoarea sunt constante

Range for

<pre>int a[] = {0, 1, 2, 3, 4, 5}; for (auto v:a) { cout << v << "\n"; }</pre>	<pre>for (auto v:{0,1,2,3,4,5}) { cout << v << "\n"; }</pre>
--	---

Semantica: Pentru fiecare element din a, de la primul element până la ultimul, copiază în variabila v;

Range for poate fi folosit cu orice secvență de elemente

Dacă vrem să evităm copierea valorii din vectorul a în variabila v putem folosi **auto&**.

Dacă vream să evităm copierea dar vrem și să nu modificăm elementele **const auto&**

<pre>for (auto& v : a) { ++v; }</pre>	<pre>for (const auto& v : a) { cout << v << "\n"; }</pre>
---	---

IO library in C++ - definit in <iostream>

cin - corespunde intrări standard (stdin), tip **istream**

cout – corespunde ieșirii standard (stdout) , tip **ostream**

cerr - corespunde stderr, tip **ostream**

```
#include <iostream>
using namespace std;

void testStandardIostreams() {
    //prints Hello World!!! to the console
    cout << "Hello World!!!\n";
    int i = 0;
    cin >> i; //read an int from the console
    cout << "i=" << i << "\n"; // prints to the console
    //write a message to the standard error stream
    cerr << "Error message";
}
```

- Operația de scriere se realizează folosind operatorul “<<”, insertion operator
- citirea se realizează folosind operatorul “>>”, extraction operator

Paradigma de programare orientată-object

Este metodă de proiectare și dezvoltare a programelor:

- Oferă o abstractizare puternică și flexibilă
- Programatorul poate exprima soluția în mod mai natural (se concentrează pe structura soluției nu pe structura calculatorului)
- Descompune programul într-un set de obiecte, obiectele sunt elementele de bază
- Obiectele interacționează pentru a rezolva problema, există relații între clase
- Tipuri noi de date modelează elemente din spațiul problemei, fiecare obiect este o instanță a unui tip de date (clasa)

Un obiect este o entitate care:

- are o stare
- poate executa anumite operații (comportament)

Poate fi privit ca și o combinație de:

- date (attribute)
- metode

Concepție:

- obiect
- clasă
- metodă (mesaj)

Proprietăți:

- abstractizare
- încapsulare
- moștenire
- polimorfism

Caracteristici:

Încapsulare:

- capacitatea de a grupa date și comportament
 - controlul accesului la date/funcții,
 - ascunderea implementării
 - separare interfață de implementare

Moștenire

- Refolosirea codului

Polimorfism

- comportament adaptat contextului
 - în funcție de tipul actual al obiectului se decide metoda apelată în timpul execuției

Clase și obiecte în C++

Class: Un tip de dată definit de programator. Descrie caracteristicile unui lucru.

Grupează:

- date – **attribute**
- comportament – **metode**

Clasa este definită într-un fișier header (.h)

Sintaxă:

```
/**  
 * Represent rational numbers  
 */  
class Rational {  
public:  
    //methods  
    /**  
     * Add an integer number to the rational number  
     */  
    void add(int val);  
    /**  
     * multiply with a rational number  
     * r rational number  
     */  
    void mul(Rational r);  
private:  
    //fields (members)  
    int a;  
    int b;  
};
```

Definiții de metode

Metodele declarate în clasă sunt definite într-un fișier separat (.cpp)

Se folosește operatorul :: (scope operator) pentru a indica apartenența metodei la clasă

Similar ca și la module se separă declarațiile (interfața) de implementări

```
/**  
 * Add an integer number to the rational number  
 */  
void Rational::add(int val) {  
    a = a + val * b;  
}
```

Se pot defini metode direct în fișierul header. - **metode inline**

```
class Rational {  
public:  
    /**  
     * Return the numerator of the number  
     */  
    int getNumerator() {  
        return a;  
    }  
    /**  
     * Get the denominator of the fraction  
     */  
    int getDenominator() {  
        return b;  
    }  
private:  
    //fields (members)  
    int a;  
    int b;
```

Putem folosi metode inline doar pentru metode simple (fără cicluri)

Compilatorul inserează (inline) corpul metodei în fiecare loc unde se apelează metoda.

Obiect

Clasa descrie un nou tip de data.

Obiect - o instanță nouă (o valoare) de tipul descris de clasă

Declarație de obiecte

```
<nume_clasă> <identificator>;
```

- se alocă memorie suficientă pentru a stoca o valoare de tipul <nume_clasă>
- obiectul se initializează apelând constructorul implicit (cel fără parametrii)
- pentru initializare putem folosi și constructori cu parametri (dacă în clasă am definit constructor cu argumente)

```
Rational r1 = Rational(1, 2);
Rational r2{1, 3};
Rational r3;
cout << r1.toFloat() << "\n";
cout << r2.toFloat() << "\n";
cout << r3.toFloat() << "\n";
```

Acces la atribute (câmpuri)

În interiorul clasei

```
int getDenominator() {  
    return b;  
}
```

Când implementăm metodele avem acces direct la atribute

```
int getNumerator() {  
    return this->a;  
}
```

Putem accesa atributul folosind pointerul **this**. Util dacă mai avem variabile cu același nume în metodă (parametru, variabilă locală)

this: pointer la instanța curentă. Avem acces la acest pointer în toate metodele clasei, toate metodele membre din clasă au acces la **this**.

Putem accesa atributele și în afara clasei (dacă sunt vizibile)

- Folosind operatorul **'.'** **object.field**
- Folosind operatorul **'->'** dacă avem o referință (pointer) la obiect **object_reference->field** is a sau **(*object reference).field**

Protecția atributelor și metodelor .

Modificatori de acces: Defineste cine poate accesa atributele / metodele din clasa

public: poate fi accesat de oriunde

private: poate fi accesat doar în interiorul clasei

Atributele (reprezentarea) se declară private

Folosiți funcții (getter/setter) pentru accesa atributele

```
class Rational {  
    public:  
        /**  
         * Return the numerator of the number  
         */  
        int getNumerator() {  
            return a;  
        }  
        /**  
         * Get the denominator of the fraction  
         */  
        int getDenominator() {  
            return b;  
        }  
    private:  
        //fields (members)  
        int a;  
        int b;  
};
```

Constructor

Constructor: Metoda specială folosită pentru inițializarea obiectelor.

Metoda este apelată când se creează instanțe noi (se declara o variabilă locală, se creează un obiect folosind **new**)

Numele coincide cu numele clasei, nu are tip returnat

Constructorul alocă memorie pentru datele membre, inițializează atributele

```
class Rational {  
public:  
    Rational();  
private:  
    //fields (members)  
    int a;  
    int b;  
};
```

```
Rational::Rational() {  
    a = 0;  
    this->b = 1;  
}
```

Este apelat de fiecare dată când un obiect nou se creează – nu se poate crea un obiect fără a apela (implicit sau explicit) constructorul

Orice clasă are cel puțin un constructor (dacă nu se declară unu există un constructor implicit)

Intr-o clasă putem avea mai mulți constructori, constructorul poate avea parametrii.

Constructorul fără parametri este constructorul implicit (este folosit automat la declararea unei variabile, la declararea unei vector de obiecte)

Constructor cu parametrii

```
Rational::Rational(int a, int b) { Rational r2(1, 3);  
    this->a = a;  
    this->b = b;  
}
```

Constructori - Listă diferită de parametrii

Obiecte ca parametrii de funcții

Se folosește **const** pentru a indica tipul parametrului (in/out,return).

Dacă obiectul nu-și schimbă valoarea în interiorul funcției, el va fi apelat ca parametru **const**

```
/*
 * Copy constructor
 *
 */
Rational(const Rational &ot);
```

```
Rational::Rational(const Rational &ot) {
    a = ot.a;
    b = ot.b;
}
```

Folosirea **const** permite definirea mai precisă a contractului dintre apelant și metodă
Oferă avantajul că restricțiile impuse se verifică la compilare (eroare de compilare dacă încercam să modificăm valoarea/adresa)

Putem folosi **const** pentru a indica faptul ca metoda nu modifică obiectul (se verifică la compilare)

```
/*
 * Get the nominator
 */
int getUp() const;
```

```
/*
 * Get the nominator
 */
int Rational::getUp() const {
    return a;
}
```

```
/*
 * get the denominator
 */
int getDown() const;
```

```
/*
 * get the denominator
 */
int Rational::getDown() const {
    return b;
}
```

Supraîncărcarea operatorilor.

Definirea de semantică (ce face) pentru operatori uzuali când sunt folosiți pentru tipuri definite de utilizator.

```
/**  
 * Compute the sum of 2 rational numbers  
 * a,b rational numbers  
 * rez - a rational number, on exit will contain the sum of a and b  
 */  
void add(const Rational &nr);  
/**  
 * Overloading the + to add 2 rational numbers  
 */  
Rational operator +(const Rational& r) const;  
/**  
 * Sum of 2 rational number  
 */  
void Rational::add(const Rational& nr1) {  
    a = a * nr1.b + b * nr1.a;  
    b = b * nr1.b;  
    int d = gcd(a, b);  
    a = a / d;  
    b = b / d;  
}  
  
/**  
 * Overloading the + to add 2 rational numbers  
 */  
Rational Rational::operator +(const Rational& r) const {  
    Rational rez = Rational(this->a, this->b);  
    rez.add(r);  
    return rez;  
}
```

Operatori ce pot fi supraîncarcați:

+, -, *, /, +=, -=, *=, /=, %, %=, ++, --, ==, !=, <>, <=, >=, !, !=, &&, ||, <<, >>, <<=,
>>=, &, ^, |, &=, ^=, |=, ~, [] , () , ->*, →, new , new[] , delete, delete[],

Curs 4

- **C++ Core Guidelines**
- **Clase si obiecte**
- **Clase predefinite: string, vector**
- **Template**

Curs 3

- Gestiunea memoriei in C
- Încapsulare, abstractizare in C
- Limbajul de programare C++
 - Elemente de limbaj noi (c++ 11/14)

C++ Core Guidelines

Editors: [Bjarne Stroustrup](#), [Herb Sutter](#)

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>

„Inside C++, there is a much smaller and cleaner language struggling to get out.” - B. Stroustrup

Reguli recomandări C++ pentru a crea aplicații mai ușor de întreținut cu mai puține defecte

Propune să ajute programatorii să adopte Modern C++ (C++ 11, 14, 17)

- [I.2: Avoid global variables](#)
- [I.5: State preconditions \(if any\)](#)
- [I.7: State postconditions](#)
- [I.10: Use exceptions to signal a failure to perform a required task](#)
- [I.13: Do not pass an array as a single pointer](#)
- [I.22: Avoid complex initialization of global objects](#)
- [I.23: Keep the number of function arguments low](#)
- [I.24: Avoid adjacent unrelated parameters of the same type](#)
- [F.1: "Package" meaningful operations as carefully named functions](#)
- [F.2: A function should perform a single logical operation](#)
- [F.3: Keep functions short and simple](#)
- [F.8: Prefer pure functions](#)

Tema pentru acasă, de citit:

- [Introduction](#)
- [Philosophy](#)

Clase și obiecte în C++

Class: Un tip de dată definit de programator. Descrie caracteristicile unui lucru.

Grupează:

- date – **attribute**
- comportament – **metode**

Clasa este definită într-un fișier header (.h)

Implementarea metodelor se pun într-un fișier .cpp

Sintaxă:

```
//in file rational.h
/**
 * Represent rational numbers
 */
class Rational {
public:
    //methods
    /**
     * Add an integer number to the rational number
     */
    void add(int val);
    /**
     * multiply with a rational number
     * r rational number
     */
    void mul(Rational r);
private:
    //fields (members)
    int a;
    int b;
};

//in file rational.cpp
/**
 * Add an integer number to the rational number
 */
void Rational::add(int val) {
    a = a + val * b;
}
```

Definiții de metode

Metodele declarate în clasă sunt definite într-un fișier separat (.cpp)

Se folosește operatorul :: (scope operator) pentru a indica apartenența metodei la clasă

Similar ca și la module se separă declarațiile (interfața) de implementări

```
/**  
 * Add an integer number to the rational number  
 */  
void Rational::add(int val) {  
    a = a + val * b;  
}
```

Se pot defini metode direct în fișierul header. - **metode inline**

```
class Rational {  
public:  
    /**  
     * Return the numerator of the number  
     */  
    int getNumerator() {  
        return a;  
    }  
    /**  
     * Get the denominator of the fraction  
     */  
    int getDenominator() {  
        return b;  
    }  
private:  
    //fields (members)  
    int a;  
    int b;
```

Putem folosi metode inline doar pentru metode simple (fără cicluri)

Compilatorul inserează (inline) corpul metodei în fiecare loc unde se apelează metoda.

Obiect

Clasa descrie un nou tip de data.

Obiect - o instanță nouă (o valoare) de tipul descris de clasă

Declarație de obiecte

```
<nume_clasă> <identificator>;
```

- se alocă memorie suficientă pentru a stoca o valoare de tipul <nume_clasă>
- obiectul se inițializează apelând constructorul implicit (cel fără parametrii)
- pentru inițializare putem folosi și constructori cu parametri (dacă în clasă am definit constructor cu argumente)

```
Rational r1 = Rational{1, 2};  
Rational r2{1, 3};  
Rational r3;  
cout << r1.toFloat() << endl;  
cout << r2.toFloat() << endl;  
cout << r3.toFloat() << endl;  
Rational r4 = Rational(1, 2);
```

Acces la atribute (câmpuri)

În interiorul clasei

```
int getDenominator() {  
    return b;  
}
```

Când implementăm metodele avem acces direct la atribute

```
int getNumerator() {  
    return this->a;  
}
```

Putem accesa atributul folosind pointerul **this**. Util dacă mai avem variabile cu același nume în metodă (parametru, variabilă locală)

this: pointer la instanța curentă. Avem acces la acest pointer în toate metodele clasei, toate metodele membre din clasă au acces la **this**.

Putem accesa atributele și în afara clasei (dacă sunt vizibile)

- Folosind operatorul **'.'** **object.field**
- Folosind operatorul **'->'** dacă avem o referință (pointer) la obiect **object_reference->field** is a sau **(*object reference).field**

Protecția atributelor și metodelor .

Modificatori de acces: Definesc cine poate accesa atributele / metodele din clasă

public: poate fi accesat de oriunde

private: poate fi accesat doar în interiorul clasei

Atributele (reprezentarea) se declară private

Folosiți funcții (getter/setter) pentru accesa atributele

```
class Rational {  
    public:  
        /**  
         * Return the numerator of the number  
         */  
        int getNumerator() {  
            return a;  
        }  
        /**  
         * Get the denominator of the fraction  
         */  
        int getDenominator() {  
            return b;  
        }  
    private:  
        //fields (members)  
        int a;  
        int b;  
};
```

Constructor

Constructor: Metoda specială folosită pentru inițializarea obiectelor.

Metoda este apelată când se creează instanțe noi (se declară o variabilă locală, se creează un obiect folosind **new**)

Numele coincide cu numele clasei, nu are tip returnat

Constructorul alocă memorie pentru datele membre, inițializează atributele

```
class Rational {                                     Rational::Rational() {  
public:                                         a = 0;  
    Rational();                                this->b = 1;  
private:                                         }  
    //fields (members)  
    int a;  
    int b;  
};
```

Este apelat de fiecare dată când un obiect nou se creează – nu se poate crea un obiect fără a apela (implicit sau explicit) constructorul

Orice clasă are cel puțin un constructor (dacă nu se declară unu există un constructor implicit) Constructorul fără parametri este constructorul implicit (este folosit automat la declararea unei variabile, la declararea unei vector de obiecte)

Intr-o clasă putem avea mai mulți constructori, constructorul poate avea parametrii.

```
class Rational {                                     Rational::Rational() {  
public:                                         a = 0;  
    Rational();                                this->b = 1;  
    Rational(int a,int b);  
private:                                         }  
    int a;  
    int b;  
};  
  
Rational r1 = Rational{1, 2}; //apeleaza constructor cu 2 parametrii  
Rational r2{1, 3}; //apeleaza constructor cu 2 parametrii  
Rational r3; //apeleaza constructor implicit (0 parametrii)
```

Atributele clasei se pot inițializa (member initialization list) adăugând o lista înainte de corpul metodei (expresia intre ":" și "{" începutul corpului constructorului)

Varianta cu lista de inițializare are câteva beneficii:

- poate fi mai eficient (inițializează direct atributul - loc de default constructor apoi copiere)
- poate fi singura modalitate de inițializare (ex atribut referință sau const)

Constructori generați automat de compilator

Default constructor – constructor fără parametru

Se generează automat doar dacă nu scriem nici un constructor pentru clasa noastră

Constructor de copiere – constructor care primește un obiect de același tip (prin referință) și creează un obiect nou care este copia parametrului

Constructor folosit când se face o copie a obiectului

- la declarare de variabila cu initializare
- la transmitere de parametrii (prin valoare)
- când se returnează o valoare dintr-o metodă

```
class Persoana {  
    string nume;  
    int varsta;  
public:  
    //constructor default  
    Persoana() {}  
    //constructor de copiere  
    Persoana(const Persoana& ot) :nume{ ot.nume }, varsta{ ot.varsta } {}  
};
```

Constructorul de copiere în general se generează automat. Implementarea default face o copie pentru fiecare atribut al clasei. În cazul în care clasa are atribute alocate dinamic, dacă are pointeri sau necesită o logică specială la copiere atunci trebuie oferit o implementare custom.

= **default** Regulile legate de când se generează automat constructor de copiere / constructor default pot fi surprinzătoare – se poate cere explicit generarea automată

```
class Persoana {  
    string nume;  
    int varsta;  
public:  
    //constructor default  
    Persoana() = default;  
    //constructor de copiere  
    Persoana(const Persoana& ot) = default;  
};
```

= **delete** În unele cazuri dorim să împiedicăm crearea de copii pentru obiecte

```
class GRASPController {  
private:  
    vector<Persoana> all;  
public:  
    GRASPController() = default;  
    //nu se mai pot copia obiectele de tip GRASPController  
    GRASPController(const GRASPController& ot) = delete;
```

C++ Core Guidelines – Clase

- [C.1: Organize related data into structures \(structs or classes\)](#)
- [C.2: Use class if the class has an invariant; use struct if the data members can vary independently](#)
- [C.3: Represent the distinction between an interface and an implementation using a class](#)
- [C.4: Make a function a member only if it needs direct access to the representation of a class](#)
- [C.5: Place helper functions in the same namespace as the class they support](#)
- [C.7: Don't define a class or enum and declare a variable of its type in the same statement](#)
- [C.8: Use class rather than struct if any member is non-public](#)
- [C.9: Minimize exposure of members](#)

Obiecte ca parametrii de funcții

Se folosește **const** pentru a indica tipul parametrului (in/out,return).

Dacă obiectul nu-și schimbă valoarea în interiorul funcției, el va fi apelat ca parametru **const**

```
/*
 * Copy constructor
 *
 */
Rational(const Rational &ot);
```

Rational::Rational(const Rational &ot) {
 a = ot.a;
 b = ot.b;
}

Folosirea **const** permite definirea mai precisă a contractului dintre apelant și metodă
Oferă avantajul că restricțiile impuse se verifică la compilare (eroare de compilare dacă încercam să modificăm valoarea/adresa)

Putem folosi const pentru a indica faptul ca metoda nu modifică obiectul (se verifică la compilare)

```
/*
 * Get the nominator
 */
int getUp() const;
/** 
 * get the denominator
 */
int getDown() const;
```

```
/*
 * Get the nominator
 */
int Rational::getUp() const {
    return a;
}
/** 
 * get the denominator
 */
int Rational::getDown() const {
    return b;
}
```

Folosirea calificativului **const** - Const Correctness

Transmitere de parametri/valoare de return

Pentru clasele definite de noi se aplică aceleași reguli ca și la tipuri built in:

- transmitere prin referință (pas by reference – folosind tipul referință sau pointeri)
- transmitere prin valoare (pass by value – se transmite o copie a obiectului – se creează copie folosind copy constructor)
- Dacă returnăm un obiect prin valoare – se face o copie (copy constructor)

Putem folosi **const** pentru a descrie mai exact efectul funcției asupra parametrilor transmiși.

Dacă parametru nu este modificat în funcție ar trebui folosit **const const**

```
/*
 * Copy constructor
 *
 */
Rational(const Rational &ot);
```

Rational::Rational(const Rational &ot) {
 a = ot.a;
 b = ot.b;
}

const permite descrierea mai precisa a contractului (interfață) între metoda și cel care apelează metoda (codul client)

Restrictions impuse sunt verificate în timpul compilării (eroare de compilare dacă încercam să modificăm un obiect transmis prin **const**)

A trebui folosit **const** pentru a indica faptul că o metodă a unei clase nu modifica starea obiectului (quey method).

<pre>/** * Get the <u>nominator</u> */ int getUp() const; /** * get the denominator */ int getDown() const;</pre>	<pre>/** * Get the <u>nominator</u> */ int Rational::getUp() const { return a; } /** * get the denominator */ int Rational::getDown() const { return b; }</pre>
--	--

Dacă într-o funcție avem un parametru formal **const&**, funcția nu poate modifica starea obiectului => poate apela doar metode **const** al obiectului

Supraîncărcarea operatorilor

C++ permite definirea semantică pentru operatori asupra tipurilor definite de utilizator.

ex. ce se întâmplă dacă scriu `a+b` unde `a, b` sunt obiecte de un tip definit de utilizator

```
/*
 * Compute the sum of 2 rational numbers
 * a,b rational numbers
 * rez - a rational number, on exit will contain the sum of a and b
 */
void add(const Rational &nr);
/*
 * Overloading the + to add 2 rational numbers
 */
Rational operator +(const Rational& r) const;
/*
 * Sum of 2 rational number
 */
void Rational::add(const Rational& nr1) {
    a = a * nr1.b + b * nr1.a;
    b = b * nr1.b;
    int d = gcd(a, b);
    a = a / d;
    b = b / d;
}

/*
 * Overloading the + to add 2 rational numbers
 */
Rational Rational::operator +(const Rational& r) const {
    Rational rez = Rational(this->a, this->b);
    rez.add(r);
    return rez;
}
```

Lista operatorilor pe care putem să definim:

`+, -, *, /, +=, -=, *=, /=, %, %=, ++, --, ==, <>, <=, >=, !, !=, &&, ||, <<, >>, <<=, >>=, &, ^, |, &=, ^=, |=, ~, [] , () , ->*, →, new , new[], delete, delete[],`

Operatorul de asignment (=)

Operatorul = (copy assignment operator) înlocuiește conținutul unui obiect a cu o copie a obiectului b (b nu se modifica)

```
Rational& operator=(const Rational& ot) {
    this->a = ot.a;
    this->b = ot.b;
    return *this;
}

Rational& operator=(const Rational& ot) = default;

Rational r1{ 2,3 };
Rational r2;
r2 = r1;
r2.operator=(r1); //acelasi lucru cu linia de mai sus
```

Daca nu definim operatorul de assignment într-o clasa, compilatorul generează o varianta implicită. În cazul în care clasa gestionează memorie alocată dinamic, varianta implicită nu este corectă (nu funcționează cum ne-am așteptă)

```
class Pet {
private:
    char* nume;
    int varsta;
public:
    Pet& operator=(const Pet& ot) {
        if (this == &ot) {
            //sa evitam problemele la a = a;
            return *this;
        }
        char* aux = new char[strlen(ot.nume)+1];
        strcpy(aux, ot.nume);
        //eliboram memoria ocupata
        delete[] nume;
        nume = aux;
        varsta = ot.varsta;
        return *this;
    }
    Pet(const char* n, int varsta) {
        nume = new char[strlen(n) + 1];
        strcpy(nume, n);
        this->varsta = varsta;
    }
}
```

Abstract datatype – ascunderea reprezentării

Folosind in C struct/module/funcții nu putem definii tipuri abstracte de date care sunt ușor de folosit corect (cel care folosește TAD-ul nostru trebuie să ia în considerare aspecte ce țin de implementare, compilatorul nu ne e de mare ajutor – Ex. nu da eroare dacă uit să apelez funcția creazaPet).

Modificând struct `Pet` de la versiunea 1 la versiunea 2 necesită schimbări în întreaga aplicație.

Pentru a gestiona corect memoria este nevoie de funcții ajutătoare de alocare/dealocare și trebuie să le folosim consistent aceste funcții în toată aplicația. Aceste funcții trebuie folosite peste tot dar sunt doar convenții (reguli pe care trebuie să le ținem minte) de care trebuie să ținem cont, nu sunt urmările/forțate de compilator.

Dacă folosim clase și ascundem detaliiile de implementare (câmpuri private) putem schimba orice detaliu de implementare (reprezentare) fără a schimba codul care folosește clasa.

Șiruri de caractere in C++

Clasa string este parte biblioteca standard C++

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s; //create empty string
    //cin >> s;//read a string
    //cout << s;
    getline(cin, s, '\n');//read entire line
    cout << s;
    for (auto c : s) {//iterate characters
        cout << c << ",";
    }
    string s2{ "asdasd" };//create string
    s2 = s2 + s;//concatenate strings
    const char* pc = s2.c_str();//transform to C-String
    return 0;
}
```

```
#include <iostream>
#include <string>
class Persoana {
    std::string nume;
    std::string prenume;
public:
    Persoana(const std::string& n, const std::string& pn)
        :nume{ n }, prenume{ pn } {
    }

    std::string getNume() {
        return this->nume;
    }
};
int main()
{
    Persoana p{ "Ion", "Ion" };
    std::cout << p.getNume();
    return 0;
}
```

Vector dinamic in C++

Clasa vector face parte din biblioteca standard C++.

Este o lista implementata folosind structura de date vector dinamic.

Este o clasa parametrizata (template), se specifica tipul elementelor continute.

```
#include <iostream>
#include<vector>
using namespace std;

int main() {
    //creare si initializare vector
    vector<int> v{ 1,2,3,5,78,2 };
    v.push_back(8); //adauga element
    cout << v[4]<<'\n'; //access element dupa index
    //iterare elemente
    for (int el : v) {
        cout << el << ",";
    }
    cout<< '\n' << v.back() << '\n';//tipareste ultimul element
    v.pop_back(); //sterge ultimul element
    int first = v.front(); //returneaza primul element
    return 0;
}
```

Template

- permite crearea de cod generic
- in loc sa repetam implementarea unei funcției pentru fiecare tip de date, putem crea o funcție parametrizata după una sau mai multe tipuri
- o metoda convenienta de reutilizare de cod si de scriere de cod generic
- codul C++ se generează automat înainte de compilare, înlocuind parametru template cu tipul efectiv.

Function template:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

```
int sum(int a, int b) {                                template<typename T> T sum(T a, T b) {  
    return a + b;                                     return a + b;  
}  
  
double sum(double a, double b) {  
    return a + b;  
}  
  
int sum = sumTemp<int>(1, 2);  
cout << sum;  
double sum2 = sumTemp<double>(1.2, 2.2);  
cout << sum2;
```

- T este parametru template (template parameter), este un tip de date , argument pentru funcția sum
- Instantierea template-ului → crearea codului efectiv înlocuind T cu tipul int:
 - `int sum = sumTemp<int>(1, 2);`

```
Class template:
```

Putem parametriza o clasa după unu sau mai multe tipuri

Template-ul este ca o matră, înlocuind parametrul template cu un tip de date se obține codul c++, in acest caz o clasa.

Când cream o clasa template tot codul trebuie pus in header (fisierul .h). Implementarea actuala se creează la compilare. Compilatorul înlocuiește (textual) tipul dat ca parametru template și generează partea de implementare.

```
template<typename ElementType>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE( ElementType r);
    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    ElementType& get(int poz);
    /**
     * Give the size of the array
     * return the number of elements in the array
     */
    int getSize();
    //.....
private:
    ElementType *elems;
    int capacity;
    int size;
};

/**
 * Add an element to the dynamic array
 * r - is a rational number
 */
template<typename ElementType>
void DynamicArray<ElementType>:: addE( ElementType r){
    ensureCapacity(size + 1);
    elems[size] = r;
    size++;
}
//.....
```

Atribute statice in clasa (campuri/metode).

Atributele statice dintr-o clasa aparțin clasei nu instanței (obiectelor)
Caracterizează clasa, nu face parte din starea obiectelor

Ne referim la ele folosind operatorul scope “::”

Sunt asemănătoare variabilelor globale doar ca sunt definite în interiorul clasei – retine o singura valoare chiar dacă am multiple obiecte

Keyword : **static**

```
/**                                     Rational::nrInstances
 * New data type to store rational
numbers
 * we hide the data representation
*/
class Rational {
public:
    /**
     * Get the nominator
     */
    int getUp();
    /**
     * get the denominator
     */
    int getDown();
private:
    int a;
    int b;
    static int nrInstances = 0;
};
```

Clase/funcții **friend**.

- **friend** permite accesul unei funcții sau clase la câmpuri/metode private dintr-o clasa
- O metoda controlata de a încalcă încapsularea
- punând declarația funcției precedat de **friend** în clasa, funcția are acces la membrii privați ai clasei
- Funcția **friend** nu este membru a clasei (nu are acces la this), are doar acces la membrii privați din clasa
- O clasa B este **friend** cu class of class A daca are acces la membri privați al lui A. Se declara clasa cu cuvântul rezervat **friend** în fata.

Clasa friend

<pre>class ItLista { public: friend class Lista; ... }</pre>	<pre>template<typename E> class Set { friend class Set_iterator<E> ; }</pre>
--	--

Funcție friend

<pre>class List { public: friend void friendMethodName(int param); }</pre>
--

Cand folosim **friend**

putem folosi la supraîncărcarea operatorilor:

<pre>class AClass { private: friend ostream& operator<<(ostream& os, const AClass& ob); int a; }; ostream& operator<<(ostream& os, const AClass& ob) { return os << ob.a; }</pre>

Util și pentru:

<pre>class AClass { public: AClass operator+(int nr); //pentru: AClass a; a+7 private: int a; friend AClass operator+(int nr, const AClass& ob); //pentru: AClass a; 7+a };</pre>

Curs 5

- Template (Programare generica)
- STL – Standard Template Library
- Tratarea exceptiilor in C++

Curs 4

- C++ Core Guidelines
- Clase si obiecte
- Clase predefinite: string, vector
- Template

C++ Core Guideline Checker

Linter/Code analyzer: software care analizează codul sursa (code analysis) a unui program și semnalează automat erori de programare, buguri, cod suspect, probleme de formatare, etc.

Activăți: Proiect->Properties->Code Analysis -> Enable Code Analysis on Build

Selectați checkere: Proiect->Properties->Code Analysis->Microsoft-> Active rules combo

<https://docs.microsoft.com/en-us/cpp/code-quality/using-the-cpp-core-guidelines-checkers?view=msvc-160>

Puteți alege ce seturi de reguli (guideline) să se verifice, puteți aplica multiple reguli (în combo selectați „Choose multiple rule set”)

Pentru reguli din C++ core Guideline puteți selecta toate seturile de reguli cu numele: „Cpp Core....”

La compilare se efectuează analiza codului, se raportează warninguri pentru încălcări de reguli din guideline:

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. Erorile se vad în fereastra „Error List”.

Ex: **Avoid calling new and delete explicitly, use std::make_unique<T> instead (r.11**
<http://go.microsoft.com/fwlink/?LinkId=845485>).

Warningul conține un sumar plus un link către regula din guideline.

Fiecare regula are explicații, exemple de cod, motivație, soluție propusă pentru problema rezolvare.

Este o metodă bună pentru:

- Explora bunele practici în scrierea de aplicații industriale
- exploră guideline-ul și învăță despre bunele practici în scrierea de cod C++ Modern
- Explora diferite alternative disponibile în C++
- modernizează cod C++ existent

Pentru alte platforme: puteți folosi clang-tidy: <http://clang.llvm.org/extra/clang-tidy/>

Funcții/clase parametrizate - Template - programare generică

- permite crearea de cod generic
- în loc să repetăm implementarea unei funcții pentru fiecare tip de date, putem crea o funcție parametrizată după una sau mai multe tipuri
- o metodă convenientă de reutilizare de cod și de scriere de cod generic
- codul C++ se generează automat înainte de compilare, înlocuind parametru template cu tipul efectiv.
- Important: În cazul claselor metodelor care folosesc template tot codul ar trebui scris în fișierul .h Implementarea efectiva este generată de compilator

Function template:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

```
int sum(int a, int b) {                                template<typename T> T sumTemp(T a, T b) {  
    return a + b;                                     return a + b;  
}  
  
double sum(double a, double b) {  
    return a + b;  
}  
  
int sum = sumTemp<int>(1, 2);  
cout << sum;  
double sum2 = sumTemp<double>(1.2, 2.2);  
cout << sum2;
```

- T este parametru template (template parameter), este un tip de date, argument pentru funcția sum
- Instantierea templatului → crearea codului efectiv înlocuind T cu tipul int:
 - `int sum = sumTemp<int>(1, 2);`

Class template:

Putem parametriza o clasa după unu sau mai multe tipuri

Templatul este ca o matrice, înlocuind parametrul template cu un tip de date se obține codul c++, in acest caz o clasa.

```
template<typename ElementType>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE( ElementType r);
    /**
     * Delete the element from the given position
     * poz - the position of the elem to be deleted, poz>=0;poz<size
     * returns the deleted element
     */
    ElementType& deleteElem(int poz);

    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    ElementType& get(int poz);
    /**
     * Give the size of the array
     * return the number of elements in the array
     */
    int getSize();
    /**
     * Clear the array
     * Post: the array will contain 0 elements
     */
    void clear();
private:
    ElementType *elems;
    int capacity;
    int size;
};
```

In general parametrizarea se face după un tip, dar putem avea și parametri valoare pentru un template

```
//in fisierul buffer.h                                     //instantiere
template<typename T,int N>                                Buffer<Pet, 10> buff;
class Buffer {
private:
    T elems[N];
public:
    T& operator[](int poz);
};

//in cazul calaselor template
//inclusiv definitiile se pun in header
template<typename T, int N>
T& Buffer<T, N>::operator[](int poz) {
    if (poz < 0 || poz >= N) {
        ...
    }
    return elems[poz];
}
```

Programare generica

Mecanismul de template permite:

- parametrizare după un tip (sau chiar o valoare) fără a pierde din precizie.
Permite scrierea de algoritmi generali (independent de tipul datelor)
- Verificare de tip întârziata. Se verifica la compilare în momentul instantiei templatului daca tipul primit ca parametru template are metodele dorite (asemănător cu duck typing dar e la compilare)
- Posibilitatea de a transmite constante si de a face calcule in timpul compilării
- codul rezultat este eficient (la instantierea templatului se generează cod C++ care este compilat/optimizat de compilator ca si orice cod scris de programator)

Programare generica se referă la crearea de algoritmi generali unde prin general se înțelege ca algoritmul poate lucra cu orice tipuri de date care satisfac un set de cerințe (au un set de operații)

Tipuri abstracte de date (Abstract Data Types)

ADT

- separat interfața (ce vede cel care folosește) de implementare (cum e implementat)
- specificații abstracte (fără referire la detaliu de implementare)
- ascundere detaliu de implementare (data protection)

Clase

- header: conține declarația de clasa / metode
- specificații pentru fiecare metoda
- folosind modifierul private , reprezentarea (câmpurile clasei), metodele care sunt folosite doar inter pot fi protejate de restul aplicației (nu sunt vizibile în afara clasei)

Exemplu: Variante de vector dinamic generic (acomodează orice tip de element)

- `typedef Telem = <type name>`
 - nu pot avea în același program liste pentru 2 sau mai multe tipuri de elemente (int, Rational)
- implementare cu `void*`
 - nu pot adăuga constante (1, 3.5) pot adăuga doar adrese
 - gestiunea memoriei devine mai dificilă (similar cu varianta C)
 - în multe locuri trebuie să folosesc `cast`
 - pot adăuga în același lista adrese la elemente de tipuri diferite
- implementare cu template
 - elimina neajunsurile abordărilor anterioare
 - lista poate conține atât adrese ca și obiectele, pot adăuga valori simple
 - pot instanția clasa template pentru oricâte tipuri

Atribute statice in clasa (câmpuri/metode).

Atributele statice dintr-o clasa aparțin clasei nu instanței (obiectelor)

Caracterizează clasa, nu face parte din starea obiectelor

Ne referim la ele folosind operatorul scope “::”

Sunt asemănătoare variabilelor globale doar ca sunt definite în interiorul clasei – retine o singură valoare chiar dacă am multiple obiecte

Obs: Variabilele statice trebuie inițializate în fișierul .cpp

Keyword : **static**

```
/*
 * New data type to store rational numbers
 * we hide the data representation
 */
class Rational {
public:
    /**
     * Get the nominator
     */
    int setUp();
    /**
     * get the denominator
     */
    int getDown();

    // functie statica
    static int getNrInstance(){
        return nrInstances;
    }

private:
    int a;
    int b;
    // declarare membru static
    static int nrInstances;
};
```

```
//in cpp
// initializare membru static (obligatoriu în cpp dacă nu este const)
int Rational:: nrInstances =0;
```

Clase/functii **friend**.

- **friend** permite accesul unei funcții sau clase la câmpuri/metode private dintr-o clasa
- O metoda controlata de a încalcă încapsularea
- punând declarația funcției precedat de **friend** în clasa, funcția are acces la membrii privați ai clasei
- Funcția **friend** nu este membru a clasei (nu are acces la this), are doar acces la membrii privați din clasa
- O clasa B este **friend** cu class of class A daca are acces la membri privati al lui A. Se declara clasa cu cuvântul rezervat **friend** în fata.

Clasa friend

```
class ItLista {  
public:  
    friend class Lista;  
...  
template<typename E>  
class Set {  
    friend class Set_iterator<E> ;
```

Functie friend

```
class List {  
public:  
    friend void friendMethodName(int param);
```

Când folosim **friend**

putem folosi la supraincarcarea operatorilor:

```
class AClass {  
private:  
    friend ostream& operator<<(ostream& os, const AClass& ob);  
    int a;  
};  
ostream& operator<<(ostream& os, const AClass& ob) {  
    return os << ob.a;  
}
```

Util si pentru:

```
class AClass {  
public:  
    AClass operator+(int nr); //pentru: AClass a;  a+7  
private:  
    int a;  
    friend AClass operator+(int nr, const AClass& ob); //pentru: AClass a;  7+a  
};
```

Standard Template Library (STL)

- The Standard Template Library (STL), este o bibliotecă de clase C++, parte din C++ Standard Library
- Oferă structuri de date și algoritmi fundamentali, folosiți la dezvoltarea de programe în C++
- STL oferă componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template).
- STL a fost conceput astfel încât componentele STL se pot compune cu ușurință fără a sacrifica performanță (generic programming)
- STL conține clase pentru:
 - containere, iteratori
 - algoritmi, function objects
 - allocators

Selected Standard Library Headers	
<code><algorithm></code>	<code>copy(), find(), sort()</code>
<code><array></code>	<code>array</code>
<code><chrono></code>	<code>duration, time_point</code>
<code><cmath></code>	<code>sqrt(), pow()</code>
<code><complex></code>	<code>complex, sqrt(), pow()</code>
<code><forward_list></code>	<code>forward_list</code>
<code><iostream></code>	<code>fstream, ifstream, ofstream</code>
<code><future></code>	<code>future, promise</code>
<code><ios></code>	<code>hex, dec, scientific, fixed, defaultfloat</code>
<code><iostream></code>	<code>istream, ostream, cin, cout</code>
<code><map></code>	<code>map, multimap</code>
<code><memory></code>	<code>unique_ptr, shared_ptr, allocator</code>
<code><random></code>	<code>default_random_engine, normal_distribution</code>
<code><regex></code>	<code>regex, smatch</code>
<code><string></code>	<code>string, basic_string</code>
<code><set></code>	<code>set, multiset</code>
<code><sstream></code>	<code>istrstream, ostrstream</code>
<code><stdexcept></code>	<code>length_error, out_of_range, runtime_error</code>
<code><thread></code>	<code>thread</code>
<code><unordered_map></code>	<code>unordered_map, unordered_multimap</code>
<code><utility></code>	<code>move(), swap(), pair</code>
<code><vector></code>	<code>vector</code>

* A tour of c++, Bjarne Stroustrup

Containeri

Un container este o grupare de date în care se pot adăuga (insera) și din care se pot șterge (extrage) obiecte. Implementările din STL folosesc şablonane ceea ce oferă o flexibilitate în ceea ce privește tipurile de date ce sunt suportate

Containerul gestionează memoria necesară stocării elementelor, oferă metode de acces la elemente (direct și prin iteratori)

Toate containerele oferă funcționalități (metode):

- accesare elemente (ex.: `[]`)
- gestiune capacitate (ex.: `size()`)
- modificare elemente (ex.: `insert`, `clear`)
- iterator (begin(), end())
- alte operații (ie: `find`)

Decizia în alegerea containerului potrivit pentru o problemă concretă se bazează pe:

- funcționalitățile oferite de container
- eficiența operațiilor (complexitate).

Containere - Clase template

- Container de tip secvență (Sequence containers): **vector<T>**, **deque<T>**, **list<T>**
- Adaptor de containere (Container adaptors): **stack<T, ContainerT>**, **queue<T, ContainerT>**, **priority_queue<T, ContainerT, CompareT>**
- Container asociativ (Associative containers): **set<T, CompareT>**, **multiset<T, CompareT>**, **map<KeyT, ValueT, CompareT>**, **multimap<KeyT, ValueT, CompareT>**, **bitset<T>**

Standard Container Summary	
vector<T>	A variable-size vector (§9.2)
list<T>	A doubly-linked list (§9.3)
forward_list<T>	A singly-linked list
deque<T>	A double-ended queue
set<T>	A set (a map with just a key and no value)
multiset<T>	A set in which a value can occur many times
map<K,V>	An associative array (§9.4)
multimap<K,V>	A map in which a key can occur many times
unordered_map<K,V>	A map using a hashed lookup (§9.5)
unordered_multimap<K,V>	A multimap using a hashed lookup
unordered_set<T>	A set using a hashed lookup
unordered_multiset<T>	A multiset using a hashed lookup

* A tour of c++, Bjarne Stroustrup

Container de tip secvență (Sequence containers):

Vector, Deque, List sunt containere de tip secvență, folosesc reprezentări interne diferite, astfel operațiile uzuale au complexități diferite

- Vector (Dynamic Array):
 - elementele sunt stocate secvențial în zone continue de memorie
 - Vector are performanțe bune la:
 - Accesare elemente individuale de pe o poziție dată (constant time).
 - Iterare elemente în orice ordine (linear time).
 - Adăugare/Ștergere elemente de la sfârșit (constant amortized time).
- Deque (double ended queue) - Coadă dublă (completă)
 - elementele sunt stocate în blocuri de memorie (chunks of storage)
 - Elementele se pot adăuga/șterge eficient de la ambele capete
- List
 - implementat ca și listă dublă înlățuită
 - List are performanțe bune la:
 - Ștergere/adăugare de elemente pe orice pozitie (constant time).
 - Mutarea de elemente sau secvențe de elemente în liste sau chiar și între liste diferite (constant time).
 - Iterare de elemente în ordine (linear time).

Operații / complexitate

<pre>#include <vector> void sampleVector() { vector<int> v; v.push_back(4); v.push_back(8); v.push_back(12); v[2] = v[0] + 2; int lg = v.size(); for (int i = 0; i<lg; i++) { cout << v.at(i) << " "; } }</pre>	<pre>#include <deque> void sampleDeque() { deque<double> dq; dq.push_back(4); dq.push_back(8); dq.push_back(12); dq[2] = dq[0] + 2; int lg = dq.size(); for (int i = 0; i<lg; i++) { cout << dq.at(i) << " "; } }</pre>	<pre>#include <list> void sampleList() { list<double> l; l.push_back(4); l.push_back(8); l.push_back(12); while (!l.empty()) { cout << " " << l.front(); l.pop_front(); } }</pre>
--	--	---

Vector : timp constant O(1) random access; insert/delete de la sfârșit

Deque: timp constant O(1) insert/delete la oricare capat

List: timp constant O(1) insert / delete oriunde în listă

Vector vs Deque

- Accesul la elemente de pe orice poziție este mai eficient la vector
- Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
- Pentru liste mari Vector aloca zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Container asociativ (Associative containers):

Sunt eficiente în accesare elementelor folosind chei (nu folosind poziții ca și în cazul containerelor de tip secvență).

- set
 - mulțime - stochează elemente distincte. Elementele sunt folosite și ca și cheie
 - nu putem avea două elemente care sunt egale
 - se folosește arbore binar de căutare ca și reprezentare internă
- Map, unordered_map
 - dicționar - stochează elemente formate din cheie și valoare
 - nu putem avea chei duplicate
- Bitset
 - container special pentru a stoca biți (elemente cu doar 2 valori posibile: 0 sau 1,true sau false, ...).

```
void sampleMap() {  
    map<int, Product*> m;  
    Product *p = new Product(1, "asdas", 2.3);  
    //add code <=> product  
    m.insert(pair<int, Product*>(p->getCode(), p));  
  
    Product *p2 = new Product(2, "b", 2.3);  
    //add code <=> product  
    m[p2->getCode()] = p2;  
  
    //lookup  
    cout << m.find(1)->second->getName()<<endl;  
    cout << m.find(2)->second->getName()<<endl;  
}
```

Iteratori in STL

Iterator: obiect care gestionează o poziție (curentă) din containerul asociat. Oferă suport pentru traversare (++,-), derefențiere (*it).

Iteratorul este un concept fundamental în STL, este elementul central pentru algoritmi oferiti de STL.

Fiecare container STL include funcții membre begin() și end(), perechea de iteratori descrie o secvență [first, last) - interval deschis: first inclusiv, last exclusiv

end() - arată după ultimul element, nu este corect să incercăm să luăm valoarea (*)

```
void sampleIterator() {
    vector<int> v;
    v.push_back(4);
    v.push_back(8);
    v.push_back(12);
    //Obtain an the start of the iteration
    vector<int>::iterator it = v.begin();
    while (it != v.end()) {
        //dereference
        cout << (*it) << " ";
        //go to the next element
        it++;
    }
    cout << endl;
}
```

Permite decuplarea între algoritmi și containere

Existe mai multe tipuri de iteratori:

- iterator input/output (istream_iterator, ostream_iterator)
- forward iterators, iterator bidirectional, iterator random access
- reverse iterators

In funcție de tipul iteratorului putem avea diferite operații suportate: ++,!-,*(forward) – (bidirectional) it+3, it-6 (random access)

Iterator adaptors

```
vector<int> v2(6); //trebuie sa pregatim loc pentru elemente
copy(v.begin(), v.end(), v2.begin());
vector<int> v3;
//back_inserter este un adaptor - face push_back la *it=elem
copy(v.begin(), v.end(), back_inserter(v3));
vector<int> v3;
//!!! gresit, functia copy nu face loc pentru elemente in vectorul destinatie
copy(v.begin(), v.end(), v3.begin()); //segmentation fault
```

Input iterator adapter

```
int main() {
    using namespace std;
    //create a istream iterator using the standard input
    istream_iterator<int> start(cin);
    istream_iterator<int> end;
    vector<int> v4;
    //copy readed ints into v4 (until EOF(ctrl+z) or cin fail)
    copy(start, end, back_inserter(v4));
    for (int e : v4) {
        cout << e << ",";
    }
}
```

Implementare iterator VectorDinamic

```
class IteratorVector {
private:
    const VectorDinamic& v;
    int poz = 0;
public:
    IteratorVector(const VectorDinamic& v) :v{v} {}
    bool valid() const {
        return poz < v.size();
    }
    Element& element() const {
        return v.elems[poz];
    }
    void next() {
        poz++;
    }
};
```

Putem suprascrie operatori: *, ++, ==, != pentru a crea iteratori similar cu cei din STL (ForwardIterator)

Daca dorim sa folosim vectorul intr-un **foreach** avem nevoie de metodele begin() si end()

<pre>IteratorVector VectorDinamic::begin() const { return IteratorVector(*this); } IteratorVector VectorDinamic::end() const { return IteratorVector(*this, lg);</pre>	<pre>Element& operator*() { return element(); } IteratorVector& operator++() { next(); return *this;</pre>
---	--

Putem folosi:

<pre>//testam iteratorul auto it = v.begin(); while (it != v.end()) { auto p = *it; assert(p.getPrice() > 0); ++it; }</pre>	<pre>for (auto& p : v) { std::cout << p.getType() << std::endl; assert(p.getPrice() > 0); }</pre>
--	--

STL Algorithms

Există o mulțime de algoritmi implementați în STL. Ele se află în modulul `<algorithm>` și în namespace-ul std.

Selected Standard Algorithms	
<code>p=find(b,e,x)</code>	<code>p</code> is the first <code>p</code> in <code>[b:e)</code> so that <code>*p==x</code>
<code>p=find_if(b,e,f)</code>	<code>p</code> is the first <code>p</code> in <code>[b:e)</code> so that <code>f(*p)==true</code>
<code>n=count(b,e,x)</code>	<code>n</code> is the number of elements <code>*q</code> in <code>[b:e)</code> so that <code>*q==x</code>
<code>n=count_if(b,e,f)</code>	<code>n</code> is the number of elements <code>*q</code> in <code>[b:e)</code> so that <code>f(*q,x)</code>
<code>replace(b,e,v,v2)</code>	Replace elements <code>*q</code> in <code>[b:e)</code> so that <code>*q==v</code> by <code>v2</code>
<code>replace_if(b,e,f,v2)</code>	Replace elements <code>*q</code> in <code>[b:e)</code> so that <code>f(*q)</code> by <code>v2</code>
<code>p=copy(b,e,out)</code>	Copy <code>[b:e)</code> to <code>[out:p)</code>
<code>p=copy_if(b,e,out,f)</code>	Copy elements <code>*q</code> from <code>[b:e)</code> so that <code>f(*q)</code> to <code>[out:p)</code>
<code>p=move(b,e,out)</code>	Move <code>[b:e)</code> to <code>[out:p)</code>
<code>p=unique_copy(b,e,out)</code>	Copy <code>[b:e)</code> to <code>[out:p)</code> ; don't copy adjacent duplicates
<code>sort(b,e)</code>	Sort elements of <code>[b:e)</code> using <code><</code> as the sorting criterion
<code>sort(b,e,f)</code>	Sort elements of <code>[b:e)</code> using <code>f</code> as the sorting criterion
<code>(p1,p2)=equal_range(b,e,v)</code>	<code>[p1:p2)</code> is the subsequence of the sorted sequence <code>[b:e)</code> with the value <code>v</code> ; basically a binary search for <code>v</code>
<code>p=merge(b,e,b2,e2,out)</code>	Merge two sorted sequences <code>[b:e)</code> and <code>[b2:e2)</code> into <code>[out:p)</code>

A tour of C++, Bjarne Stroustrup

Este doar un subset, o listă mai completă de algoritmi disponibili găsiți aici:

<http://en.cppreference.com/w/cpp/algorithm>

În general sunt funcții care primesc o pereche de iteratori (`begin()`, `end()`).

Perechea de iteratori descrie o secvență de elemente (un range) – $[a,b)$

Astfel acești algoritmi pot fi folosiți cu orice container STL, cu array-uri (`int a[]`), cu pointeri.

```
#include <vector>
#include <algorithm>

int main(){
    std::vector<int> v{ 3,2,8,1,4,5,7,6 };
    std::sort(v.begin(),v.end());
    for (auto a : v) {
        std::cout << a << " ";
    }
    std::cout << std::endl;
}

#include <algorithm>

int main(){
    int v[]{ 3,2,8,1,4,5,7,6 };
    std::sort(v,v+8);
    for (auto a : v) {
        std::cout << a << " ";
    }
    std::cout << std::endl;
}
```

Predicat / Functor

Majoritatea algoritmilor in STL au ca parametru un predicat.

Predicat poate fi:

- o funcție care returnează bool

```
bool simpleFct(int a) {
    return a % 2 == 0;
}
.....
int nrPare = count_if(v.begin(), v.end(), simpleFct);
```

- **function object/functor:** orice obiect care supraîncarcă operatorul "()" și returnează bool

```
class FunctionObj {
public:
    bool operator()(int a){return a % 2 == 0;}
};
.....
int nrPare = count_if(v.begin(), v.end(), FunctionObj{});
```

- **funcție lambda**

```
int nrPare = count_if(v.begin(), v.end(), [] (int a) {return a % 2 == 0;});
```

Exista functori gata definiti in STL (#include <functional>)

```
#include <functional>
....
vector<int> v{ 1,2,3,4,5,6 };
sort(v.begin(), v.end(), less<int>());
....
sort(v.begin(), v.end(), greater<int>());
```

Functii lambda

Functii anume (fără nume), se pot defini direct în locul în care e nevoie de o funcție

Foarte utili în cazul algoritmilor STL (și nu numai)

Practic este o metodă ușoară de a crea functori (e doar o sintaxă simplificată, compilatorul generează o clasa care suprascrie operator())

Sintaxa:

[capture-list](params){body}

capture-list – care sunt variabilele din scopul curent care se vad în interiorul funcției lambda

poate fi vid [] - nu captează nimic – nu se vede nici o variabilă

[=] - se vad toate variabilele din afara în corpul funcției lambda, se transmit prin valoare

[&] - se vad toate variabilele din afara în corpul funcției lambda, se transmit prin referință

[a,&b] – se vede a (prin valoare) și b (prin referință)

params – parametrii funcției lambda – exact ca și în cazul funcțiilor obișnuite

body – corpul funcției lambda

```
sort(v.begin(), v.end(), [](const Pet& p1, const Pet& p2) {
    return strcmp(p1.getType(),p2.getType());
});
```

Functii care primesc ca parametru alte functii (higher order functions, callback)

Parametru formal poate fi pointer la functie:

```
vector<Pet> PetStore::generalSort(bool(*maiMicF)(const Pet&, const Pet&)) {
    vector<Pet> v{ rep.getAll() };//fac o copie
    for (size_t i = 0; i < v.size(); i++) {
        for (size_t j = i + 1; j < v.size(); j++) {
            if (!maiMicF(v[i], v[j])) {
                //interschimbam
                Pet aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
    return v;
}
```

In acest caz putem apela cu o functie cu același semnatura sau un lambda care nu captează nimic

```
bool cmpSpecies(const Pet& p1, const Pet& p2) {
    return p1.getSpecies() < p2.getSpecies();
}

...
vector<Pet> PetStore::sortBySpecies() {
    return generalSort(cmpSpecies);
}

vector<Pet> PetStore::sortBySpecies() {
    return generalSort([](const Pet&p1, const Pet&p2) {
        return p1.getSpecies() < p2.getSpecies();
    });
}
```

Daca vrem sa permitem apelul cu functii lambda care capteaza ceva folosim clasa `function`:

```
#include <functional>
vector<Pet> PetStore::filtreaza(function<bool(const Pet&)> fct) {
    vector<Pet> rez;
    for (const auto& pet : rep.getAll()) {
        if (fct(pet)) {
            rez.push_back(pet);
        }
    }
    return rez;
}

...
vector<Pet> PetStore::filtrarePret(int pretMin, int pretMax) {
    return filtreaza([](const Pet& p) {
        return p.getPrice() >= pretMin && p.getPrice() <= pretMax;
    });
}
```

Tratarea exceptiilor

Situări anormale apar în timpul execuției (nu există fișier, nu mai există spațiu pe disk, etc), trebuie să trătam aceste situații

In general situații în care o funcție/metoda nu poate efectua operația promisa

Problema: Cum raportam eroarea, cum propagam eroarea (in cazul in care avem un lanț de apeluri, cum separam partea in care apare eroarea de partea unde tratam eroarea tratarea

Logica aplicației (in general aici putem trata eroarea – mesaj pe

Layer consola/fereastra, raspuns HTTP, etc)

Layer

Layer

Layer

Low level implementation - in general aici apar erori (nu pot scrie in fisier, nu mai am memorie, etc), in general in aceasta parte a aplicatiei nu vrem/putem rezolva situatia

Obs: este valabil in general -nu doar in cazul UI-GraspController-Repository. Codul este in general organizat pe nivele logice, actiunea efectuata este rezultatul unui lant de apeluri de metode

Soluții pentru raportarea de erori

In limbaje fără mecanismul de excepții soluțiile sunt (ex in C):

- returnare cod de eroare
- setarea de flag-uri (variabile globale)

Probleme cu aceste abordări:

- implicit se ignora eroarea (daca nu verific valoare de return sau flagurile)
- se compune greu (diferite coduri, daca pe stack-ul de apel cineva ignoră eroarea nu mai e propagat)
- fluxul normal este amestecat cu tratarea situațiilor excepționale

Alternative:

Putem folosi un tip de date special care modelează idea: returnam optional ceva. In C++ (începând cu standardul c++ 17) există tipul de date **std::optional<T>**, în headerul: #include <optional>

Clase similare există și în alte limbaje (Optional, Maybe, etc)

<https://en.cppreference.com/w/cpp/utility/optional>

Ex. `optional<Pet>` poate conține un obiect Pet sau poate e gol. Putem folosi de exemplu la funcția de căutare după id (`optional<Pet> find(int id);`) funcția să returneze un optional gol dacă nu s-a gasit Pet.

Putem folosi excepții (raportăm eroarea aruncând o excepție)

Tratarea excepțiilor în c++

excepții - situații anormale ce apar în timpul execuției

tratarea excepțiilor - mod organizat de a gestiona situațiile exceptionale ce apar în timpul execuției

O excepție este un eveniment ce se produce în timpul execuției unui program și care provoacă întreruperea cursului normal al execuției.

Elemente:

- **try block** marchează blocul de instrucțiuni care poate arunca excepții.
- **catch block** bloc de instrucțiuni care se executa în cazul în care apare o excepție (tratează excepția).
- Instrucțiunea **throw** mecanism prin care putem arunca (genere excepții) pentru a semnala codului client apariția unei probleme.

```
void testTryCatch() {  
    ...  
    try {  
        //code that may throw an exception  
        ErrorClass errObj;  
        throw errObj;  
        //code  
    } catch (ErrorClass& e){//e- ca si un param. de functie  
        //error handling - daca eroarea era de tip ErrorClass  
        // sau orice alt tip derivat din ErrorClass  
        cout << "Error occurred.";  
    } catch (...) {  
        //error handling - intra aici la orice eroare  
    }  
}
```

Tratarea exceptiilor

- Codul care susceptibil de a arunca exceptie se pune intr-un bloc de try.
- Adaugam unu sau mai multe sectiuni de **catch**. Blocul de instructiuni din interiorul blocului catch este responsabil sa trateze exceptia aparuta.
- Daca codul din interiorul blocului try (sau orice cod apelat de acesta) arunca exceptie, se transfera execuția la clauza catch corespunzătoare tipului exceptiei aparute. (exception handler)

```
void testTryCatchFlow(bool throwEx) {  
    // some code  
    try {  
        cout << "code before the exception" << endl;  
        if (throwEx) {  
            cout << "throw (raise) exception" << endl;  
            throw MyError();  
        }  
        cout << "code after the exception" << endl;  
    } catch (MyError& error) {  
        cout << "Error handling code " << endl;  
    }  
}  
  
testTryCatchFlow(0);  
testTryCatchFlow(1);
```

- Clauza catch nu trebuie neaparat sa fie in același metodă unde se arunca exceptia. Exceptia se propagă.
- Când se arunca o exceptie, se caută cel mai apropiat bloc de **catch** care poate trata exceptia ("unwinding the stack").
- Dacă nu avem clauză **catch** în funcția în care a apărut exceptia, se caută clauza **catch** în funcția care a apelat funcția.
- Căutarea continuă pe stack până se găsește o clauză **catch** potrivită. Dacă exceptia nu se tratează (nu există o clauză **catch** potrivită) programul se oprește semnalând eroarea apărută.
- Potrivirea cu clauza catch:
 - foarte asemănător cu potrivirea intre parametru actual și parametru formal

Excepții - obiecte

- Când se aruncă o excepție se poate folosi orice tip de date.

Tipuri predefinite (int, char, etc) sau tipuri definite de utilizator (obiecte). Nu este recomandat să folosim pointeri (chiar dacă este posibil)

- Este recomandat să se creeze clase pentru diferite tipuri de excepții care apar în aplicație
- Se arunca un obiect (**nu referință sau pointer**) și se prinde prin referință (pentru a evita copierea)
- Obiectul excepție este folosit pentru a transmite informații despre eroarea apărută

```
class POSError {
public:
    POSError(string message) :
        message(message) {
    }
    const string& getMessage() const {
        return message;
    }
private:
    string message;
};
```

```
class ValidationError: public POSError {
public:
    ValidationError(string message) :
        POSError(message) {
    }
};
```

```
void Sale::addSaleItem(double quantity, const Product& product) {
    if (quantity < 0) {
        throw ValidationError("Quantity must be positive");
    }
    saleItems.push_back(SaleItem{quantity, product});
}

try {
    pos.enterSaleItem(quantity, product);
    cout << "Sale total: " << pos->getSaleTotal() << endl;
} catch (ValidationError& err) {
    cout << err.getMessage() << endl;
}
```

Curs 6 Gestiunea memoriei in C++

- **Alocare dinamica. Destructor. RAll. Rule of three.**
- **Tratarea exceptiilor – exception safe code**
- **Moștenire**

Curs 5

- Template (Programare generica)
- STL – Standard Template Library
- Tratarea exceptiilor in C++

Alocare dinamică de obiecte

operatorul new alocă memorie pe heap și initializează obiectul (apelând constructorul clasei)

```
Rational *p1 = new Rational;  
Rational *p2 = new Rational{2, 5};  
cout << p1->toFloat() << endl;  
cout << (*p2).toFloat() << endl;  
delete p1;  
delete p2;
```

Orice variabilă creată cu **new** trebuie distrusă cu **delete**. (fiecare **new** exact un **delete**). Programatorul este responsabil cu eliberarea memoriei

Pentru a distruge vectori statici se folosește **delete []**

```
char* nume = new char[20];  
delete[] nume;  
  
//se apeleaza constructorul fara parametrii de 10 ori  
Pet* pets = new Pet[10];  
delete[] pets;
```

De preferat să nu se folosească malloc/free și **new/ delete** în același program, în special să distrugă cu free obiecte alocate cu **new**

new alocă memorie și apelează constructorul clasei

Destructor

O metoda specială a clasei.

Destructorul este apelat de fiecare data când se dealocă un obiect

- dacă am alocat pe heap (**new**), se apelează destructorul când apelez **delete/delete[]**
- dacă e variabilă alocată static, se dealoca în momentul în care nu mai e vizibil (out of scope)

```
DynamicArray::DynamicArray()
{
    cap = 10;
    elems = new Rational[cap];
    size = 0;
}
```

```
DynamicArray::~DynamicArray()
{
    delete[] elems;
}
```

Gestiunea memoriei in C++

Destructorul este apelat:

- Dacă obiectul a fost creat cu new → când apelam delete
- Dacă a fost alocat pe stack -> cand părăsește domeniul de vizibilitate (out of scope)
- Implementați destructor și eliberați memoria alocată cu new !!!!

Constructorul este apelat:

- când declarăm o variabilă pe stack
- Dacă creăm o variabilă cu new (pe heap)
- Dacă creăm o copie a obiectului (copy constructor/assignment operator)
 - atribuire
 - transmitere parametrii prin valoare
 - returnam obiect prin valoare dintr-o funcție
- implementați constructor de copiere și supraîncărcați operatorul = !!!

Cum gestionam memoria in C++ (RAII)

Orice memorie alocată pe heap (new) ar trebui încapsulată într-o clasa (in general ne referim la o astfel de clasă clasă Handler)

Ar trebui să alocăm memoria în constructor și să dealocăm memoria în destructor.
Astfel memoria alocată este strâns legată de ciclul de viață a obiectului Handler
Când cream obiectul se alocă memoria, când obiectul este distrus se eliberează memoria.

RAII (Resource Acquisition Is Initialization) este o tehnică prin care legăm ciclul de viață a unei resurse (memorie, thread, socket, fisier, mutex, conexiune la baza de date) de ciclul de viață a unui obiect.

Folosind RAII în programele C++:

gestiunea memoriei se simplifică – destructor / constructor / constructor de copiere sunt apelate automat

putem controla cât timp este memoria ocupată/ când este dealocată (are același viață ca și domeniul de vizibilitate a obiectului care încapsulează resursa)

Clase – încapsulare, abstractizare

clasele ajuta sa rationam asupra codului local, la un nivel de abstractizare mai mare

```
void someFunction(Pet p){  
    ...  
}
```

```
Pet someFunction(int a){  
    ...  
}
```

Parametrul p este transmis prin valoare

Daca Pet este un struct din C, trebuie sa inspectam definitia structului Pet pentru a decide dacă a transmite prin valoare este sau nu o abordare legitimă

Daca Pet este o clasa scrisa bine (rules of three) codul functioneaza corect (nu am nevoie sa stiu detalii despre clasa – lucrez la un nivel de abstractizare mai mare, clasa incapsuleaza/ascunde detaliiile de implementare

```
//vers 1  
typedef struct {  
    char name[20];  
    int age;  
} Pet;  
//a simple copy (bitwise) will do
```

```
//vers 2  
typedef struct {  
    char* name;  
    int age;  
} Pet;  
// when we make a copy we need to  
take care of the name field
```

Dacă am un struct C folosirea pointerilor nu rezolva dilema.

In acest caz nu trebuie sa ma gandesc la copiere dar trebuie sa ma gandesc la responsabilitatea dealocarii memoriei.(-> este nevoie sa ma uit la implementarea functiei **someFunction** nu pot sa ignore detaliile de implementare si sa rationez local asupra codului)

```
...  
Pet* p = someFunction(7);  
//do i need to delete/free p?  
//need to inspect someFunction
```

```
Pet* someFunction(int a){  
    ...  
}
```

Gestiunea memoriei alocate dinamic C vs C++

C	C++
<code>Pet p; // p este neinitializat</code>	<code>Pet p; // p este initializat, // se apelează constructorul</code>
Funcție pentru creare, distrugere Trebuie urmărit in cod folosirea corectă a acestor funcții (este ușor sa uiți să apelez)	Constructor / Destructor Sunt apelate automat de compilator
Funcție care copiază Trebuie urmărit/decis momentul in care dorim sa facem copie	Constructor de copiere, operator de assignment Sunt apelate automat de compilator
Pointeri peste tot in aplicație Este greu de decis cine este responsabil cu alocarea de-alocarea	Pointerii se încapsula într-o clasă handler Gestiunea memoriei este încapsulat într-o clasa (RAII). Ciclu de viată pentru memorie este strâns legat de ciclu de viată a obiectului Exista clase handler predefinite ex: unique_ptr
Se folosesc pointeri doar pentru a evita copierea Nu este clar dacă/când se dealocă	Tipul referință oferă o metodă mult mai transparentă pentru a evita copierea unde nu este necesar
Se compune greu Daca am o lista (care alocă dinamic) cu elemente alocate dinamic (poate chiar alta listă) este relativ greu de implementat logica de dealocare	Se compune ușor Daca am o clasa cu atribute, destructorul obiectului apelează si destructorul atributelor.
Gestiunea memoriei afectează toată aplicația.	Încapsulare/ascunderea detaliilor de implementare Modificările se fac doar in clasa

Rule of three

```
void testCopy() {
    DynamicArray ar1;
    ar1.addE(3);
    DynamicArray ar2 = ar1;
    ar2.addE(3);
    ar2.set(0, -1);
    printElems(&ar2);
    printElems(&ar1);
}
```

Daca o clasa este responsabila de gestiunea unei resurse (heap memory, fisier, etc) trebuie sa definim obligatoriu:

- copy constructor

```
DynamicArray::DynamicArray(const DynamicArray& d) {
    this->capacity = d.capacity;
    this->size = d.size;
    this->elems = new TElem[capacity];
    for (int i = 0; i < d.size; i++) {
        this->elems[i] = d.elems[i];
    }
}
```

- Assignment operator

```
DynamicArray& DynamicArray::operator=(const DynamicArray& ot) {
    if (this == &ot) {
        return *this;// protect against self-assignment (a = a)
    }
    delete[] this->elems;      //delete the allocated memory
    this->elems = new TElem[ot.capacity];
    for (int i = 0; i < ot.size; i++) {
        this->elems[i] = ot.elems[i];
    }
    this->capacity = ot.capacity;
    this->size = ot.size;
    return *this;
}
```

- Destructor

```
DynamicArray::~DynamicArray() {
    delete[] elems;
}
```

Review PetStore varianta OOP

Pentru o gestiune corecta a memoriei: Clasele care aloca memorie trebuie sa implementeze:

- constructor de copiere – copierea valorilor atributelor in noul obiect
- destructor – eliberare memorie
- operator = - eliberare memorie valori existente, copierea valorilor atributelor

Valabil pentru orice clasa care este responsabil de gestiunea unei resurse.

TAD VectorDinamic

- are operațiile listei – metodele publice din clasa – interfața clasei (.h)
- detaliile de implementare sunt ascunse – specificații abstracte, implementări in cpp
- elementele sunt obiecte nu pointeri – simplifica gestiunea memoriei

Clasele care nu aloca memorie pot folosi constructor/ destructor/ copy-constructor / assignment default (oferte implicit de compilator)

Daca dorim sa evitam anumite operații (ex. Din motive de performanta) putem folosi

```
PetController(const PetController& ot) = delete; //nu vreau sa se copieze controller
```

In afara de clasa Pet si clasa VectorDinamic nu e nevoie sa folosim pointeri si alocare dinamica.

Gestiunea memoriei se simplifica, se folosește RAII

obiectele au un ciclu de viață bine delimitat (domeniul de vizibilitate)
la creare se apelează constructor, la ieșire din domeniul de vizibilitate se apelează destructor, când folosim transmitere prin valoare sau return prin valoare se apelează copy constructor

Avantaje: creste nivelul de abstractizare: codul se simplifica, gestiunea memoriei se simplifica, folosirea clasei VectorDinamic nu necesita înțelegerea detaliilor de implementare

Dezavantajul abordării: Se fac multe copieri de obiecte

Masuri pentru a evita aceste copieri:

Transmitere prin referință, const correctness

Move Constructor, Move assignment (discutam la următoarele cursuri), Rule of five

Exception-safe code

Dezavantajele folosirii excepțiilor:

Dacă scriem cod care folosește excepții, ar trebui să luăm în considerare apariția unei excepții oriunde în cod.

Este greu să scrii cod care se comportă predictibil (fără buguri) chiar și atunci când apare o excepție. (nu rezultă probleme cu memorie - leak/dangling - sau cu alte resurse)

```
void g() {  
    ...  
}  
void f() {  
    char* s = new char[10];  
    ...  
    g(); //daca g arunca exceptie avem memory leak (nu se mai ajunge la  
delete s)  
    ...  
    delete[] s;  
}
```

Obs: astfel probleme sunt generale (chiar dacă nu folosim excepții) – putem avea un simplu return înainte de delete și rezulta memory leak

Ce înseamnă exception-safe code:

Basic: chiar dacă apare o excepție:

invariantii rămân valabili (obiectul nu ajunge într-o stare inconsistentă)
nu avem resource leak

Strong:

apariția unei excepții nu are nici un efect vizibil
operația ori se face în totalitate ori nu se modifică nimic (tranzacție)

Exception-safe code:

Pentru orice resursă pe care o gestionam (ex. Memorie) creăm o clasă

Orice pointer o să fie încapsulat într-un obiect, obiect automat - gestionat de compilator, declarat local în funcție (nu creat pe heap cu new) - No raw pointer

Ne folosim de RAI – beneficiem de faptul că destructorul se apelează când execuția părăsește scopul local (chiar dacă se iese aruncând o excepție)

Facem asta:

```
void f() {
    A a{ "asda",10 };
    ...
    g(); //daca g arunca excepție destructorul lui A se apelează
    ...
}
```

În loc de:

```
void f() {
    A* a = new A{ "asda",10 };
    ...
    g(); //daca g arunca excepție avem memory leak (nu se mai ajunge la delete s)
    ...
    delete a;
}
```

Unde chiar avem nevoie de pointeri și alocare pe heap putem folosi **unique_ptr**

```
#include <memory>
using std::unique_ptr;
void f() {
    unique_ptr<char[]> ptr_s = std::make_unique<char[]>( 10 );
    ...
    g(); //daca g arunca excepție destructorul lui A se apelează
    ...
    //când ieșim din funcție (excepție sau normal) destructorul lui ptr_s
    // apelează delete[] pentru char* de 10 caractere alocate pe heap
}

void f2() {
    auto ptr_s= make_unique<A>("asda",10);
    //când ieșim din funcție destructorul lui ptr_s apelează delete
    // pentru obiectul A create pe heap de metoda make_unique
}
```

unique_ptr – smart pointer

Clasa care conține un pointer, la apelul destructorului eliberează memoria ocupată de obiectul referit (face delete)

Util pentru a gestiona corect memoria, unique_ptr modelează „unique ownership”

```
int* f() {  
    ....  
}  
  
int main() {  
    int* pi = f();  
    //trebuie sa dealoc pi? cine este responsabil cu dealocarea?  
    ....  
    return 0;  
}  
  
#include <memory>  
std::unique_ptr<int> f() {  
    ....  
}  
int main() {  
    std::unique_ptr<int> pi = f();  
    //sunt responsabil cu dealocarea, se va dealoca automat cand pi  
    //iese din scope  
    ....  
    return 0;  
}
```

Regula: No raw pointer - No raw owning pointer

De evitat folosirea de pointeri (naked pointer) – folosirea excesiva de pointer face (aproape) imposibila gestiunea corecta a memoriei.

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasă de bază). Clasa nou creată moștenește comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele și variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adăuga noi membri (variabile, metode) pe lângă cele moștenite de la clasa A.

```
class Person {  
public:  
    Person(string cnp, string name);  
    const string& getName() const {  
        return name;  
    }  
  
    const string& getcNP() const {  
        return cnp;  
    }  
    string toString();  
protected:  
    string name;  
    string cnp;  
};
```

```
class Student: public Person {  
public:  
    Student(string cnp, string name,  
            string faculty);  
    const string& getFaculty() const {  
        return faculty;  
    }  
    string toString();  
private:  
    string faculty;  
};
```

Moștenire simplă. Clase derivate.

Dacă clasa B moștenește de la clasa A atunci:

- orice obiect de tip B are toate variabilele membre din clasa A
- funcțiile din clasa A pot fi aplicate și asupra obiectelor de tip B (daca vizibilitatea permite)
- clasa B poate adăuga variabile membre și sau metode pe lângă cele moștenite din A

```
class A:public B{  
...  
}
```

clasa B = Clasă de bază (superclass, base class, parent class)

clasa A = Clasă derivată (subclass, derived class, descendant class)

membrii (metode, variabile) moșteniți = membrii definiți în clasa A și nemodificați în clasa B

membrii redefiniți (overridden) = definit în A și în B (în B se crează o nouă definiție)

membrii adăugați = definiți doar în B

Vizibilitatea membrilor moșteniți

Dacă clasa A este derivat din clasa B:

- clasa A are acces la membri publici din B
- clasa A nu are acces la membrii privați din B

```
class A:public B{  
...  
}
```

public membrii publici din clasa B sunt publice și în clasa A

```
class A:private B{  
...  
}
```

private membrii publici din clasa B sunt private în clasa A

```
class A:protected B{  
...  
}
```

protected membrii publici din clasa B sunt protejate în clasa A (se vad doar în clasa A și în clase derivate din A).

Modificatori de acces

Definesc reguli de acces la variabile membre și metode dintr-o clasă

public: poate fi accesat de oriunde

private: poate fi accesat doar în interiorul clasei

protected: poate fi accesat în interiorul clasei și în clasele derivate.

protected se comportă ca și **private**, dar se permite accesul din clase derivate

Access	public	protected	private
clasa	Da	Da	Da
clasa derivată	Da	Da	Nu
În exterior	Da	Nu	Nu

Constructor/Destructor în clase derivate

- Constructorii și destructorii nu sunt moșteniți
- Constructorul din clasa derivată trebuie să apeleze constructorul din clasa de bază. Sa ne asigurăm ca obiectul este inițializat corect.
- Similar și pentru destructor. Trebuie să ne asigurăm ca resursele gestionate de clasa de bază sunt eliberate.

```
Student::Student(string cnp, string name, string faculty) :  
    Person(cnp, name) {  
    this->faculty = faculty;  
}
```

- Dacă nu apelăm explicit constructorul din clasa de bază, se apelează automat constructorul implicit
- Dacă nu există constructor implicit se generează o eroare la compilare

```
Student::Student(string cnp, string name, string faculty) {  
    this->faculty = faculty;  
}
```

Se apelează destructorul clasei de bază

```
Student::~Student() {  
    cout << "destroy student\n";  
}
```

Inițializare.

Când definim constructorul putem inițializa variabilele membre chiar înainte să se execute corpul constructorului.

```
Person::Person(string c, string n) :  
    cnp(c), name(n) {  
}
```

Initializare clasă de bază

```
Manager(std::string name, int yearInFirm, float payPerHour, float bonus) :  
    Employee(name, yearInFirm, payPerHour) {  
    this->bonus = bonus;  
}
```

Apel metodă din clasa de bază

```
float Manager::payment(int hoursWorked) {  
    float rez = Employee::payment(hoursWorked);  
    rez = rez + rez * bonus;  
    return rez;  
}
```

Creare /distrugere de obiecte (clase derivate)

Creare

- se alocă memorie suficientă pentru variabilele membre din clasa de bază
- se alocă memorie pentru variabile membre noi din clasa derivată
- se apelează constructorul clasei de bază pentru a inițializa atributele din clasa de bază
- se execută constructorul din clasa derivată

Distrugere

- se apelează destructorul din clasa derivată
- se apelează destructorul din clasa de bază

Principiul substituției.

Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază. (upcast implicit!)

```
Person p = Person("1", "Ion");
cout << p.toString() << "\n";

Student s("2", "Ion2", "Info");
cout << s.toString() << "\n";

Teacher t("3", "Ion3", "Assist");
cout << t.getName() << " " << t.getPosition() << "\n";

p = s;
cout << p.getName() << "\n";

p = t;
cout << p.getName() << "\n";

s = p;//not valid, compiler error
```

Pointer

```
Person *p1 = new Person("1", "Ion");
cout << p1->getName() << "\n";

Person *p2 = new Student("2", "Ion2", "Mat");
cout << p2->getName() << "\n";

Teacher *t1 = new Teacher("3", "Ion3", "Lect");
cout << t1->getName() << "\n";

p1 = t1;
cout << p1->getName() << "\n";

t1 = p1;//not valid, compiler error
```

Diagramme UML (Is a vs Has a)



- Un Sale are una sau mai multe SaleItem
- Un SaleItem are un Product

Relația de asociere UML (Associations): Descriu o relație de dependență structurală între clase

Elemente posibile:

- nume
- multiplicitate
- nume rol
- uni sau bidirectional

***Tipuri de relații de asociere**

- Asociere
- Agregare (compoziție) (whole-part relation)
- Dependență
- Moștenire

Are (has a):

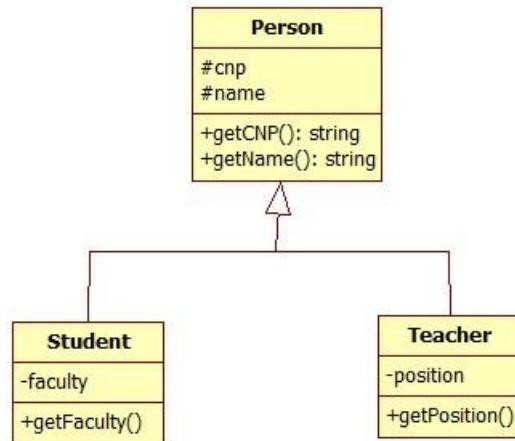
- Orice obiect de tip A are un obiect B.
- SaleItem are un Product. Persoana are nume (string)
- în cod apare ca și o variabilă membră

Este ca și (is a ,is like a):

- Orice instanță de tip A este și de tip B
- Orice student este o persoană
- se implementează folosind moștenirea

Relația de specializare/generalizare – Reprezentarea UML .

Folosind moștenirea putem defini ierarhii de clase



Studentul este o Persoană cu câteva atribute adiționale

Studentul moștenește (variabile și metode) de la Persoană

Student este derivat din Persoană. Persoana este clasă de bază,

Student este clasa derivată

Persoana este o generalizare a Studentului

Student este o specializare a Persoanei

Curs 7 Moștenire, Polimorfism

- Moștenire
- Polimorfism – Metode pur virtuale, Clase abstracte
- Operații de intrări ieșiri în C++
 - Fișiere

Curs 6 Gestiunea memoriei in C++

- Alocare dinamica. Destructor. RAII. Rule of three.
- Tratarea exceptiilor – exception safe code
- Moștenire

De ce moștenire, polimorfism:

- 1. Unele concepte din lumea reală au o natură ierarhică, aceste lucruri de multe ori se pot modela în program cel mai ușor creând ierarhii de clase folosind moștenire.**

Ex. clasele din QT care modeleză elemente de interfață grafică utilizator

- 2. Dorim să adăugăm o nouă clasă în aplicație și avem deja o clasă existentă care are funcționalități comune cu ce dorim noi să adăugăm.**

O opțiune este să folosim moștenire pentru a reutiliza partea de funcționalitate deja scrisă în clasa existentă și să adăuga doar partea care diferă în clasa fiu nou creată. Încercăm să reprezentăm relația „is a”, ex: FileRepository este („is a”) Repository.

- 3. Vrem să scriem o parte a aplicației astfel încât să fie ușor de extins ulterior cu noi funcționalități. Vrem să tratăm un set de obiecte de tipuri diferite în același fel, să grupăm clase diferite.**

O să scriem cod care funcționează doar cu interfață (metodele expuse) de la clasa de bază. Ulterior putem să adăugăm clase noi care moștenesc din clasa de bază și codul deja scris o să funcționeze și cu aceste noi clase fără a rescriem nimic din ceea ce există înainte.

Obs. De multe ori moștenirea/polimorfismul nu este singura soluție. În funcție de situație, capabilitățile limbajului de programare și natura aplicației/problemei alte alternative sunt posibile și căte-o dată preferate.

Mecanisme/concepte care pot constitui alternative de luat în considerare:

Compoziție: În loc să moștenim dintr-o clasă putem folosi clasa (un câmp în clasa nouă) și să refolosim astfel ceea ce există deja scris.

Higher order functions (Funcții care primesc ca parametru alte funcții): pot fi o soluție pentru a crea cod generic, pentru a elimina duplicare de cod, pentru a oferi un punct de extensie în aplicație.

Duck typing : ajută în scrierea de cod general, care funcționează cu un set de obiecte de tipuri diferite

Template: Permite scrierea de metode/clase generice, similar cu duck typing în acest context.

Mixins, Aspects: Mecanisme existente în unele limbaje

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasă de bază). Clasa nou creată moștenește comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele și variabilele membre din clasa A
- clasa B poate redifini metode din clasa A
- clasa B poate adăuga noi membri (variabile, metode) pe lângă cele moștenite de la clasa A.

```
class Person {
public:
    Person(string cnp, string name);
    const string& getName() const
    {
        return name;
    }
    const string& getCNP() const
    {
        return cnp;
    }
    string toString();
protected:
    string name;
    string cnp;
};
```

```
class Student: public Person {
public:
    Student(string cnp, string name,
            string faculty);
    const string& getFaculty() const {
        return faculty;
    }
    string toString();
private:
    string faculty;
};
```

Moștenire simplă. Clase derivate.

Dacă clasa B moștenește de la clasa A atunci:

- orice obiect de tip B are toate variabilele mebre din clasa A
- funcțiile din clasa A pot fi aplicate și asupra obiectelor de tip B (daca vizibilitatea permite)
- clasa B poate adăuga variabile membre și sau metode pe lângă cele moștenite din A

```
class A:public B{  
...  
}
```

clasa B = Clasă de bază (superclass, base class, parent class)

clasa A = Clasă derivată (subclass, derived class, descendant class)

membrii (metode, variabile) moșteniți = membrii definiți în clasa A și nemodificați în clasa B

membrii redefiniți (overridden) = definit în A și în B (în B se crează o nouă definiție)

membrii adăugați = definiți doar în B

Vizibilitatea membrilor moșteniți

Dacă clasa A este derivat din clasa B:

- clasa A are acces la membri publici din B
- clasa A nu are acces la membrii privați din B

```
class A:public B{  
...  
}
```

public membrii publici din clasa B sunt publice și în clasa A

```
class A:private B{  
...  
}
```

private membrii publici din clasa B sunt private în clasa A

```
class A:protected B{  
...  
}
```

protected membrii publici din clasa B sunt protejate în clasa A (se vad doar în clasa A și în clase derivate din A).

Modifieri de acces

Definesc reguli de acces la variabile membre și metode dintr-o clasă

public: poate fi accesat de oriunde

private: poate fi accesat doar în interiorul clasei

protected: poate fi accesat în interiorul clasei și în clasele derivate.

protected se comportă ca și **private**, dar se permite accesul din clase derivate

Access	public	protected	private
clasa	Da	Da	Da
clasa derivată	Da	Da	Nu
În exterior	Da	Nu	Nu

Constructor/Destructor în clase derivate

- Constructorii și destructorii nu sunt moșteniți
- Constructorul din clasa derivată trebuie să apeleze constructorul din clasa de bază. Sa ne asigurăm ca obiectul este inițializat corect.
- Similar și pentru destructor. Trebuie să ne asigurăm ca resursele gestionate de clasa de bază sunt eliberate.

```
Student::Student(string cnp, string name, string faculty) :  
    Person(cnp, name) {  
    this->faculty = faculty;  
}
```

- Dacă nu apelăm explicit constructorul din clasa de bază, se apelează automat constructorul implicit
- Dacă nu există constructor implicit se generează o eroare la compilare

```
Student::Student(string cnp, string name, string faculty) {  
    this->faculty = faculty;  
}
```

Se apelează destructorul clasei de bază

```
Student::~Student() {  
    cout << "destroy student\n";  
}
```

Inițializare.

Când definim constructorul putem inițializa variabilele membre chiar înainte sa se execute corpul constructorului.

```
Person::Person(string c, string n) :  
    cnp(c), name(n) {  
}
```

Initializare clasă de bază

```
Manager(std::string name, int yearInFirm, float payPerHour, float  
bonus) :  
    Employee(name, yearInFirm, payPerHour) {  
        this->bonus = bonus;  
    }
```

Apel metodă din clasa de bază

```
float Manager::payment(int hoursWorked) {  
    float rez = Employee::payment(hoursWorked);  
    rez = rez + rez * bonus;  
    return rez;  
}
```

Creare /distrugere de obiecte (clase derivate)

Creare

- se alocă memorie suficientă pentru variabilele membre din clasa de bază
- se alocă memorie pentru variabile membre noi din clasa derivată
- se apelează constructorul clasei de bază pentru a inițializa atributele din clasa de bază
- se execută constructorul din clasa derivată

Distrugere

- se apelează destructorul din clasa derivată
- se apelează destructorul din clasa de bază

Principiul substituției.

Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază. (upcast implicit!)

```
Person p = Person("1", "Ion");
cout << p.toString() << "\n";

Student s("2", "Ion2", "Info");
cout << s.toString() << "\n";

Teacher t("3", "Ion3", "Assist");
cout << t.getName() << " " << t.getPosition() << "\n";

p = s;
cout << p.getName() << "\n";

p = t;
cout << p.getName() << "\n";

s = p;//not valid, compiler error
```

Pointer

```
Person *p1 = new Person("1", "Ion");
cout << p1->getName() << "\n";

Person *p2 = new Student("2", "Ion2", "Mat");
cout << p2->getName() << "\n";

Teacher *t1 = new Teacher("3", "Ion3", "Lect");
cout << t1->getName() << "\n";

p1 = t1;
cout << p1->getName() << "\n";

t1 = p1;//not valid, compiler error
```

Diagrame UML (Is a vs Has a)



- Un Sale are una sau mai multe SaleItem
- Un SaleItem are un Product

Relația de asociere UML (Associations): Descriu o relație de dependență structurală între clase

Elemente posibile:

- nume
- multiplicitate
- nume rol
- uni sau bidirectional

Tipuri de relații de asociere

- Asociere
- Agregare (compoziție) (whole-part relation)
- Dependență
- Moștenire

Are (has a):

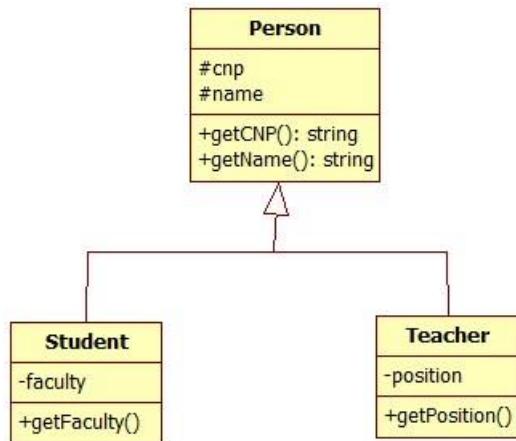
- Orice obiect de tip A are un obiect B.
- SaleItem are un Product. Persoana are nume (string)
- în cod apare ca și o variabilă membră

Este ca și (is a ,is like a):

- Orice instanță de tip A este și de tip B
- Orice student este o persoană
- se implementează folosind moștenirea

Relația de specializare/generalizare – Reprezentarea UML .

Folosind moștenirea putem defini ierarhii de clase



Studentul este o Persoană cu câteva atribute adiționale

Studentul moștenește (variabile și metode) de la Persoană

Student este derivat din Persoană. Persoana este clasă de bază, Student este clasa derivată

Persoana este o generalizare a Studentului

Student este o specializare a Persoanei

Suprascriere (redefinire) de metode.

Clasa derivată poate redefini metode din clasa de bază

<pre>string Person::toString() { return "Person:" + cnp + " " + name; }</pre>	<pre>string Student::toString() { return "Student:" + cnp + " " + name + " " + faculty; }</pre>
<pre>Person p = Person("1", "Ion"); cout << p.toString() << "\n";</pre>	<pre>Student s("2", "Ion2", "Info"); cout << s.toString() << "\n";</pre>

În clasa derivată descriem ce este specific clasei derivate, ce diferă față de clasa de bază

Suprascriere (overwrite) \neq Supraîncărcare (overload)

```
string Person::toString() {
    return "Person:" + cnp + " "
+ name;
}

string Person::toString(string prefix) {
    return prefix + cnp + " "
+ name;
}
```

toString este o metodă supraîncărcată(**toString()**, **toString(string prefix)**)

Polimorfism

Proprietatea unor entități de:

- a se comporta diferit în funcție de tipul lor
- a reacționa diferit la același mesaj

Obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj (apel de metodă).

Proprietate a unui limbaj OO de a permite manipularea unor obiecte diferite prin intermediul unei interfețe comune

Tipul declarat vs tipul actual

Orice variabilă are un tip declarat (la declararea variabilei se specifică tipul). În timpul execuției valoarea referită de variabila are un tip actual care poate差别 de tipul declarat

```
Student s("2", "Ion2", "Info");
Teacher t("3", "Ion3", "Assist");
Person p = Person("1", "Ion");

cout << p.toString() << "\n";

p = s; //slicing
cout << p.toString() << "\n";

p = t; //slicing
cout << p.toString() << "\n";
```

Tipul declarat pentru p este Persoană, dar în timpul execuției p are valori de tip Person, Student și Teacher.

Object slicing

Dacă asignăm un obiect de tipul clasei derivate la o variabilă de tipul clasei de bază, obiectul pierde partea adăugată de clasa derivată.

Ex. p=s unde p e de tip Person și s e de tip Student: în acest caz p va să conțină doar atributelor din clasa Person.

Slicing se întâmplă la orice copiere de acest fel: assignment, transmitere prin valoare, return prin valoare.

Pentru a evita slicing se pot folosi referințe (&) sau pointeri (*).

```

string Person::toString() {
    return "Person:" + cnp + " " + name;
}
string Student::toString() {
    return "Student:" + cnp + " " + name + " " + faculty;
}
string Teacher::toString() {
    string rez = Person::toString();
    return "Teacher " + rez;
}
Student s("2", "Ion2", "Info");
Person* aux = &s;
cout << aux->toString() << "\n";
Person p = Person("1", "Ion");
aux = &p;
cout << aux->toString() << "\n";

```

- Person, Student, Teacher are metoda `toString`, fiecare clasă definește propria versiune de `toString`.
- Sistemul trebuie să determine dinamic care dintre variante trebuie executată în momentul în care metoda `toString` este apelată.
- Decizia trebuie luată pe baza tipului actual al obiectului.
- Funcționalitate importantă (prezent în limbaje OO) numită legare dinamică - dynamic binding (late binding, runtime binding).

Legare dinamică (Dynamic binding).

Legarea (identificarea) codului de executat pe baza numelui de metode se poate face:

- în timpul compilării => legare statică (static binding)
- în timpul execuției => legare dinamică (dynamic binding)

Legare dinamică:

- selectarea metodei de executat se face timpul execuției.
- Când se apelează o metodă, codul efectiv executat (corpul funcției) se alege la momentul execuției (la legare statică decizia se ia la compilare)
- legarea dinamica în C++ funcționează doar pentru referințe și pointeri
- În C++ doar metodele virtuale folosesc legarea dinamică

Metode virtuale.

Legarea dinamică în c++: Folosind metode virtuale

O metodă este declarată virtual în casa de bază:

virtual <function-signature>

- metoda suprascrisă în clasele derivate are legarea dinamică activată
 - metoda apelată se va decide în funcție de tipul actual al obiectului (nu în funcție de tipul declarat).
- Constructorul nu poate fi virtual – pentru a crea un obiect trebuie să știm tipul exact
 - Destructorul poate fi virtual (este chiar recomandat să fie când avem ierarhii de clase)

```
class Person {  
protected:  
    string name;  
    string cnp;  
  
public:  
    Person(string cnp, string name);  
    virtual ~Person();  
  
    const string& getName() const {  
        return name;  
    }  
  
    const string& getcNP() const {  
        return cnp;  
    }  
    virtual string toString();  
    string toString(string prefix);  
};
```

Mecanism C++ pentru polimorfism

Orice obiect are atașat informații legate de metodele obiectului

Pe baza acestor informații apelul de metodă este efectuat folosind implementarea corectă (cel din tipul actual). Orice obiect are referință la un tabel prin care pentru metodele virtuale se selectează implementarea corectă.

Orice clasă care are cel puțin o metodă virtuală (clasă polimorfică) are un tabel numit VTABLE (virtual table). VTABLE conține adrese la metode virtuale ale clasei.

Când invocăm o metodă folosind un pointer sau o referință compilatorul generează un mic cod adițional care în timpul execuției o să folosească informația din VTABLE pentru a selecta metoda de executat.

Destructor virtual

- Destructorul este responsabil cu dealocarea resurselor folosite de un obiect
- Dacă avem o ierarhie de clasă atunci este de dorit să avem un comportament polimorfic pentru destructor (să se apeleze destructorul conform tipului actual)
- Trebuie să declarăm destructorul ca fiind virtual

Moștenire multiplă

În C++ este posibil ca o clasă să aibă multiple clase de bază, să moșteneasca de la mai multe clase

```
class Car : public Vehicle , public InsuredItem {  
};
```

Clasa moștenește din toate clasele de bază toate atributele.

Moștenirea multipla poate fi periculoasă și în general ar trebui evitat

- se poate moșteni același atribut de la diferite clase
- putem avea clase de bază care au o clasă de bază comună

Functii pur virtuale

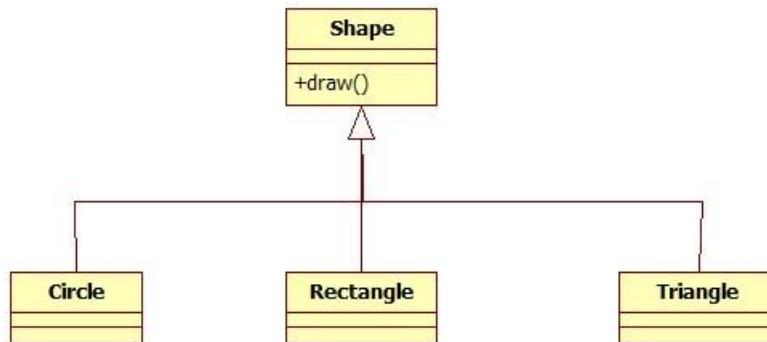
Functiile pur virtuale nu sunt definite (avem doar declaratia metodei).

Folosim metode pur virtuale pentru a ne asigura ca toate clasele deriveate (concrete) o sa defineasca metoda.

```
class Shape {  
public:  
    Shape();  
    virtual ~Shape();  
    virtual void draw() = 0; //pure virtual  
};
```

=0 indica faptul ca nu exista implementare pentru aceasta metoda in clasa.
Clasele care au metode pur virtuale nu se pot instanta.

Shape este o clasa abstracta - defineste doar interfața, dar nu conține implementări.



Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;

Oferă:

- o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)
- pot conține atrbute comune tuturor claselor derivate

o clasă abstractă nu are instanțe

o clasă abstractă are cel puțin o metodă pur virtuală:

virtual <return-type> <name> (<parameters>) = 0;

clăsă pur abstractă = clăsă care are doar metode pur virtuale

clăsă pur abstractă = interfață

În UML font italic

Clase care extind clase abstracte

- O clasă derivată dintr-o clasă abstractă moștenește interfața publică a clasei abstracte
- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstractă
- putem avea instanțe

Functii pur virtuale

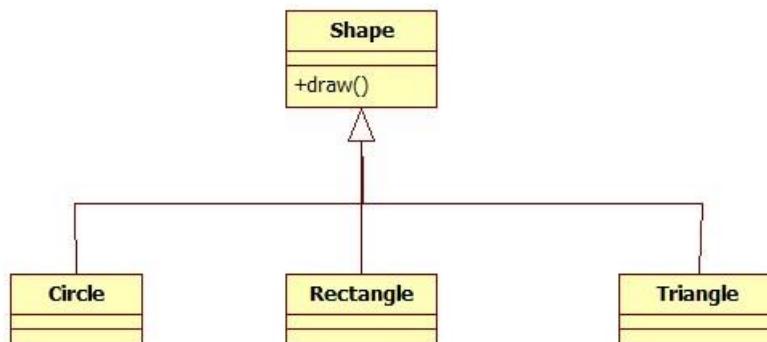
Functiile pur virtuale nu sunt definite (avem doar declaratia metodei).

Folosim metode pur virtuale pentru a ne asigura ca toate clasele deriveate (concrete) o sa defineasca metoda.

```
class Shape {  
public:  
    Shape();  
    virtual ~Shape();  
    virtual void draw() = 0; //pure virtual  
};
```

=0 indica faptul ca nu exista implementare pentru aceasta metoda in clasa. Clasele care au metode pur virtuale nu se pot instanta.

Shape este o clasa abstracta - defineste doar interfața, dar nu contine implementari.



Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;

Oferă:

- o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)
- pot conține atrbute comune tuturor claselor derivate

o clasă abstractă nu are instanțe

o clasă abstractă are cel puțin o metodă pur virtuală:

virtual <return-type> <name> (<parameters>) = 0;

clasă pur abstractă = clasă care are doar metode pur virtuale

clasă pur abstractă = interfață

În UML font italic

Clase care extind clase abstracte

- O clasă derivată dintr-o clasă abstractă moștenește interfața publică a clasei abstracte
- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstractă
- putem avea instanțe

Moștenire. Polimorfism

De ce folosim moștenire:

Moștenire de implementare (Implementation Inheritance):

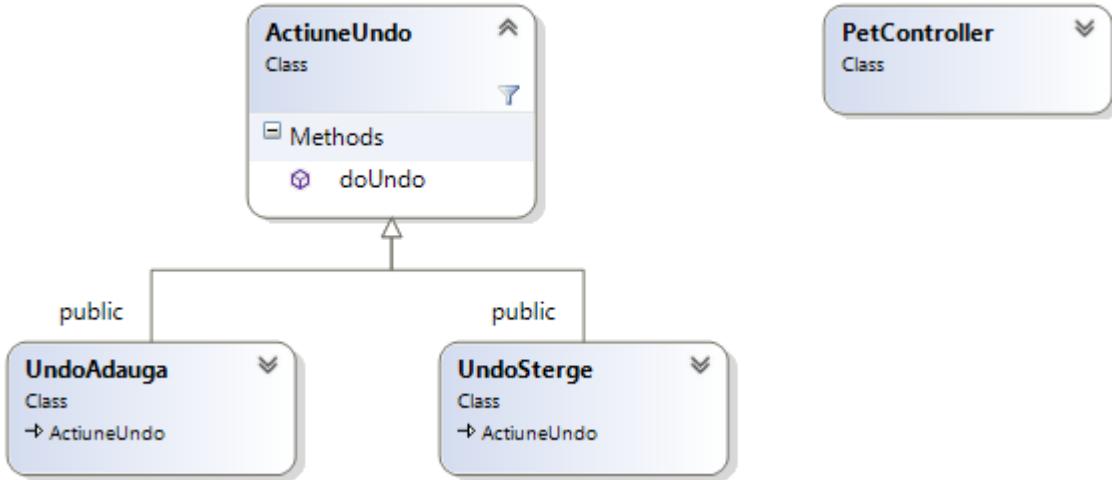
Clasa de baza oferă funcționalitate (metode, câmpuri) ce ușurează implementarea clasei derivate. Folosim moștenire pentru a reutiliza codul din clasa de baza (Ex. MemoryRepository -> FileRepository)

Moștenire de interfață (Interface Inheritance):

Obiectul de clasa derivată poate fi folosit oriunde se cere ceva de tipul clasei de baza (cele două expun același interfață). Folosim pentru a oferi un punct de extensie (adăugare de funcționalitate nouă fără a modifica codul existent). Ex: Undo

- **reutilizare de cod**
 - **clasa derivată moștenește din clasa de bază**
 - **se evită copy/paste – mai ușor de întreținut, înțeles**
- **extensibilitate**
 - **permite adăugarea cu ușurință de noi funcționalități**
 - **extindem aplicația fără să modificăm codul existent**

Exemplu : Undo



```
class ActiuneUndo {
public:
    virtual void doUndo() = 0;
    //destructorul e virtual pentru a ne asigura ca daca dau delete pe un
    //pointer se apeleaza destructorul din clasa care trebuie
    virtual ~ActiuneUndo(){};

};

class UndoAdauga : public ActiuneUndo {
    Pet petAdaugat;
    PetRepo& rep;
public:
    UndoAdauga(PetRepo& rep, const Pet& p) :rep{ rep }, petAdaugat{ p } {}
    void doUndo() override {
        rep.sterge(petAdaugat);
    }
};

void PetController::add(const char* type, const char* species, int price) {
    Pet p{ type,species,price };
    repo.store(p);
    undoActions.push_back(new UndoAdauga{repo, p});
}

void PetController::undo() {
    if (undoActions.empty()) {
        throw PetException{"Nu mai exista operatii"};
    }
    ActiuneUndo* act = undoActions.back();
    act->doUndo(); //nu e exception safe - vezi varianta cu unique_ptr
    undoActions.pop_back();
    delete act;
}
```

Operații de intrare/ieșire

IO (Input/Output) în C

<stdio.h> -> **scanf()**, **printf()**, **getchar()**, **getc()**, **putc()**, **open()**, **close()**, **fgetc()**, etc.

- Funțiile din C nu sunt extensibile
- funcționează doar cu un set limitat de tipuri de date (**char**, **int**, **float**, **double**).
- Nu fac parte din librăria standard => Implementările pot差别 (ANSI standard)
- pentru fiecare clasă nouă, ar trebui să adăugăm o versiune nouă (supraîncărcare) de funcții **printf()** și **scanf()** și variantele pentru lucru cu fișiere, siruri de caractere
- Metodele supraîncărcate au același nume dar o listă de parametrii diferit. Metodele **printf** și variantele pentru string, fișier folosesc o listă de argumente variabilă – nu putem supraîncărca.

Biblioteca de intrare/ieșire din C++ a fost creat să:

- fie ușor de extins
- ușor de adăugat/folosit tipuri noi de date

I/O streams. I/O Hierarchies of classes.

Iostream este o bibliotecă folosit pentru operații de intrări ieșiri in C++.

Este orientat-obiect și oferă operații de intrări/ieșiri bazat pe noțiunea de flux (stream)

iostream este parte din C++ Standard Library și conține un set de clase template și funcții utile in C++ pentru operații IO

Biblioteca standard de intrări/ieșiri (iostream) conține:

Clase template

O ierarhie de clase template, implementate astfel încât se pot folosi cu orice tip de date.

Instanțe de clase template

Biblioteca oferă instanțe ale claselor template speciale pentru manipulare de caractere **char** (narrow-oriented) respectiv pentru elemente de tip wchar (wide-oriented).

Obiecte standard

În fișierul header <iostream> sunt declarate obiecte care pot fi folosite pentru operații cu intrare/ieșire standard.

Tipuri

conține tipuri noi, folosite biblioteca standard, cum ar fi: streampos, streamoff and streamsiz (reprezintă poziții, offset, dimensiuni)

Manipulatori

Funții globale care modifică proprietăți generale, oferă informații de formatare pentru streamuri.

Streamuri standard – definite în <iostream>

cin - corespunde intrării standard (stdin), este de tip **istream**

cout - corespunde ieșirii standard (stdout) , este de tip **ostream**

cerr - corespunde ieșirii standard de erori (stderr), este de tip **ostream**

```
#include <iostream>
using namespace std;

void testStandardIOStreams() {
    cout << "!!!Hello World!!!" << endl;
    // prints !!!Hello World!!! to the console
    int i = 0;
    cin >> i; //read an int from the console
    cout << "i=" << i << endl; // prints !!!Hello World!!! to the console

    cerr << "Error message";//write a message to the standard error stream
}
```

Operatorul de inserție (Insertion operator - output)

- Pentru operațiile de scriere pe un stream (ieșire standard, fișier, etc) de folosește operatorul “<<”, numit operator de inserție
- pe partea stângă trebuie să avem un obiect de tip ostream (sau derivat din ostream). Pentru a scrie pe ieșire standard (consolă) se folosește **cout** (declarat în modulul iostream)
- pe dreapta putem avea o expresie.
- Operatorul este supraîncărcat pentru tipurile standard, pentru tipurile noi programatorul trebuie să supraîncarce.

```
void testWriteToStandardStream() {  
    cout << 1 << endl;  
    cout << 1.4 << endl << 12 << endl;  
    cout << "asdasd" << endl;  
    string a("aaaaaaaa");  
    cout << a << endl;  
  
    int ints[10] = { 0 };  
    cout << ints << endl; //print the memory address  
}
```

- Se pot înlăntui operații de inserție, evaluarea se face în ordinea inversă a scrierii. Înlăntuirea funcționează fiindcă operatorul << returnează o referință la stream

Operatorul de extragere – citire (Extraction operator)

- Citirea dintr-un stream se realizează folosind operatorul “>>“
- operandul din stânga trebuie să fie un obiect de tip istream (sau derivat din istream). Pentru a citi din intrarea standard (consolă) putem folosi cin, obiect declarat în iostream
- operandul de pe dreapta poate fi o expresie, pentru tipuri standard operatorul de extragere este supraîncărcat.
- programatorul poate supraîncărca operatorul pentru tipuri noi.

```
void testStandardInput() {  
    int i = 0;  
    cout << "Enter int:";  
    cin >> i;  
    cout << i << endl;  
    double d = 0;  
    cout << "Enter double:";  
    cin >> d;  
    cout << d << endl;  
    string s;  
    cin >> s;  
    cout << s << endl;  
}
```

Fișiere

Pentru a folosi fișiere în aplicații C++ trebuie să conectăm streamul la un fișier de pe disk

fstream oferă metode pentru citire/scriere date din/in fișiere.

<fstream.h>

- ifstream (input file stream)
- ofstream (output file stream)

Putem ataşa fişierul de stream folosind constructorul sau metoda open

După ce am terminat operațiile de IO pe fișier trebuie sa închidem (dezasociem) streamul de fișier folosind metoda close. Ulterior, folosind metoda **open**, putem folosi streamul pentru a lucra cu un alt fișier.

Metoda is_open se poate folosi pentru a verifica dacă streamul este asociat cu un fișier.

Output File Stream

```
#include <fstream>

void testOutputToFile() {
    ofstream out("test.out");
    out << "asdasdasd" << endl;
    out << "kkkkkkk" << endl;
    out << 7 << endl;
    out.close();
}
```

- Dacă fișierul “test.out” există pe disc, se deschide fișierul pentru scriere și se conectează streamul la fișier. Continutul fișierului este șters la deschidere.
- Dacă nu există “test.out”: se creează, se deschide fișierul pentru scriere și se conectează streamul la fișier.

Input File Stream

```
void testInputFromFile() {  
    ifstream in("test.out");  
    //verify if the stream is opened  
    if (in.fail()) {  
        return;  
    }  
    while (!in.eof()) {  
        string s;  
        in >> s;  
        cout << s << endl;  
    }  
    in.close();  
}
```

```
void testInputFromFileByLine() {  
    ifstream in;  
    in.open("test.out");  
    //verify if the stream is opened  
    if (!in.is_open()) {  
        return;  
    }  
    while (in.good()) {  
        string s;  
        getline(in, s);  
        cout << s << endl;  
    }  
    in.close();  
}
```

- Dacă fișierul “test.out” există pe disc, se deschide pentru citire și se conectează streamul la fișier.
- Dacă nu există fișierul streamul nu se asociază, nu se poate citi din stream
- Unele implementări de C++ creează fișier dacă acesta nu există.

Open

Funcția open deschide fișierul și asociază cu stream-ul:

open (filename, mode);

filename sir de caractere ce indică fișierul care se deschide
mode parametru optional, indică modul în care se deschide fișierul. Poate fi o combinație dintre următoarele flaguri:

ios::in Deschide pentru citire.

ios::out Deschide pentru scriere.

ios::binar Mod binar.
y

ios::ate Se poziționează la sfârșitul fișierului.

Dacă nu e setat, după deschidere se poziționează la început.

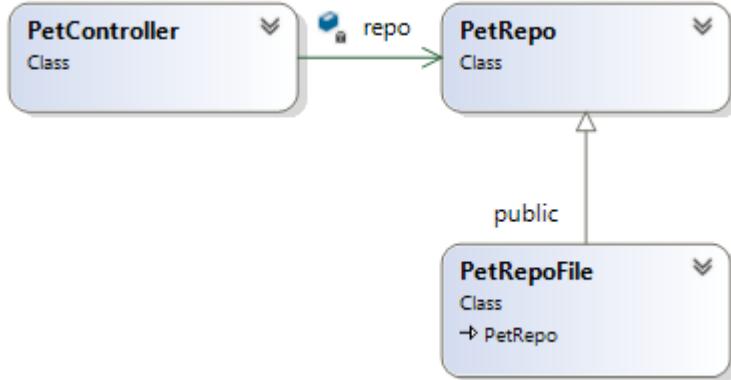
Toate operațiile de scriere se efectuează la sfârșitul fișierului, se adaugă

ios::app la conținutul existent. Poate fi folosit doar pe stream-uri deschise pentru scriere.

ios::trunc Sterge conținutul existent.

Flag-urile se pot combina folosind operatorul pe biți OR ()).

Salvare date in fisier - FileRepository



```

class PetRepoFile :public PetRepo {
private:
    std::string fName;
    void loadFromFile();
    void writeToFile();
public:
    PetRepoFile(std::string fName) :PetRepo(), fName{ fName } {
        loadFromFile(); //incarcam datele din fisier
    }
    void store(const Pet& p) override {
        PetRepo::store(p); //apelam metoda din clasa de baza
        writeToFile();
    }
    void sterge(const Pet& p) override {
        PetRepo::sterge(p); //apelam metoda din clasa de baza
        writeToFile();
    }
};

#include <iostream>
void PetRepoFile::loadFromFile(){
    std::ifstream in(fName);
    if (!in.is_open()) {
        //verify if the stream is opened
        throw PetException("Error open file");
    }
    while (!in.eof()) {
        std::string species;
        in >> species;
        //poate am linii goale
        if (in.eof()) break;
        std::string type;
        in >> type;
        int price;
        in >> price;
    }
    Pet p{type.c_str(),species.c_str(), price};
    PetRepo::store(p);
}
in.close();
}

void PetRepoFile::writeToFile() {
    std::ofstream out(fName);
    if (!out.is_open()) {
        //verify if the stream is opened
        std::string msg("Error open file");
        throw PetException(msg);
    }
    for (auto& p:getAll()) {
        out << p.getSpecies();
        out << std::endl;
        out << p.getType();
        out << std::endl;
        out << p.getPrice();
        out << std::endl;
    }
    out.close();
}
  
```

Erori la citire scriere - Flag-uri

Indică starea internă a unui stream:

Flag	Descriere	Metodă
fail	Date invalide	fail()
badbit	Eroare fizică	bad()
goodbit	OK	good()
eofbit	Sfârșid de stream detectat	eof()

```
void testFlags(){
    cin.setstate(ios::badbit);
    if (cin.bad()){
        //something wrong
    }
}
```

Flag de control :

```
cin.setf(ios::skipws); //Skip white space. (For input; this is the
default.)
cin.unsetf(ios::skipws);
```

```
/*
    Citeste date de la consola (int,float, double, string, etc)
    Reia citirea pana cand utilizatorul introduce corect
*/
template<typename T>
T myread(const char* msg) {
    T cmd;
    while (true) {
        std::cout<<std::endl << msg;
        std::cin >> cmd;
        bool fail = std::cin.fail();
        std::cin.clear(); //resetez failbit
        auto& aux = std::cin.ignore(1000, '\n');
        if (!fail && aux.gcount()<=1) {
            break; //am reusit sa citesc numar
        }
    }
    return cmd;
}
```

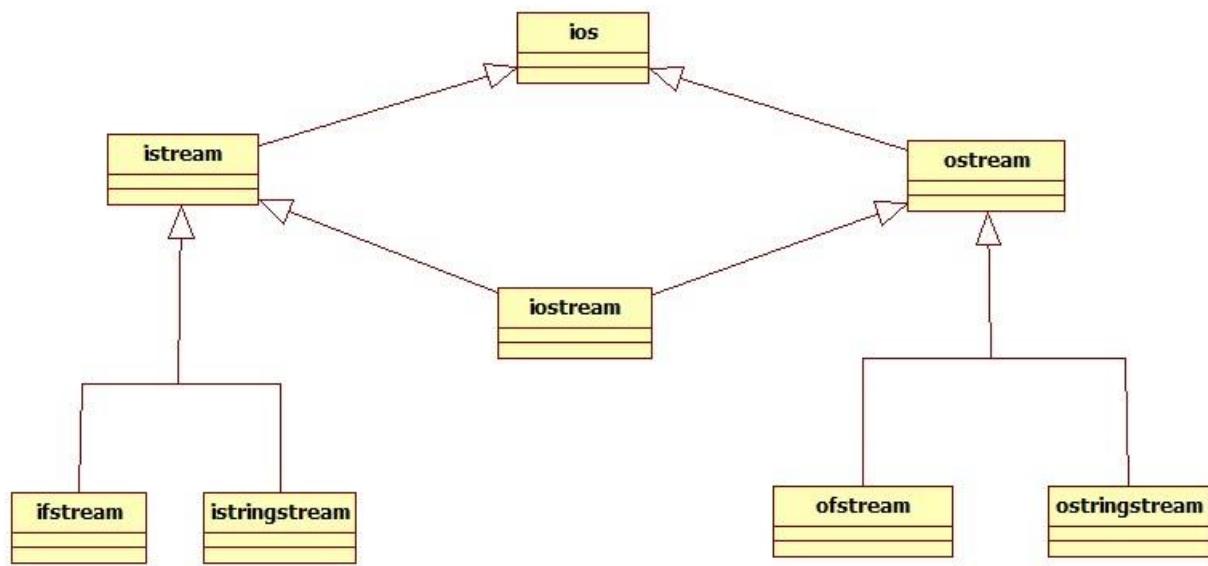
Formatare scriere

width(int x)	Numărul minim de caractere pentru scrierea următoare
fill(char x)	Caracter folosit pentru a umple spațiu dacă e nevoie să completeze cu caractere (lungime mai mică decât cel setat folosind width).
precision(int x)	Numărul de zecimale scrise

```
void testFormatOutput() {
    cout.width(5);
    cout << "a";
    cout.width(5);
    cout << "bb" << endl;
    const double PI = 3.1415926535897;
    cout.precision(3);
    cout << PI << endl;
    cout.precision(8);
    cout << PI << endl;
}

/*
Tipareste lista de pet
*/
void PetUI::printPets(const std::vector<Pet>& v) {
    std::cout << "\n Pets("<<v.size()<<"):\n";
    printTableHeader();
    for (const Pet& p : v) {
        std::cout.width(10);
        std::cout << p.getType();
        std::cout.width(20);
        std::cout << p.getSpecies();
        std::cout.width(10);
        std::cout << p.getType();
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
void printTableHeader() {
    std::cout.width(10);
    std::cout << "Type";
    std::cout.width(20);
    std::cout << "Species";
    std::cout.width(10);
    std::cout << "Price";
    std::cout << std::endl;
}
```

Hierarhie de clase din biblioteca standard IO C++



Clasele folosite pentru intrări/ieșiri sunt definite în fișiere header:

- <iostream> formatare , streambuffer.
- <iostream> intrări formatare
- <ostream> ieșiri formatare
- <iostream> implementează intrări/ieșiri formatare
- <fstream> intrări/ieșiri fișiere.
- <sstream> intrări/ieșiri pentru streamuri de tip string.
- <iomanip> conține manipulatori.
- <iosfwd> declarații pentru toate clasele din biblioteca IO.

Suprîncărcare operatori <<, >> pentru tipuri utilizator

- Se face similar ca și pentru orice operator
- sunt operatori binari
- primul operand este un stream (pe stânga), pe partea dreaptă avem un obiect de tipul nou (user defined).

<pre>class Product { public: Product(int code, string desc, double price) ~Product(); double getCode() const { return code; } double getPrice() const { return price; } friend ostream& operator<< (ostream& stream, const Product& prod); private: int code; string description; double price; };</pre>	<pre>ostream& operator<<(ostream& stream, const Product& prod) { stream << prod.code << " "; stream << prod.description << " "; stream << prod.price; return stream; }</pre>
<pre>void testStandardOutputUserType() { Product p = Product(1, "prod", 21.1); cout << p << "\n"; Product p2 = Product(2, "prod2", 2.4); cout << p2 << "\n"; }</pre>	
<pre>void testInsertionOperator() { Product p{ 1,"prod1",100 }; std::ostringstream os; os << p; assert(os.str() == "1 prod1 100"); }</pre>	

Manipulatori.

- Manipulatorii sunt funcții cu semantică specială, folosite împreună cu operatorul de inserare/extragere (<< , >>)
- Sunt funcții obișnuite, se pot și apela (se da un argument de tip stream)
- Manipulatorii se folosesc pentru a schimba modul de formatare a streamului sau pentru a insera caractere speciale.
- Există o variabilă membră în ios (x_flags) care conține informații de formatare pentru operare I/O , x_flags poate fi modificat folosind manipulatori
- sunt definite în modulul **iostream.h** (endl, dec, hex, oct, etc) și **iomanip.h** (setbase(int b),setw(int w),setprecision(int p))

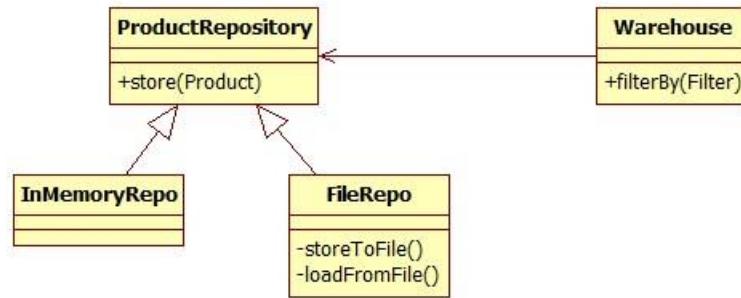
```
void testManipulators() {
    cout << oct << 9 << endl << dec << 9 << endl;
    oct(cout);
    cout << 9;
    dec(cout);
}
```

Citire/scriere obiecte

```
void testWriteReadUserObjFile() {
    ofstream out;
    out.open("test2.out", ios::out | ios::trunc);
    if (!out.is_open()) {
        return;
    }
    Product p1(1, "p1", 1.0);
    out << p1 << endl;
    Product p2(2, "p2", 2.0);
    out << p2;
    out.close();

    //read
    ifstream in("test2.out");
    if (!in.is_open()) {
        cout << "Unable to open";
        return;
    }
    Product p(0, "", 0);
    while (!in.eof()) {
        in >> p;
        cout << p << endl;
    }
    in.close();
}
```

Exemplu: FileRepository



Varianta 2

Flux - Stream

Noțiunea de flux este o abstractizare, reprezentă orice dispozitiv pe care executăm operații de intrări / ieșiri (citire/scriere)

Stream este un flux de date de la un set de surse (tastatură, fișier, zonă de memorie) către un set de destinații (ecran, fișier, zonă de memorie)

În general fiecare stream este asociat cu o sursă sau destinație fizică care permite citire/scriere de caractere.

De exemplu: un fișier pe disk, tastatura, consola. Caracterele citite/scrise folosind streamuri ajung / sunt preluate de la dispozitive fizice existente (hardware).

Stream de fișiere - sunt obiecte care interacționează cu fișiere, dacă am atașat un stream la un fișier orice operație de scriere se reflectă în fișierul de pe disc.

Buffer

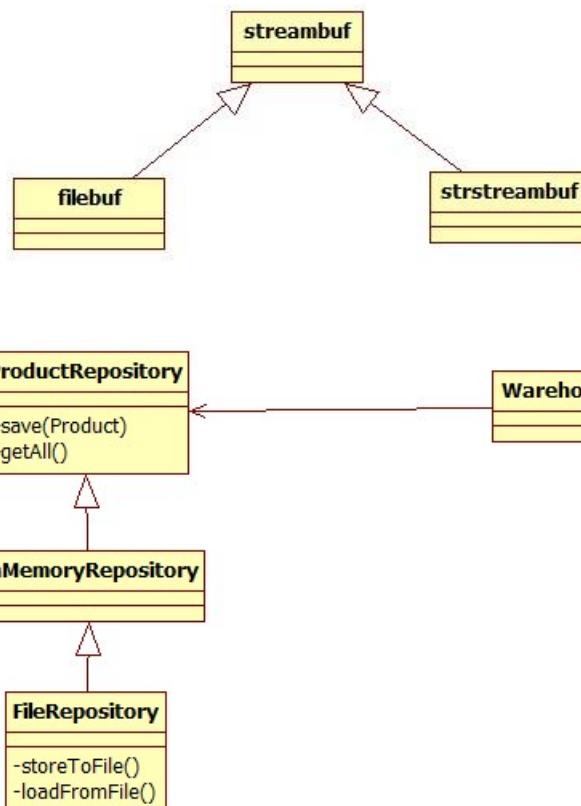
buffer este o zonă de memorie care este un intermediar între stream și dispozitiv.

De fiecare dată când se apelează metoda `put` (scrive un caracter), caracterul nu este trimis la dispozitivul destinație (ex. Fișier) cu care este asociat streamul. Defapt caracterul este inserat în buffer

Când bufferul este golit (`flush`), toate datele se trimit la dispozitiv (daca era un stream de ieșire). Procesul prin care conținutul bufferului este trimis la dispozitiv se numește sincronizare și se întâmplă dacă:

- streamul este închis, toate datele care sunt în buffer se trimit la dispozitiv (se scriu în fișier, se trimit la consolă, etc.)
- bufferul este plin. Fiecare buffer are o dimensiune, dacă se umple atunci se trimit conținutul lui la dispozitiv
- programatorul poate declanșa sincronizarea folosind manipulatoare: **flush**, **endl**
- programatorul poate declanșa sincronizarea folosind metoda **sync()**

Fiecare obiect stream din biblioteca standard are atașat un buffer (**streambuf**)



Curs 8

- **Polimorfism**
- **Interfețe grafice utilizator - Qt**

Curs 7 Moștenire, Polimorfism

- **Moștenire**
- **Polimorfism – Metode pur virtuale, Clase abstracte**
- **Operații de intrări ieșiri în C++**
 - **Fișiere**

Polimorfism

Proprietatea unor entități de:

- a se comporta diferit în funcție de tipul lor
- a reacționa diferit la același mesaj

Obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj (apel de metodă).

Proprietate a unui limbaj OO de a permite manipularea unor obiecte diferite prin intermediul unei interfețe comune

Polimorfism in C++

1 Moștenire

2 Suprascriere metoda in clasa derivata

3 Metoda virtuala in clasa de baza (sau pur virtuala)

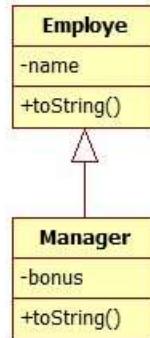
4 Accesat prin pointer sau referință (nu prin valoare => apare fenomenul de slicing)

Metoda potrivita se alege in timpul execuției (legare întârziata/dynamic binding/late binding) in loc sa fie ales la compilare (legare statica/static binding)

Mecanismul din spate: VTable – tabela cu adresa metodelor virtuale

Exercițiu polimorfism

Scriți codul C++ care corespunde diagramei UML de clase:



Compania are mai mulți angajați (angajați normali și manageri).

Metoda **toString** din clasa **Employe** returnează un string cu numele angajatului.

Metoda **toString** din clasa **Manager** returnează un string ce începe cu „Manager:” apoi numele managerului.

Scriți un program care creează o listă de angajați (atât angajați cât și manageri) și tipărește lista. Creați o funcție care primește o listă de angajați și tipărește stringul returnat de metoda **toString** de la **Employe** respectiv **Manager**.

Ex de output:

Ion

Manager: Pop

Rezolvare:

```
class Employee {
private:
    string name;
public:
    Employee(string n) :name{ n } {}
    virtual void toString() {
        cout << name;
    }
    virtual ~Employee() {
    }
};

class Manager :public Employee {
private:
    double bonus;
public:
    Manager(string n, double b):
        Employee{ n }, bonus{ b } {}

    void toString() override {
        cout << "Manager:";
        Employee::toString();
    }
};

void printAll(const vector<Employee*>& emps) {
    for (auto& emp : emps) {emp->toString();}
}

int main(){
    vector<Employee*> emps;
    emps.push_back(new Employee{ "Ion" });
    emps.push_back(new Manager{ "Ion",5.0 });
    emps.push_back(new Employee{ "Pop" });
    printAll(emps);
    for (auto emp : emps) {delete emp;} //delocam obiectele
}

//varianta cu unique_ptr
void printAllSmartPointer(const vector<unique_ptr<Employee>>& emps) {
    //unique_ptr nu se poate copia, trebuie sa folosim&
    for (auto& emp : emps) {emp->toString();}
}

void createAndPrintSmartPointer() {
    vector<unique_ptr<Employee>> emps;
    //make_unique - varianta preferata de a crea un unique_ptr
    //make_unique apeleaza constructorul de la Employee
    emps.push_back(make_unique<Employee>("Ion" ));
    emps.push_back(make_unique<Manager>("Ion",5.0 ));

    unique_ptr<Employee> up{ new Employee{ "Pop" } };
    //fiindca unique_ptr nu se poate copia trebuie sa facem move
    //dupa move variabila up nu mai contine pointerul
    emps.push_back(std::move( up));
    printAllSmartPointer(emps);
}
```

Exercițiu:

Scrieți un program C++ care:

- a. Definește o clasă **B** având un atribut privat **b** de tip întreg și o metodă de tipărire care afișează atributul **b** la ieșirea standard.
- b. Definește o clasă **D** derivată din **B** având un atribut **d** de tip sir de caractere și de asemenea o metodă de tipărire pe ieșirea standard care va afișa atributul **b** din clasa de bază și atributul **d**.
- c. Definește o funcție care construiește o listă conținând:
un obiect **o1** de tip **B** având **b** egal cu 8;
un obiect **o2** de tip **D** având **b** egal cu 5 și **d** egal cu "D5";
un obiect **o3** de tip **B** având **b** egal cu -3;
un obiect **o4** de tip **D** având **b** egal cu 9 și **d** egal cu "D9".
- d. Definește o funcție care primește o listă cu obiecte de tip **B** și tipărește lista de obiecte folosind metoda de tipărire.

Spații de nume

Introduc un domeniu de vizibilitate care nu poate conține duplicate

```
namespace testNamespace1 {  
    class A {  
    };  
}  
namespace testNamespace2 {  
    class A {  
    };  
}
```

Accesul la elementele unui spațiu de nume se face folosind operatorul de rezoluție

```
void testNamespaces() {  
    testNamespace1::A a1;  
    testNamespace2::A a2;  
}
```

Folosind directiva using putem importa:

toate elementele definite într-un spațiu de nume

```
void testUsing() {  
    using namespace testNamespace1;  
    A a;  
}
```

Doar clasa/metoda pe care vrem sa folosim

```
using std::vector;  
using std::copy_if;  
using std::cout;
```

Qt Toolkit

Qt este un framework pentru crearea de aplicații cross-platform (același code pentru diferite sisteme de operare, dispozitive) în C++.

Folosind QT putem crea interfețe grafice utilizator. Codul odată scris poate fi compuat pentru diferite sisteme de operare, platforme mobile fără a necesita modificări în codul sursă.

Qt suportă multiple platforme de 32/64-bit (Desktop, embedded, mobile).

- Windows (MinGW, MSVC)
- Linux (gcc)
- Apple Mac OS
- Mobile / Embedded (Windows CE, Symbian, Embedded Linux)

Este o librărie C++ dar există posibilitatea de a folosi și din alte limbiage: C# ,Java, Python(PyQt), Ada, Pascal, Perl, PHP(PHP-Qt), Ruby(RubyQt)

Qt este disponibil atât sub licență GPL v3, LGPL v2 cât și licențe comerciale.

Exemple de aplicații create folosind Qt:

Google Earth, KDE (desktop environment for Unix-like OS), Adobe Photoshop Album, etc

Resurse: <https://www.qt.io/>

QT - Module și utilitare

- **Qt Library** - bibliotecă de clase C++, oferă clasele necesare pentru a crea aplicații (cross-platform applications)
- **Qt Creator** - mediu de dezvoltare integrat (IDE) pentru a crea aplicații folosind QT
- **Qt Designer** – instrument de creare de interfețe grafice utilizator folosind componente QT
- **Qt Assistant** – aplicație ce conține documentație pentru Qt și facilitează accesul la documentațiile diferitelor partii din QT
- **Qt Linguist** – suport pentru aplicații care funcționează în diferite limbi (internationalizare)
- **qmake** – aplicație folosită în procesul de creare proiecte/compilare

Download instal QT

Ultima versiune QT 6.0.3

<https://www.qt.io/download>

Alegeți varianta Open Source (Downloads for open source users).

Installer ce downloadeaza cele necesare. Atenție daca instalați tot ocupa destul de mult spațiu pe disk (peste 10Gb), puteți sa instalați doar cele esențiale (ex. Instalați doar varianta compilata cu VisualStudio 2019)

Important: Sa va asigurati ca folositi varianta potrivita pentru calculatorul vostru:

- procesorul 32 vs 64 biti
- sistemul de operare (windows, linux, mac)
- compilatorul folosit: MinGW (eclipse) vs Microsoft (Visual studio) – in timpul instalarii

După ce aveți Qt instalat aveți mai multe variante de a dezvolta aplicații C++ cu Qt:

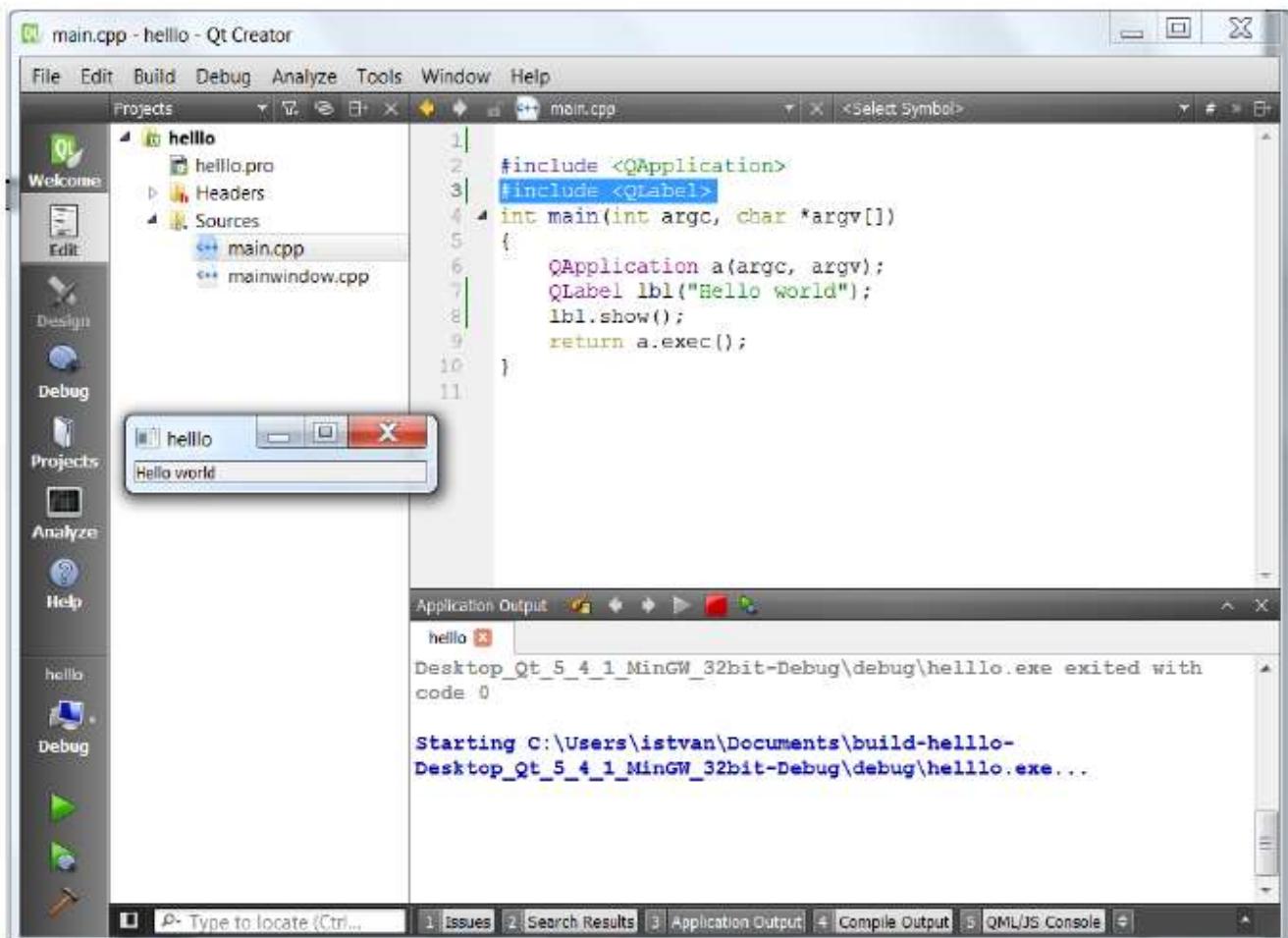
- Folosind Visual Studio + Extensia Qt - varianta preferata la aceasta materie
- QtCreator – IDE instalat cu Qt
- Din linie de comanda folosind utilitarul qmake – puteți ulterior folosi orice IDE care poate lucra cu proiecte care folosesc makefile

QT Creator – Hello world

Prerequisite: Download/install Qt

Qt Creator - IDE pentru dezvoltare de aplicații Qt

File -> New File or Project -> Qt Widget Application



Qt Hello World – Exemplu folosire linie de comanda

Prerequisite: MinGW; MinSYS; Qt installed

Variabila PATH trebuie sa contine calea catre qt (qmake.exe), mingw (g++.exe), minsys (make.exe)

```
set PATH=C:\MinGW\msys\1.0\bin\  
set PATH=%PATH%;C:\Qt\5.10.1\mingw491_32\bin\  
set PATH=%PATH%;C:\Qt\Tools\mingw491_32\bin\
```

Pas 1: Creati un fisier .cpp (main.cpp)

```
#include <QApplication>  
#include <QLabel>  
int main(int argc, char *argv[]){  
    QApplication a(argc, argv);  
    QLabel lbl("Hello world");  
    lbl.show();  
    return a.exec();  
}
```

Pas 2: Executați: **qmake -project** => se generează un fișier .pro

Pas 3: Editați fișierul .pro Adaugati: **QT += core gui**

Pas 4: Executați: **qmake** => Makefiles generated

Pas 5: Executați: **make** => proiectul este compilat, link-editat => exe

Pas 6: rulați aplicația **main.exe**

Puteți edita fișierele .cpp cu ce editor doriți, compilați cu comanda make

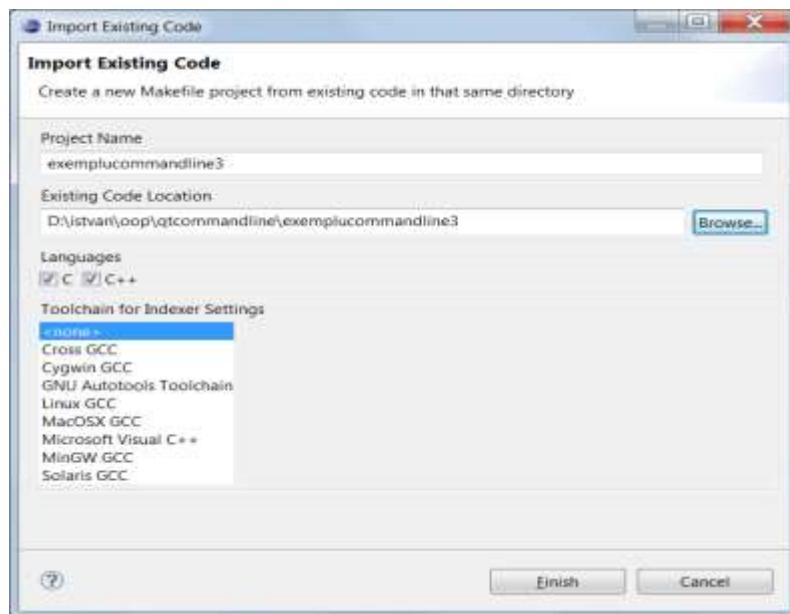
La pasul 4 s-a generat un proiect care folosește makefile, astfel de proiecte se pot importa în majoritatea IDE-urilor care suportă C++

Qt Hello World - Eclipse

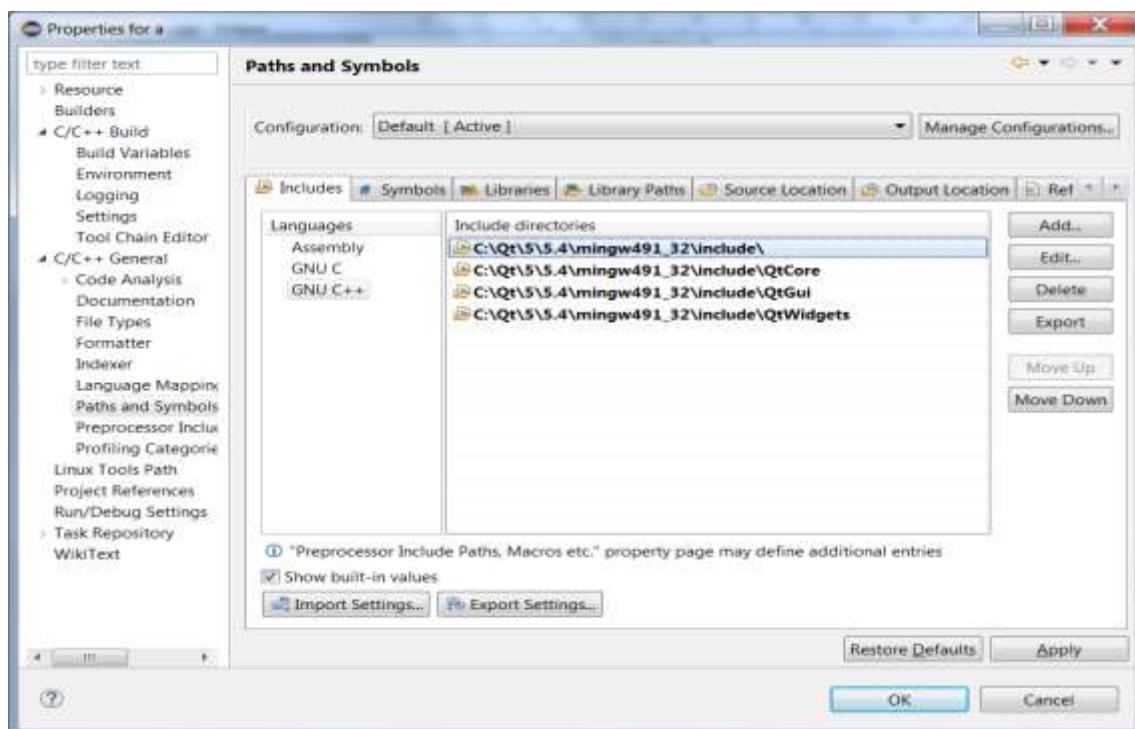
Prerequisite: Working Eclipse CDC (MinGW), Qt Library installed

Execuția Pașii 1 - 5

Proiectul se poate importa în Eclipse: File->New-> Makefile project with Existing Code



Adaugă libraria Qt: Project->Properties->C/C++ General->Path and Symbols → Includes



Qt Hello World – Visual Studio

Instalați pachetul de extensie QT in Visual Studio:

Tools->Extension and Updates->Qt Visual Studio Tools

Apare un nou meniu (Qt VS Tools) și noi tipuri de proiect pe care puteți crea în Visual Studio (File-> New->Project-> **Qt Widgets Application**)

Adăugați în main:

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QLabel *label = new QLabel("hello world");
    label->show();
    return app.exec();
}
```

Comparați/Rulați aplicația

QApplication

Clasa QApplication gestionează fluxul de evenimente și setările generale pentru aplicațiile Qt cu interfață grafică utilizator (GUI)

QApplication preia evenimentele de la sistemul de operare și distribuie către componentele Qt (main event loop), toate evenimentele ce provin de la sistemul de ferestre (windows, kde, x11,etc) sunt procesate folosind această clasă.

Pentru orice aplicație Qt cu GUI, există un obiect QApplication (indiferent de numărul de ferestre din aplicație există un singur obiect QApplication)

Responsabilități:

- inițializează aplicația conform setărilor sistem
- gestionează fluxul de evenimente - generate de sistemul de ferestre (x11, windows) și distribuite către componentele grafice Qt (widgets).
- Are referințe către ferestrele aplicației
- definește look and feel

Pentru aplicații Qt fără interfață grafică se folosește QCoreApplication.

`app.exec();`

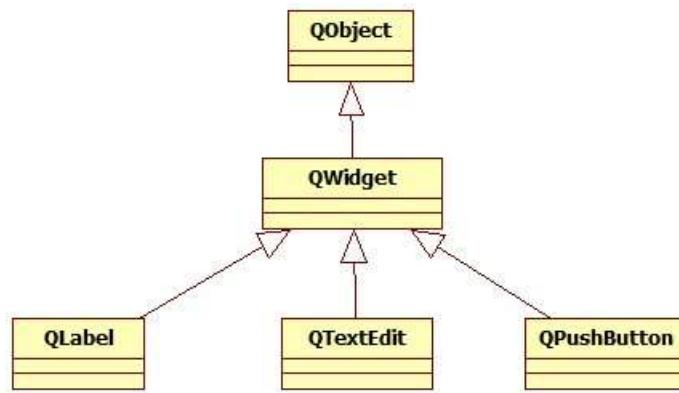
- pornește procesarea de evenimente din QApplication (event loop). În timp ce rulează aplicația evenimente sunt generate și trimise către componentele grafice.

Componente grafice QT (widgets)

Sunt elementele de bază folosite pentru a construi interfețe grafice utilizator

- butoane, etichete, căsuțe de text , etc

Orice componentă grafică Qt (widget) poate fi adăugat pe o fereastră sau deschis independent într-o fereastră separată.



Dacă componenta nu este adăugat într-o componentă părinte avem de fapt o fereastră separată .

Fereastrele separă vizual aplicațiile între ele și sunt decorate cu diferite elemente (bară de titlu, instrumente de poziționare, redimensionare, etc)

Widget : Etichetă, buton, căsuță de text, listă

QLabel

- QLabel se folosește pentru a prezenta un text sau o imagine. Nu oferă interacțiune cu utilizatorul
- QLabel este folosit de obicei ca și o etichetă pentru o componentă interactivă. QLabel oferă mecanism de mnemonic, o scurtătură prin care se setează focusul pe componenta atașată (numit "buddy").

```
QLabel *label = new QLabel("hello world");
label->show();

QLineEdit txt(parent);
QLabel lblName("&Name:", parent);
lblName.setBuddy(&txt);
```

QPushButton

QPushButton widget - buton

- Se apasă butonul (click) pentru a efectua o operație
- Butonul are un text și optional o iconă. Poate fi specificat și o tastă rapidă (shortcut) folosind caracterul & în text

```
QPushButton btn("&TestBTN");
btn.show();
```

Widget : Etichetă, buton, căsuță de text, listă

QLineEdit

- Căsuță de text (o singură linie)
- Permite utilizatorului să introducă informații. Oferă funcții de editare (undo, redo, cut, paste, drag and drop).

```
QLineEdit txtName;  
txtName.show();
```

QTextEdit este o componentă similară care permite introducere de text pe mai multe linii și oferă funcționalități avansate de editare/vizualizare.

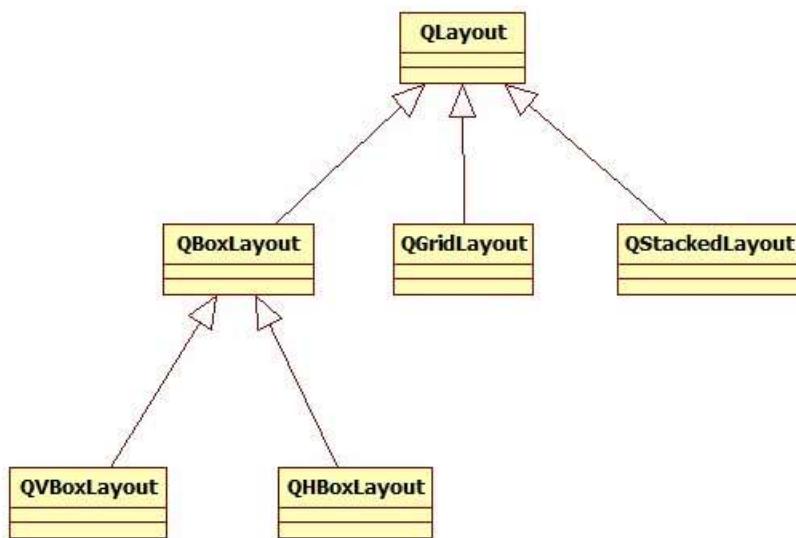
QListWidget

Prezintă o listă de elemente

```
QListWidget *list = new QListWidget;  
new QListWidgetItem("Item 1", list);  
new QListWidgetItem("Item 2", list);  
QListWidgetItem *item3 = new QListWidgetItem("Item 3");  
list->insertItem(0, item3);  
list->show();
```

Layout management

- Orice QWidget are o componentă părinte.
- Sistemul Qt layout oferă o metodă de a aranja automat componentele pe interfață.
- Qt include un set de clase pentru layout management, aceste clase oferă diferite strategii de aranjare automată a componentelor.
- Componentele sunt poziționate / redimensionate automat conform strategiei implementate de layout manager și luând în considerare spațiul disponibil pe ecran. Folosind diferite layouturi putem crea interfețe grafice utilizator care acomodează diferite dimensiuni ale ferestrei.



Layout management

```
QWidget *wnd = new QWidget;
QHBoxLayout *hLay = new QHBoxLayout();
QPushButton *btn1 = new QPushButton("Bt &1");
QPushButton *btn2 = new QPushButton("Bt &2");
QPushButton *btn3 = new QPushButton("Bt &3");
hLay->addWidget(btn1);
hLay->addWidget(btn2);
hLay->addWidget(btn3);
wnd->setLayout(hLay);
wnd->show();
```

```
QWidget *wnd2 = new QWidget;
QVBoxLayout *vLay = new QVBoxLayout();
QPushButton *bttn1=new QPushButton("B&1");
QPushButton *bttn2= new QPushButton("B&2");
QPushButton *bttn3= new QPushButton("B&3");
vLay->addWidget(bttn1);
vLay->addWidget(bttn2);
vLay->addWidget(bttn3);
wnd2->setLayout(vLay);
wnd2->show();
```

GUI putem compune multiple componente care folosesc diferite strategii de aranjare pentru a crea interfață utilizator dorită

```
QWidget * wnd3 = new QWidget;
QVBoxLayout * vL = new QVBoxLayout;
wnd3->setLayout(vL);
//create a detail widget
QWidget * details = new QWidget;
QFormLayout * fL = new QFormLayout;
details->setLayout(fL);
QLabel * lblName = new QLabel("Name");
QLineEdit * txtName = new QLineEdit;
fL->addRow(lblName, txtName);
QLabel * lblAge = new QLabel("Age");
QLineEdit * txtAge = new QLineEdit;
fL->addRow(lblAge, txtAge);
//add detail to window
vL->addWidget(details);
QPushButton * store = new QPushButton("&Store");
vL->addWidget(store);
//show window
wnd3->show();
```

Layout management

addStretch() se folosește pentru a consuma spațiu. Practic se adaugă un spațiu care se redimensionează în funcție de strategia de aranjare

```
QHBoxLayout* btsL = new QHBoxLayout;
bts->setLayout(btsL);
QPushButton* store = new QPushButton("&Store");
btsL->addWidget(store);
btsL->addStretch();
QPushButton* close = new QPushButton("&Close");
btsL->addWidget(close);
```

Layout manager o să adauge spațiu între butonul Store și Close

Layout management

Cum construim interfețele grafice:

- creăm componentele necesare
- setăm proprietățile componentelor dacă este necesar
- adăugăm componenta la un layout (layout manager se ocupă cu dimensiunea poziția componentelor)
- conectăm componentele între ele folosind mecanismul de signal și slot

Avantaje:

- oferă un comportament consistent indiferent de dimensiunea ecranului/ferestrei, se ocupa de rearanjarea componentelor în caz de redimensionare a componentei
- setează valori implicate pentru componentele adăugate
- se adaptează în funcție de fonturi și alte setări sistem legate de interfețele utilizator.
- Se adaptează în funcție de textul afișat de componentă. Aspect important dacă avem aplicații care funcționează în multiple limbi (se adaptează componenta pentru a evita trunchierea de text).
- Dacă adugăm/ștergem componente restul componentelor sunt rearanjate automat (similar și pentru `show()` `hide()` pentru o componentă)

Pozitionare cu coordonate absolute

```
/**  
 * Create GUI using absolute positioning  
 */  
void createAbsolute() {  
    QWidget* main = new QWidget();  
    QLabel* lbl = new QLabel("Name:", main);  
    lbl->setGeometry(10, 10, 40, 20);  
    QLineEdit* txt = new QLineEdit(main);  
    txt->setGeometry(60, 10, 100, 20);  
    main->show();  
    main->setWindowTitle("Absolute");  
}  
  
/**  
 * Create the same GUI using form layout  
 */  
void createWithLayout() {  
    QWidget* main = new QWidget();  
    QFormLayout *fL = new QFormLayout(main);  
    QLabel* lbl = new QLabel("Name:", main);  
    QLineEdit* txt = new QLineEdit(main);  
    fL->addRow(lbl, txt);  
    main->show();  
    main->setWindowTitle("Layout");  
    //fix the height to the "ideal" height  
    main->setFixedHeight(main->sizeHint().height());  
}
```

Dezavantaje:

- Utilizatorul nu poate redimensiona fereastra (la redimensionare componentele rămân pe loc și fereastra nu arată bine, nu folosește spațiu oferit).
- Nu ia în considerare fontul, dimensiunea textului (orice schimbare poate duce la text trunchiat).
- Pentru unele stiluri (look and feel) dimensiunea componentelor trebuie ajustată.
- Pozițiile și dimensiunile trebuie calculate manual (ușor de greșit, greu de întreținut)

Documentație Qt

- Qt Reference Documentation – conține descrieri pentru toate clasele, metodele din Qt
- Este disponibil în format HTML (directorul doc/html din instalarea Qt) și se poate citi folosind orice browser
- Qt Assistant – aplicație care ajută programatorul să caute în documentația Qt (mai ușor de folosit decât varianta cu browser)
- Documentația este disponibilă și online <http://doc.qt.io/qt-5/index.html>
- Pentru orice clasă găsiți descrieri detaliate pentru metode, atribute, semnale , sloturi

Curs 9-10 – Interfețe grafice utilizator

- **Semnale și sloturi**
- **Componente definite de utilizator**
- **Callback/Observer**
- **Evenimente de mouse/tastatura**
- **Graphics View Framework**

Curs 8

- **Polimorfism – exercitii**
- **Spații de nume (Namespace)**
- **Interfețe grafice utilizator - Qt**

Semnale și sloturi (Signals and slots)

- Semnalele și sloturile sunt folosite în Qt pentru comunicare între obiecte
- Este mecanismul central în Qt pentru crearea de interfețe utilizator
- Mecanismul este specific Qt, diferă de mecanismele folosite în alte biblioteci de GUI.
- Când facem modificări la o componentă (scriem un text, apăsăm butonul, selectăm un element, etc.) dorim ca alte părți ale aplicației să fie notificate (să actualizăm alte componente, să executăm o metodă, etc).

Ex. Dacă utilizatorul apasă pe butonul **Close**, dorim să închidem fereastra, adică să apelăm metoda **close()** al ferestrei.

- În general bibliotecile pentru interfețe grafice folosesc callback pentru această interacțiune.
- Callback
 - este un pointer la o funcție,
 - dacă într-o metodă dorim să notificăm apariția unui eveniment, putem folosi un pointer la funcție (callback, primit ca parametru).
 - În momentul în care apare evenimentul se apelează această funcție (call back)
- Dezavantaje callback în c++ :
 - dacă avem mai multe evenimente de notificat, ne trebuie funcții separate callback sau să folosim parametrii generici (`void*`) care nu se pot verifica la compilare
 - metoda care apelează metoda callback este cuplat tare de callback (trebuie să știe semnatura funcției, parametrii, etc. Are nevoie de referință la metoda callback).

Signal. Slot.

- Semnalul (**signal**) este emis (ex.: `&QPushButton::clicked`) la apariția unui eveniment
- Componentele Qt (widget) emit semnale pentru a indica schimbări de stări generate de utilizator
- Un **slot** este o funcție care este apelat ca și răspuns la un semnal.
- Semnalul se poate conecta la un slot, astfel la emitera semnalului slotul este automat executat

```
QApplication a(argc, argv);
QPushButton* btn = new QPushButton("&Close");
QObject::connect(btn, &QPushButton::clicked, &a, &QApplication::quit);
btn->show();
```

- **Slot** poate fi folosit pentru a reacționa la semnale, dar ele sunt de fapt metode normale sau funcții lambda.
- Semnalele și sloturile sunt decuplate între elementele
 - Obiectele care emit semnale nu au cunoștințe despre sloturile care sunt conectate la semnal
 - slotul nu are cunoștință despre semnalele conectate la el
 - Decuplarea permite crearea de componente cu adevărat independente folosind Qt.
- În general componentele Qt's au un set predefinit de semnale. Se pot adăuga și semnale noi folosind moștenire (clasa derivată poate adăuga semnale noi) sau se pot folosi funcții lambda

Conecțarea semnalelor cu sloturi

Folosind metoda `QObject::connect` putem conecta semnale și sloturi

```
QWidget* createButtons(QApplication &a) {
    QWidget* btns = new QWidget;
    QHBoxLayout* btnsL = new QHBoxLayout;
    btns->setLayout(btnsL);
    QPushButton* store = new QPushButton("&Store");
    btnsL->addWidget(store);
    QPushButton* close = new QPushButton("&Close");
    btnsL->addWidget(close);
    //connect the clicked signal from close button to the quit slot
    (method)
    QObject::connect(close, QPushButton::clicked,
                      &a, QApplication::quit);
    return btns;
}
```

În urma conectării – slotul este apelat în momentul în care se generează semnalul. Sloturile sunt funcții normale, apelul este la fel ca la orice funcție C++. Singura diferență între un slot și o funcție este că slotul se poate conecta la semnale .

La un semnal putem conecta mai multe sloturi, în urma emiteri semnalului se vor apela sloturile în ordinea în care au fost conectate

Există o corespondență între signatura semnalului și signatura slotului (parametrii trebuie să corespundă).

Slotul poate avea mai puține parametrii, parametrii de la semnal care nu au corespondent la slot se vor ignora.

Conecțarea semnalelor cu sloturi – referinta metode, lambda

Înainte de QT 5 semnale și sloturile se indicau folosind macrourile SIGNAL,SLOT

Odată cu QT 5 este permis folosirea de funcții lambda, referințe la funcții/metode

```
QObject::connect(close, &QPushButton::clicked,  
                  &a, QCoreApplication::quit);
```

```
QObject::connect(close, &QPushButton::clicked, [&a]() {  
    a.quit();  
});
```

```
QObject::connect(close, SIGNAL(clicked()), &a, SLOT(quit()));
```

`&QPushButton::clicked` este o referință la funcția `clicked`, metoda din clasa `QPushButton`

Cele trei variante de mai sus sunt echivalente (realizează același lucru) dar este de preferat să folosim variantele tipizate atât pentru semnale (`&QPushButton::clicked`) cat și pentru sloturi (`QCoreApplication::quit`).

Folosind funcții lambda și referințe la funcții compilatorul poate verifica dacă semnalul și slotul este compatibil

Conecțarea semnalelor cu sloturi

```
QSpinBox *spAge = new QSpinBox();
QSlider *slAge = new QSlider(Qt::Horizontal);

//Synchronise the spinner and the slider
//Connect spin box - valueChanged to slider setValue
QObject::connect(spAge, &QSpinBox::valueChanged, slAge,
&QSlider::setValue);
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge,
SIGNAL(valueChanged(int)), spAge, SLOT(setValue(int)));

//prutem preluă valori de la semnal
QObject::connect(spAge, &QSpinBox::valueChanged,
                  slAge, &QSlider::setValue);
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge, &QSlider::valueChanged, [spAge](int val) {
    spAge->setValue(val);});
);
```

Dacă utilizatorul schimbă valoarea în spAge (Spin Box):

- se emite semnalul valueChanged(int) argumentul primește valoarea curentă din spinner
- fiindcă cele două componente (spinner, slider) sunt conectate se apelează metoda setValue(int) de la slider.
- Argumentul de la metoda valueChanged (valoarea curent din spinner) se transmite ca și parametru pentru slotul, metoda setValue din slider
- sliderul se actualizează pentru a reflecta valoarea primită prin setValue și emite la rândul lui un semnal valueChanged (valoarea din slider s-a modificat)
- sliderul este conectat la spinner, astfel slotul setValue de la spinner este apelat ca și răspuns la semnalul valueChanged.
- De data asta setValue din spinner nu emite semnal fiindcă valoarea curentă nu se schimba (este egal cu ce s-a primit la setValue) astfel se evită ciclul infinit de semnale

Componente definite de utilizator

- Se creează clase separate pentru interfață grafică utilizator
- componentele grafice create de utilizator extind componentele existente în Qt
- scopul este crearea de componente independente cu semnale și sloturi proprii

```
/**  
 * GUI for storing Persons  
 */  
class StorePersonGUI: public QWidget {  
public:  
    StorePersonGUI();  
private:  
    QLabel *lblName;  
    QLineEdit *txtName;  
    QLabel *lblAdr;  
    QLineEdit *txtAdr;  
    QSpinBox *spAge;  
    QLabel *lblAge2;  
    QSlider *slAge;  
    QLabel *lblAge3;  
    QPushButton* store;  
    QPushButton* close;  
    /**  
     * Assemble the GUI  
     */  
    void buildUI();  
    /**  
     * Link signals and slots to define the behaviour of the GUI  
     */  
    void connectSignalsSlots();  
};
```

QMainWindow

- În funcție de componenta ce definim putem extinde clasa QWidget, QMainWindow, QDialog etc.
- Clasa QMainWindow poate fi folosit pentru a crea fereastra principală pentru o aplicație cu interfață grafică utilizator.
- QMainWindow are propriul layout, se poate adăuga toolbar, meniuri, status bar.
- QMainWindow definește elementele de bază ce apar în mod general la o aplicație

Layout QMainWindow:

- Meniu – pe partea de sus

```
QAction *openAction = new QAction("&Load", this);
QAction *saveAction = new QAction("&Save", this);
QAction *exitAction = new QAction("E&xit", this);
fileMenu = menuBar()->addMenu("&File");
fileMenu->addAction(openAction);
fileMenu->addAction(saveAction);
fileMenu->addSeparator();
fileMenu->addAction(exitAction);
```

- Toolbar

```
QToolBar* fileToolBar = addToolBar("&File");
fileToolBar->addAction(openAction);
fileToolBar->addAction(saveAction);
```

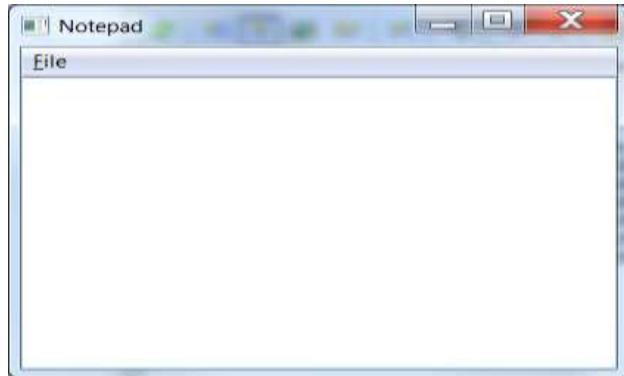
- Componenta din centru

```
middle = new QTableWidget(10, 10, this);
setCentralWidget(middle);
```

- Status bar - în partea de jos

```
statusBar()->showMessage("Status Message ....");
```

Notepad



```
class Notepad : public QMainWindow
{
public:
    Notepad();

private:
    void open();
    void save();
    void quit2();

    QAction *openAction;
    QAction *saveAction;
    QAction *exitAction;

    connect(openAction, &QAction::triggered, this, &Notepad::open);
    connect(saveAction, &QAction::triggered, this, &Notepad::save);
    connect(exitAction, &QAction::triggered, this, &Notepad::quit2);

void Notepad::open()
{
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "",
                                                tr("Text Files (*.txt);;C++ Files (*.cpp *.h)"));

    if (fileName != "") {
        QFile file(fileName);
        if (!file.open(QIODevice::ReadOnly)) {
            QMessageBox::critical(this, tr("Error"), tr("Could not open file"));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    }
}
```

Exemplu PetStore:

```
class PetStoreGUI : public QWidget{
private:
    PetController& ctr;
    QListWidget* lst;
    QPushButton* btnSortByPrice;
    QPushButton* btnSortByType;
    QLineEdit* txtSpecies;
    QLineEdit* txtType;
    QLineEdit* txtPrice;
    void initGUICmps();
    void connectSignalsSlots();
    void reloadList(std::vector<Pet> pets);
public:
    PetStoreGUI(PetController& ctr) :ctr{ ctr } {
        initGUICmps();
        connectSignalsSlots();
        reloadList(ctr.getAllPets());
    }
};

void PetStoreGUI::reloadList(std::vector<Pet> pets) {
    lst->clear();
    for (auto& p : pets) {
        QListWidgetItem* item = new QListWidgetItem(p.getSpecies(), lst);
        item->setData(Qt::UserRole, p.getType()); //adaug in lista (invizibil) si type
        //lst->addItem(p.getSpecies());
    }
}

void PetStoreGUI::connectSignalsSlots() {
    //cand se emite semnalul clicked de pe buton reincarc lista
    QObject::connect(btnSortByPrice, &QPushButton::clicked, [&]() {
        reloadList(ctr.getSortByPrice());
    });
    //cand se emite semnalul clicked de pe buton reincarc lista
    QObject::connect(btnSortByType, &QPushButton::clicked, [&]() {
        reloadList(ctr.getSortByType());
    });
    //cand se selecteaza elementul din lista incarc detaliile
    QObject::connect(lst, &QListWidget::itemSelectionChanged, [&]() {
        if (lst->selectedItems().isEmpty()) {
            //nu este nimic selectat (golesc detaliile)
            txtSpecies->setText("");
            txtType->setText("");
            txtPrice->setText("");
            return;
        }
        QListWidgetItem* selItem = lst->selectedItems().at(0);
        txtSpecies->setText(selItem->text());
        txtType->setText(selItem->data(Qt::UserRole).toString());
    });
}
```

Clase QT - utile

QString – sir de caractere Unicode

Metodele care primesc un parametru QString accepta si char* (QString are un constructor cu char*)

```
txtSpecies->setText("ceva text");
```

Exista metode de a transforma din QString in std::string si invers

```
QString s = "145";
std::string ss = s.toStdString();
QString s2 = QString::fromStdString(ss);
```

Exista posibilitatea de a transforma numere in QString si invers

```
QString s = "145";
int a = s.toInt();
double d = 3.8;
QString s3 = QString::number(d);
```

QList/QVector lista/vector variante Qt pentru containere din STL

au operatii similare ca cele din stl: [0], at(0), pop(), isEmpty(), size(), etc

Se pot transforma in variantele stl

```
QList<int> l;
l.push_back(3);
l.push_front(4);
l[1] = 4; l.at(0);
```

```
std::list<int> ll = l.toStdList();
QList<int> l2 = QList<int>::fromStdList(ll);
```

QMessageBox

Putem afisa o fereastra cu informatii

```
QMessageBox::information(this, "Info", "Selection changed");//afiseaza mesaj
int ret = QMessageBox::warning(this, "My Application",
    "The document has been modified.\nDo you want to save your changes?",
    QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel,
    QMessageBox::Save);
if (ret == QMessageBox::Save) {
    //do save
}
```

QDebug – stream pentru a scrie informații ce ajuta la depanarea programelor

```
qDebug() << "Date:" << QDate::currentDate();
```

Item based Widgets – QListWidget, QTableWidgetItem

QListWidget / QListWidgetItem	QTableWidget / QTableWidgetItem
<pre> QListWidget* lst = new QListWidget; //se pot adauga elemente QListWidgetItem* item = new QListWidgetItem("Bla", lst); </pre>	<pre> //se creaza int nrLinii = 4; int nrColoane = 3; QTableWidget* tbl = new QTableWidget{ nrLinii,nrColoane }; //se pot adauga elemente QTableWidgetItem* cellItem1 = new QTableWidgetItem("Linie1"); tbl->setItem(0, 0, cellItem1); tbl->setItem(0, 1, new QTableWidgetItem("Linie1 coloana2")); </pre>
<pre> //se poate configura modul de selectie lst->setSelectionMode(QAbstractItemView::SingleSelection); //se poate obtine selectia auto selItms = lst->selectedItems(); </pre>	<pre> //se poate configura modul de selectie tbl->setSelectionBehavior(QAbstractItemView::SelectRows); tbl->setSelectionMode(QAbstractItemView::SingleSelection); //se poate obtine selectia auto selTblItms = tbl->selectedItems(); </pre>
<pre> //putem reacționa la semnale QObject::connect(lst, &QListWidget::itemSelectionChanged, [lst]() { qDebug() << "Selectie \n" << lst->selectedItems() << "\n"; }); </pre>	<pre> //putem reacționa la semnale QObject::connect(tbl, &QTableWidget::itemSelectionChanged, [tbl]() { qDebug() << "Selectie tabel\n" << tbl->selectedItems()<<"\n"; }); </pre>

Fiecare celula din tabel / lista conține textul dar și alte informații:

<pre> //informatii suplimentare in item item->setBackground(QBrush{ Qt::red, Qt::SolidPattern }); item->setForeground(Qt::blue); item->setData(Qt::UserRole, QString{ "informatii care nu se vad" }); item->setCheckState(Qt::Checked); item->setIcon(QApplication::style()->standardIcon(QStyle::SP_BrowserReload)); </pre>
<pre> //informatii suplimentare pentru fiecare celula cellItem1->setBackground(QBrush{ Qt::red, Qt::SolidPattern }); cellItem1->setForeground(Qt::blue); cellItem1->setData(Qt::UserRole, QString{ "informatii care nu se vad" }); cellItem1->setCheckState(Qt::Unchecked); cellItem1->setIcon(QApplication::style()->standardIcon(QStyle::SP_ArrowBack)); </pre>

Qt Build system

O aplicație c++ conține fișiere header (.h) și fișiere (.cpp)

Procesul de build pentru o aplicație c++ :

- se compilează fișierele cpp folosind un compilator (fișierele sursă pot referi alte fișiere header) → fișiere obiect (.o)
- folosind un linker, se combină fișierele obiect (link edit) → fișier executabil(.exe)

Qt introduce pași adiționali:

Meta-object compiler (moc)

- compilatorul meta-object compiler ia toate clasele care încep cu macroul

`Q_OBJECT` și generează fișiere sursă C++ `moc_*.cpp`. Aceste fișiere sursă conțin informații despre clasele compilate (nume, ierarhia de clase) și informații despre signale și sloturi. Practic în fișierele surse generate găsim codul efectiv care apelează metodele slot când un semnal este emis (generate de moc).

User interface compiler

– Compilatorul pentru interfețe grafice are ca intrare fișiere create de Qt Designer și generează cod C++ (ulterior putem apela metoda `setupUi` pentru a instanția componentele GUI).

Qt resource compiler

– Se pot include icoane, imagini, fișiere text în fișierul executabil. Fișierele astfel incluse în executabil se pot accesa din cod ca și orice fișier de pe disc.

Qt Build – din linia de comandă

Util daca folosiți altceva decât Visual Studio. In cazul folosirii Visual Studio aceste aspecte sunt rezolvate automat folosind extensia Qt.

Se execută :

- qmake -project
 - generează un fișier de proiect Qt (.pro)
- qmake
 - pe baza fișierului .pro se generează un fișier make
- make
 - execută fișierul make (generat de qmake). Apelează tot ce e necesar pentru a transforma fișierele surse în fișier executabil(meta-object compiler, user interface compiler, resource compiler, c++ compiler, linker)

Semnale și sloturi definite - Q_OBJECT

Putem defini semnale și sloturi în componentele pe care le creăm

```
class Notepad : public QMainWindow
{
    Q_OBJECT
    ...
}
```

Macroul Q_OBJECT trebuie să apară la începutul definiției clasei. El este necesar în orice clasă unde vrem să adăugam semnale și sloturi noi.

Qt introduce un nou mecanism **meta-object system** care oferă :

- funcționalitate de semnale și sloturi (signals–slots)
- introspecție.

Introspecția este un mecanism care permite obținerea de informații despre clase dinamic, programatic în timpul rulării aplicației. Este un mecanism folosit pentru semnale și sloturi transparent pentru programator.

Prin introspecție se pot accesa meta-informații despre orice QObject în timpul execuției – lista de semanle, lista de sloturi, numele metodelor numele clasei etc.

Orice clasă care începe cu Q_OBJECT este QObject .

Instrumentul moc (meta-object compiler, moc.exe) inspectează clasele ce au Q_OBJECT în definiție și expun meta-informații prin metode normale C++. Moc generează cod c++ ce permite introspecție (in fișiere separate *.moc)

Semnale proprii

Folosind macroul *signals* se pot declara semnale proprii pentru componentele pe care le creăm.

```
signals:  
    void storeRq(QString* name,QString* adr );
```

Cuvântul rezervat **emit** este folosit pentru a emite un semnal.

```
emit storeRq(txtName->text(),txtAdr->text());
```

Semnalele sloturile definite de programator au același status și comportament ca și cele oferite de componentele Qt

Semnale proprii exemplu

```
class BrickGameEngine: public QObject{
    Q_OBJECT //e nevoie de acest macro daca vrem semnale custom
    int score = 0;
    int dead = 0;
    int nrBricks = 0;
    QTimer timer;
    int ballMoveDelay = 20;
    int elapsedMoves = 0;
signals:
    //semnale generate de engine
    void scoreChanged(int currentScore);
    void deadChanged(int currentNrDead);
    void advanceBoard();
    void brickCreated(int x, int y, int brickW, int brickH);
    void gameFinished(bool win);

public:
    BrickGameEngine() {
        emit scoreChanged(score);
        emit deadChanged(dead);
    }

    void brickHit() {
        score += 1;
        nrBricks--;
        emit scoreChanged(score);
    }

QObject::connect(&engine, &BrickGameEngine::gameFinished, [&](bool win) {
    if (win) {
        QMessageBox::information(this, "Info", "You win!!!");
    }
    else {
        QMessageBox::information(this, "Info", "You lose!!!");
    }
});
```

Function Callback

Mecanismul de semnal/slot este proprietar Qt, este o implementare specială (folosește moc compiler pentru a genera codul C++ automat) nu o să le găsiți în alte biblioteci. În general același idee se poate implementa folosind ideea de callback, idee des implementată în proiecte reale care nu folosesc Qt.

Ce este:

o funcție callback este o funcție apelată folosind o referință la funcție (pointer la funcție)

Ce problema rezolva:

Dorim să decuplăm obiectul care apelează metoda de obiectul care are metoda de apelat. Cel care apelează metoda nu trebuie să știe nimic despre obiectul apelat, numitorul comun între cele două obiecte este declarația funcției.

Când este util:

Ex1: creez o bibliotecă de sortare, doresc ca sortarea să fie independentă de metoda de comparare

Ex2: este util pentru notificări.

Dacă doresc să creez un obiect timer (măsoară timpul și emite notificări la intervale de timp). În momentul în care dezvolt clasa Timer nu știu cine o să folosească această clăsă și în ce scop. Ar trebui să creez o soluție generală astfel încât oricine are nevoie de funcționalitatea de timer să poată folosi cu codul lui.

Implementatorul clasei Timer o să definească prototipul funcției callback, și cel care dorește notificări trebuie să implementeze o funcție cu această semnătură

```
//callback function prototype
typedef void(*ProgressListener)(int percent);

//function that notify progress using callback
void someComputation(ProgressListener callback)
{
    for (int i = 0; i < 100; i++) {
        //do stuff
        callback(i);
    }
}

void onProgress(int percent) {
    std::cout << "progress:" << percent;
}

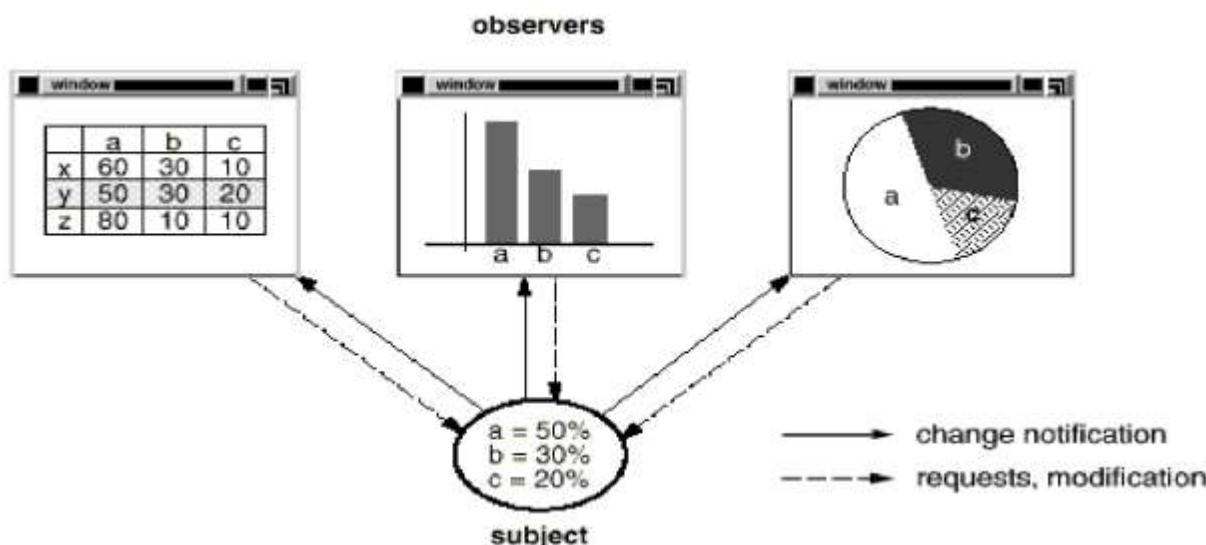
int main(int argc, char *argv[])
{
    someComputation(onProgress);
}
```

Sablonul Observer (Observer Design pattern)

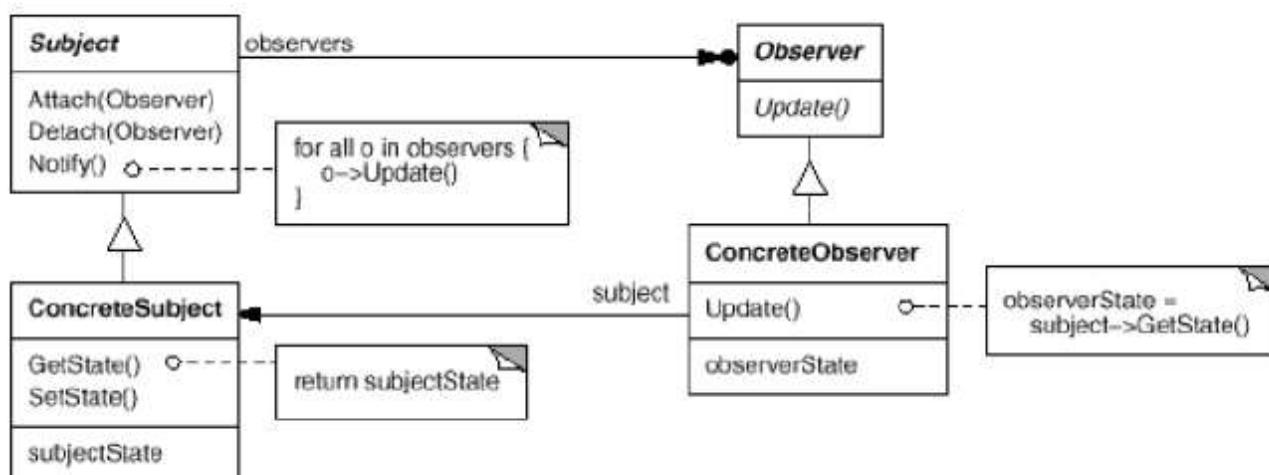
Intent : Definește o relație de dependență one-to-many între obiecte astfel încât în momentul în care obiectul schimba starea toate obiectele dependente sunt notificate automat

Also Known As: Publish-Subscribe

Motivation: O consecință a partitării sistemului în clase care cooperează este ca apare nevoie de a menține consistență între obiecte. Scopul este să menținem consistență dar în același timp să evităm cuplarea între obiecte (cuplarea reduce reutilizabilitatea).



Patten class structure



Observer – cod c++

```
class Observer {
public:
    virtual void update()=0;
};

class InterestedObj: public Observer {
public:
    void update() {
        std::cout << "Notified" << std::endl;
    }
};

void notify(Observer* obs) {
    obs->update();
}

class Observable {
public:
    void addObserver(Observer *obs) {
        observers.push_back(obs);
    }
    void doStuff() {
        //some stuff
        notifyObservers();
    }
private:
    std::vector<Observer*> observers;
    void notifyObservers() {
        for_each(observers.begin(), observers.end(), notify);
    }
};

int main() {
    Observable someObject;
    Observer* obs1 = new InterestedObj();
    Observer* obs2 = new InterestedObj();
    someObject.addObserver(obs1);
    someObject.addObserver(obs2);
    someObject.doStuff();
    return 0;
}
```

Gestiunea evenimentelor de mouse/tastatura/desenare

Evenimentele de mouse si de tastatura sunt transmise componentelor GUI Qt (orice clasa care extinde QWidget/QObject)

Când apare evenimentul Qt creează un obiect **QEvent** cu detalii despre eveniment (clase derivate: **QResizeEvent**, **QPaintEvent**, **QMouseEvent**, **QKeyEvent**, and **QcloseEvent**)

Gestiunea evenimentelor (**Event handlers**)

Evenimentele ajung la componentele Qt prin metode virtuale (sistemul Qt apelează metode virtuale definite în clasa QWidget).

Intr-o componentă definită de utilizator putem suprascrie metodele pentru a trata evenimentul:

```
class SampleW :public QWidget {
public:
    SampleW() {
        //If mouse tracking is disabled(default)the widget only receives mouse move events
        //when at least one mouse button is pressed while the mouse is being moved
        setMouseTracking(true);
    }
protected:
    void paintEvent(QPaintEvent* ev) override {
        qDebug() << "paint requested";
        ...
    }
    void mouseMoveEvent(QMouseEvent* ev) override {
        qDebug() << ev->pos();
    }
};
```

Trimitere de evenimente

Este posibil generarea de evenimente proprii folosind metoda **QCoreApplication::sendEvent()**

Se creează obiectul QEvent dorit și prin funcția sendEvent se poate trimite către componentele Qt.

QCoreApplication Gestionează o coadă de evenimente care sunt gestionate de un singur fir de execuție (Event loop).

Desenare low-level

Clasa **QPainter** permite desenarea “manuala”, are funcții optimizate pentru a desena orice elemente avem nevoie într-o aplicație grafică (linii, dreptunghi, elipse, imagini, etc)

In general obiect **QPainter** se folosește în interiorul metodei `paintEvent`, metoda ce este apelată de fiecare dată când s-a cerut redesenarea widgetului(`repaint()` sau `update()` au fost apelate ori de Qt automat ori de programatic din cod).

```
void paintEvent(QPaintEvent* ev) override {
    QPainter p{ this };

    p.drawLine(0, 0, width(), height());
    p.drawImage(x, 0, QImage("sky.jpg"));

    p.setPen(Qt::blue);
    p.setFont(QFont("Arial", 30));
    p.drawText(rect(), Qt::AlignTop | Qt::AlignHCenter, "Qt QPainter");

    p.fillRect(0, 100, 100, 100, Qt::Dense1Pattern);
}
```

Metoda `paintEvent` se poate suprascrie în orice clasa care moșteneste din `QWidget`

Graphics View Framework

Graphics View oferă suport pentru aplicații care gestionează și interacționează cu un număr mare de obiecte 2D (**2D graphical items**)

Oferă suport pentru:

- zoom/rotatii/transformari
- animatii
- tiparire
- drag and drop
- OpenGL

Arhitectura frameworkului Graphic View

QGraphicScene – conține elementele grafice **QGraphicItem**, se ocupa cu propagarea evenimentelor către obiectele grafice 2D

QGraphicView – vizualizează conținutul unei scene, este defapt un “scroll area” oferă posibilitatea de a naviga prin scena. Se pot atașa multiple vederi la același scena

QGraphicItem – și clasele derivate din el (**QGraphicsRectItem**, **QGraphicsEllipseItem**, **QGraphicsTextItem**, **QGraphicsRectItem**) sunt elementele grafice ce se pot adăuga într-o scena.

Oferă suport pentru: evenimente (mouse/tastatura/context menu), grupare de elemente grafice, drag and drop, detectare de coliziuni

```
QGraphicsView* view = new QGraphicsView;
QGraphicsScene* scene = new QGraphicsScene;
view->setScene(scene);
view->setFixedSize(800, 600);
scene->setSceneRect(0, 0, 800, 600);

//add items to scene
QGraphicsEllipseItem* ball = new QGraphicsEllipseItem{ 0,0,20,20 };
ball->setPos(scene->width() / 2, scene->height()/2 );
scene->addItem(ball);

QGraphicsRectItem* r = new QGraphicsRectItem{ 0,0,40,30 };
r->setPos(20, scene->height() / 2);
r->setBrush(QBrush(QImage("wood1.jpg")));
scene->addItem(r);

view->show();
```

Graphics View Framework



```
class BrickGame:public QGraphicsView {
    QGraphicsScene* scene;
    Paddle* player;
    Ball* ball;
public:
    BrickGame() {
        setMouseTracking(true);
        initScene();
        createPlayer();
        loadLevel();
        addBall();
        startGame();
    }
    void startGame() {
        QTimer* timer = new QTimer;
        //advanceGame invoked every time
        QObject::connect(timer, &QTimer::timeout,
                          this,&BrickGame::advanceGame);
        //generate timeout signal every 20ms
        timer->start(20);
    }
    //handle mouse move
    void mouseMoveEvent(QMouseEvent* ev) override{
        //works only if setMouseTracking(true);
        auto x = ev->pos().x();
        player->setPos(x, player->y());
    }
}

class Ball :public QGraphicsEllipseItem {
private:
    QGraphicsDropShadowEffect * effect;
    int dx = 5;
    int dy = -5;
public:
    Ball() {
        setRect(0, 0, 20, 20);
        setBrush(QBrush(QImage("ball.jpg")));
        effect = new QGraphicsDropShadowEffect();
        effect->setBlurRadius(30);
        setGraphicsEffect(effect);
    }

    void move() {
        setPos(x() + dx, y() + dy);
        setTransformOriginPoint(rect().width() / 2,
                               rect().height() / 2);
        setRotation(rotation() + 45);
    }

    void changeXDir() {
        dx *= -1;
    }
}
```

Curs 11

- **Interfețe grafice utilizator**
 - **Şablonul observer**
 - **Prezentare de volume mari de date**
 - **Model View Controller**

Curs 9-10 – Interfețe grafice utilizator

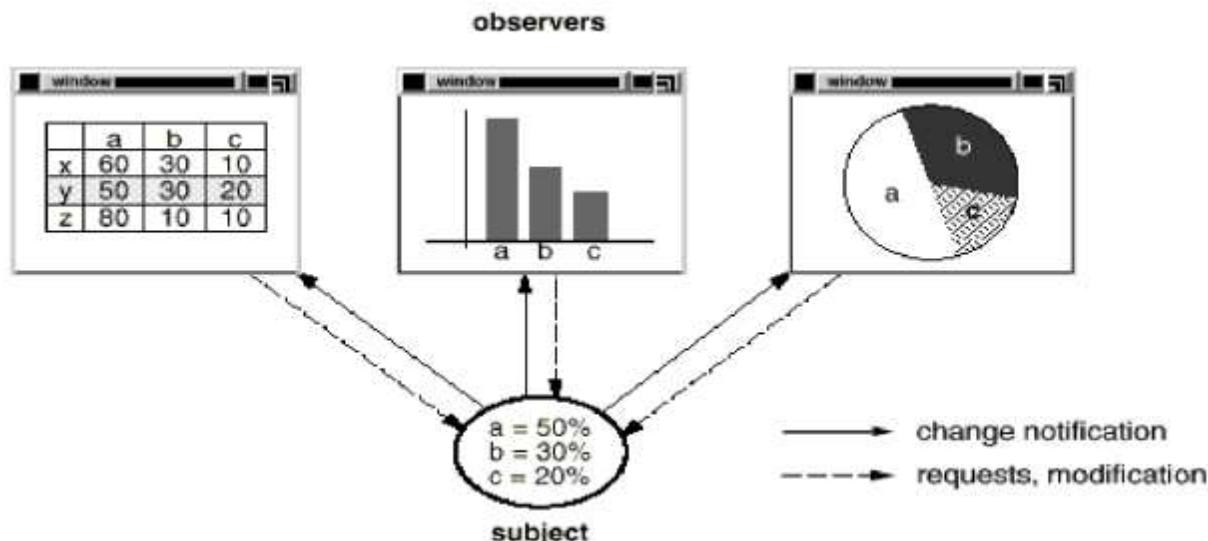
- **Semnale și sloturi**
- **Componente definite de utilizator**
- **Callback/Observer**
- **Evenimente de mouse/tastatura**
- **Graphics View Framework**

Sablonul Observer (Observer Design pattern)

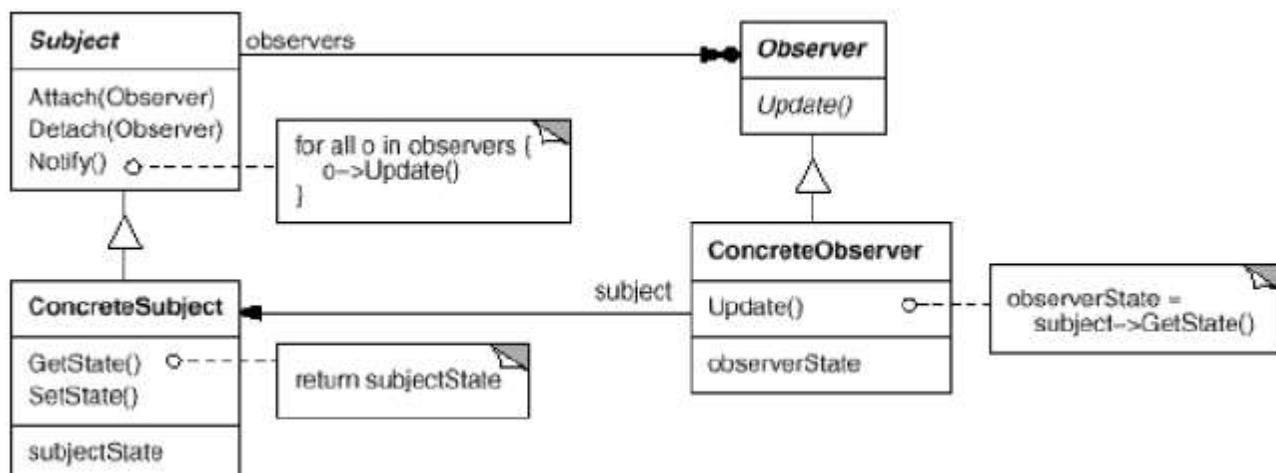
Intent : Definește o relație de dependență one-to-many între obiecte astfel încât în momentul în care obiectul schimba starea toate obiectele dependente sunt notificate automat

Also Known As: Publish-Subscribe*

Motivation: O consecință a partitării sistemului în clase care cooperează este ca apare nevoie de a menține consistență între obiecte. Scopul este să menținem consistență dar în același timp să evităm cuplarea între obiecte (cuplarea reduce reutilizabilitatea).



Pattern class structure



Observer – cod c++

```
/* Update method needs to be implemented by observers
   Alternative names: Listener */
class Observer {
public:
    /* Invoked when Observable change
       Alternative names:propertyChanged           */
    virtual void update() = 0;
};

/* Derive from this class if you want to provide notifications
   Alternative names: Subject, ChangeNotifier */
class Observable {
private:
    /*Non owning pointers to observer objects*/
    std::vector<Observer*> observers;
public:
    /* Observers use this method to register for notification
       Alternative names: attach, register, subscribe, addListener */
    void addObserver(Observer *obs) {
        observers.push_back(obs);
    }
    /* Observers use this to cancel the registration
       !!!Before an observer is destroyed need to cancel the registration
       Alternative names: detach, unregister, removeListener */
    void removeObserver(Observer *obs) {
        observers.erase(std::remove(begin(observers), end(observers), obs),
                        observers.end());
    }
protected:
    /* Invoked by the observable object
       in order to notify interested observer */
    void notify() {
        for (auto obs : observers) {
            obs->update();
        }
    }
};

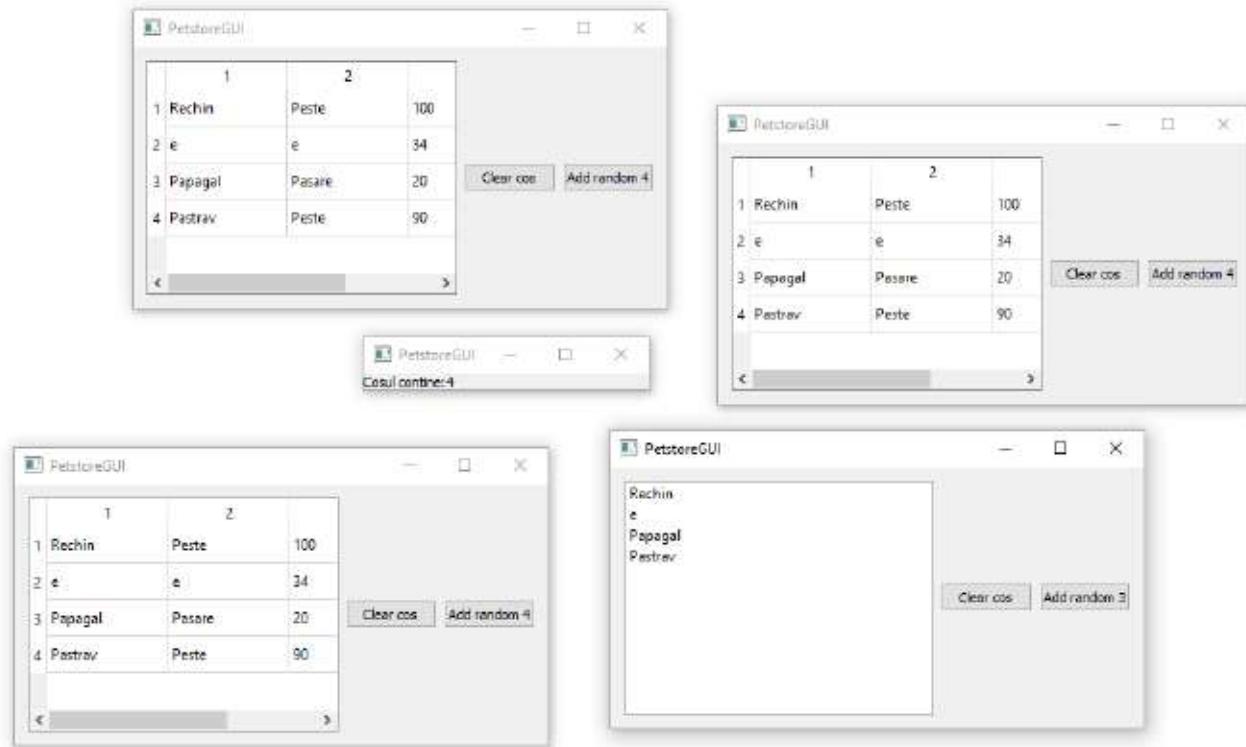
class InterestedObj : public Observer {
public:
    void update() override{
        std::cout << "Notified\n";
    }
};

class ConcreteSubject: public Observable {
public:
    void doStuff() {
        //...
        notify(); //inherited
    }
};
```

Exemplu: cos de cumpărături

Problema: sa avem multiple ferestre (de diferite tipuri) care prezinta același cos cu animale. Fiecare fereastra trebuie actualizata in momentul in care se schimba conținutul coșului.

Scop: gestiunea dependentelor – sa decuplăm clasele intre ele



Clasa Cos extinde **Observable** – conține lista de Pet din cos, notifica orice modificare in lista

Clasele CosTableGUI, CosListGUI, CosLabelGUI - extind Observer si se înscrui pentru notificare (addObserver)

Folosind şablonul Observer obţinem:

- Clasa Cos nu este dependent de clasele GUI
- Clasele GUI nu depind una de cealaltă
- Se pot adăuga cu ușurință noi clase GUI

Şablonul Observer – variante

Pull vs Push

<pre>class Observer { public: virtual void update() = 0; };</pre>	<pre>class Observer { public: virtual void update(int changedState)=0; };</pre>
<p>Pull: cel care este notificat trebuie să obțină datele.</p> <p>In general obiectul interesat are referință la subiect (așa obține informațiile dorite)</p>	<p>Push: cel care este notificat primește informații despre ce s-a schimbat.</p> <p>Obiectul interesat nu are nevoie de referință la subiect.</p> <p>Există și varianța în care se primește subiectul ca parametru la update</p>

Notificări diferite în funcție de ce s-a schimbat în obiectul subiect

<pre>class Observer { public: virtual void itemAdded()=0; virtual void itemRemoved()=0; virtual void itemUpdated()=0; };</pre>	<pre>class Observer { public: virtual void itemAdded(Item item)=0; virtual void itemRemoved(Item item)=0; virtual void itemUpdated(Item item)=0; };</pre>
<p>Obiectul interesat trebuie să implementeze toate metodele pure.</p> <p>Poate reacționa diferit în funcție de ce schimbări au apărut</p>	

Observer Varianta Qt – semnale, sloturi/lambda, QObject::connect, emit

<pre>addObserver -> QObject::connect removeObserver -> QObject::disconnect</pre>	<p>Elimina nevoia de a implementa o anume interfață (extinde Observer, Observable)</p> <p>Funcționează atât cu varianta Pull cat și Push (daca un semnal trimite și valori acestea se vor primi ca și parametru la slot sau lambda care s-a conectat la semnal)</p> <p>Sursa semnalului și slotul (codul care se executa când apare semnalul) sunt total independente</p> <p>Se bazează pe generare de cod C++ (moc compiler)</p>
--	---

Publisher/Subscriber vs Observer

In cartea: Design Patterns: Elements of Reusable Object-Oriented Software (1994) scris de Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides la prezentarea sablonului Observer se mentioneaza: Publish Subscribe

In zilele noastre in general denumirile Observer si Publish/Subscribe se refera la doua concepte similare dar nu echivalente.

Ambele şabloane rezolva acelaşi problema dar:

- In cazul Observer, Subiectul menține o lista de Observer (cei care sunt interesați de modificările apărute in Subiect)
- In cazul Publish / Subscribe: Publisher (Subiectul) nu are nici o dependență față de Subscriber (Observer)
- In cazul Publish/Subscribe apare un nou element numit: Broker, Message Broker, Event Bus. Publisher nu trimită direct mesaj la Subscriber, mesajul este trimis la Broker și Broker livrează mesaje către lista de Subscriber

Publish/Subscribe – Este folosita in general in cazul in care dorim notificări intre procese (nu in același aplicație).

Clasele Qt ItemView

QListWidget, QTableWidget , QTreeWidget

Componentele se populează, adăugând toate elementele de la început (items: QListWidgetItem, QTableWidgetItem, QTreeWidgetItem).

Afișarea, căutarea, editarea sunt efectuate direct asupra datelor cu care este populat componența

Datele care se modifică trebuie sincronizate, actualizat sursa de unde au fost încărcate (fișier, bază de date, rețea, etc)

Avantaje:

- simplu de înțeles
- simplu de folosit

Dezavantaje:

- nu poate fi folosit dacă avem volume mari de date
- este greu de lucrat cu multiple vederi asupra aceluiași date
- necesită duplicare de date

Model-View-Controller

Abordare flexibilă pentru vizualizare de volume mari de date

model: reprezintă setul de date responsabil cu:

- încarcă datele necesare pentru vizualizare
- scrie modificările înapoi la sursă
- folosește şablonul Observer pentru a decupla modelul de View

view: prezintă datele utilizatorului.

- Chiar dacă avem un volum mare de date, doar o porțiune mică este vizibilă la un moment dat. View este responsabil să ceară doar datele care sunt necesare pentru vizualizare (nu toate datele)

controller: mediază între model și view

- transformă acțiunile utilizator în cereri (de navigare, de editare date)
- diferit de GRASP Controller

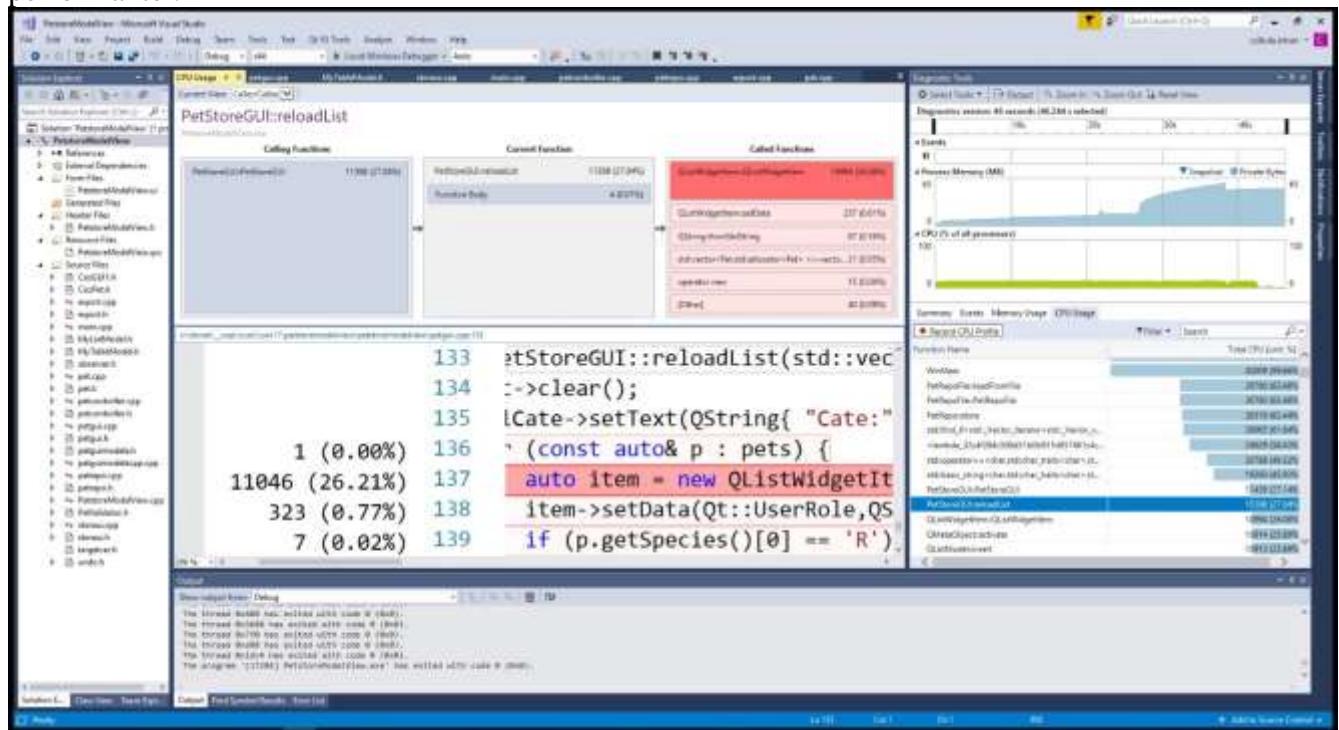
Model/View în Qt

- Separarea datelor de prezentare (views)
- permite vizualizarea de volume mari de date, date complexe , are integrat lucrul cu baze de date , vederi multiple asupra datelor
- Qt 4 > oferă un set de clase model/view (list, table, tree)
- Arhitectura Model/View din Qt este inspirat din şablonul MVC (Model-View-Controller), dar în loc de controller, Qt foloseşte o altă abstractizare numită **delegate**
- **delegate** – oferă control asupra modului de prezentare a datelor și asupra editării
- Qt oferă implementări default pentru delegate pentru toate tipurile de vederi (listă, tabel, tree,etc.) - în general este suficient
- Qt Item Views : QListView, QTableView, QTreeView și clase model asociate

Analiza performantei – Profiling

Instrumentele de tip Profiler facilitează analiza performantei sistemului (consum de memorie, timp de execuție).

In cazul in care avem probleme de performanță (anumite operații merg prea încet) putem folosi un profiler pentru a obține detalii despre care parte a aplicației contribuie la degradarea performanței.



Obs:

- Înainte de a optimiza codul să ne asigurăm că este nevoie de optimizare.
- Regula de aur pentru îmbunătățirea performanței este măsurarea. Fără a măsura nu putem fi siguri că am îmbunătățit ceva.
- În cazul C++ important să compilăm cu opțiunile de optimizare activate (release build, -O3) – nu folosiți debug build pentru a măsura timpul de execuție.

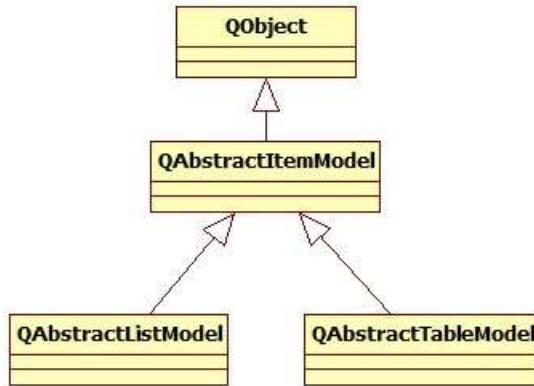
CPU Profiling: putem vedea care metoda și cu cat contribuie la timpul de execuție total.

Memory profiling: putem urmări numarul de obiecte create/distruse, metodele responsabile de crearea de obiecte, putem detecta memory leak și identifica cauza

Profiler-ul folosește **sampling** (verifică din când în când care metoda/instrucțiune este executată) sau **instrumentare** (adaugă cod special în aplicație pentru măsurători)

Creare de modele noi

- Se creează o nouă clasă pentru model (model de listă, model de tabel)
- se extinde o clasă existentă din Qt



QAbstractItemModel – clasă model pentru orice clasă Qt Item View .

Poate conține orice fel de date tabelare (row, columns) sau ierarhice (structură de tree)
Datele sunt expuse ca și un tree unde nodurile sunt tabele

Fiecare item are atașat un număr de elemente cu roluri diferite (DisplayRole, BackgroundRole, UserRole, etc)

Creare de modele noi

```
class MyTableModel: public QAbstractTableModel {
public:
    MyTableModel(QObject *parent);
    /**
     * number of rows
     */
    int rowCount(const QModelIndex &parent = QModelIndex()) const override;
    /**
     * number of columns
     */
    int columnCount(const QModelIndex &parent = QModelIndex()) const override;
    /**
     * Value at a given position
     */
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const override;
};

MyTableModel::MyTableModel(QObject *parent) :
    QAbstractTableModel(parent) {}

int MyTableModel::rowCount(const QModelIndex & /*parent*/) const {
    return 100;
}

int MyTableModel::columnCount(const QModelIndex & /*parent*/) const {
    return 2;
}

QVariant MyTableModel::data(const QModelIndex &index, int role) const {
    if (role == Qt::DisplayRole) {
        return QString("Row%1, Column%2").arg(index.row() + 1).arg(
            index.column() + 1);
    }
    return QVariant();
}
```

Puteți crea modele care încarcă doar datele care sunt efectiv necesare (sunt vizibile)

Modele predefinite

Qt oferă modele predefinite:

- **QStringListModel** – Lucrează cu o listă de stringuri
- **QStandardItemModel** - Date ierarhice
- **QFileSystemModel** - Sistem de fișiere (**QDirModel** în qt 5.x)
- **QSqlQueryModel** - SQL result set
- **QSqlTableModel** - SQL table
- **QSqlRelationalTableModel** - SQL table cu chei străine
- **QSortFilterProxyModel** - oferă sortare/filtrare

```
void createTree() {  
    QTreeView *tV = new QTreeView;  
    QFileSystemModel *model = new QFileSystemModel;  
    model->setRootPath(QDir::currentPath());  
    tV->setModel(model);  
    tV->show();  
}
```

Modificare atribute legate de prezentarea datelor

enum Qt::ItemDataRole	Meaning	Type
Qt::DisplayRole	text	QString
Qt::FontRole	font	QFont
Qt::BackgroundRole	brush for the background of the cell	QBrush
Qt::TextAlignmentRole	text alignment	enum Qt::AlignmentFlag
Qt::CheckStateRole	suppresses checkboxes with QVariant(), sets checkboxes with Qt::Checked or Qt::Unchecked	enum Qt::ItemDataRole

```
QVariant MyTableModel::data(const QModelIndex &index, int role) const {
    int row = index.row();
    int column = index.column();
    if (role == Qt::DisplayRole) {
        return QString("Row%1, Column%2").arg(row + 1).arg(column + 1);
    }
    if (role == Qt::FontRole) {
        QFont f;
        f.setItalic(row % 4 == 1);
        f.setBold(row % 2 == 1);
        return f;
    }
    if (role == Qt::BackgroundRole) {
        if (column == 1 && row % 2 == 0) {
            QBrush bg(Qt::red);
            return bg;
        }
    }
    return QVariant();
}
```

Selectie in QListView/QTableView

```
lst->setModel(model); //Obs. Inainte de connect trebuie setat modelul
QObject::connect(lst->selectionModel(), &QItemSelectionModel::selectionChanged, [&]() {
    btnAddToCos->setEnabled(!lst->selectionModel()->selectedIndexes().isEmpty());
});

QObject::connect(lst->selectionModel(), &QItemSelectionModel::selectionChanged, [&]() {
    if (lst->selectionModel()->selectedIndexes().isEmpty()) {
        //nu este nimic selectat (golesc detaliile)
        return;
    }
    auto selIndex = lst->selectionModel()->selectedIndexes().at(0);
    //putem lua date din lista
    QString species = selIndex.data(Qt::DisplayRole).toString();
    QString type = selIndex.data(Qt::UserRole).toString();
});

//selectia in tabel
QObject::connect(tblV->selectionModel(), &QItemSelectionModel::selectionChanged, [this]() {
    if (tblV->selectionModel()->selectedIndexes().isEmpty()) {
        txtSpecies->setText("");
        return;
    }
    int selRow = tblV->selectionModel()->selectedIndexes().at(0).row();
    auto cel0Index = tblV->model()->index(selRow, 0);
    auto cel1Index = tblV->model()->index(selRow, 1);
    auto cellValue= tblV->model()->data(cel0Index, Qt::DisplayRole).toString();
    txtSpecies->setText(cellValue);
});
```

Cap de tabel (Table headers)

- Modelul controlează și capul de tabel (header de coloane, rânduri) pentru tabel
- Suprascriem metoda `QVariant headerData(int section, Qt::Orientation orientation, int role)`

```
QVariant MyTableModel::headerData(int section, Qt::Orientation orientation,
                                  int role) const {
    if (role == Qt::DisplayRole) {
        if (orientation == Qt::Horizontal) {
            return QString("col %1").arg(section);
        }else {
            return QString("row %1").arg(section);
        }
    }
    return QVariant();
}
```

Sincronizare model și prezentare

Dacă se schimbă datele (modelul) trebuie să se schimbe și prezentarea (view)

View este conectat (automat, în metoda view.setModel) la semnalul **dataChanged**.

Dacă se schimbă ceva în model trebuie să emitem semnalul dataChanged și se actualizează interfața grafică

```
/**  
 * If we change the content of the table. Ex. reorder pets  
 */  
void setPets(const vector<Pet>& pets) {  
    this->pets = pets;  
    QModelIndex topLeft = createIndex(0, 0);  
    QModelIndex bottomRight = createIndex(rowCount(), columnCount());  
    emit dataChanged(topLeft, bottomRight);  
}
```

Vederi multiple pentru același date

Putem avea multiple vederi asupra acelorași date, astfel permitând diferite tipuri de interacțiuni cu data

Folosind mecanismul de semnale și sloturi modificările în model se vor reflecta în toate vederile asociate

```
QTableView* tV = new QTableView();
MyTableModel *model = new MyTableModel(tV);
tV->setModel(model);
tV->show();

QListView *tVT = new QListView();
tVT->setModel(model);
tVT->show();
```

Editare/modificare valori

Se suprascrie metodele:

```
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value, int role)
Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/)

/*
 * Invoked on edit
 */
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value,
    int role) {
    if (role == Qt::EditRole) {
        int row = index.row();
        int column = index.column();
        //save value from editor to member m_gridData
        m_gridData[index.row()][index.column()] = value.toString();
        //make sure the dataChange signal is emitted so all the views will be
notified
        QModelIndex topLeft = createIndex(row, column);
        emit dataChanged(topLeft, topLeft);
    }
    return true;
}

Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/) const {
    return Qt::ItemIsSelectable | Qt::ItemIsEditable | Qt::ItemisEnabled;
}
```

Când schimb modelul trebuie să emitem semnalul dataChanged (să ne asigurăm că vederile se actualizează)

QtDesigner

Project Qt in Visual Studio

- generează structura proiectului qt (setează modulele incluse, directoare , etc)
- .ui – fișier ce conține descrierea interfeței grafice
 - UIC (user interface compiler) utilitar ce transformă fișierul .ui în fișier c++ care construiește interfața grafică (ui_<name>.h)
- crează o componentă GUI component, o clasă (.h, .cpp) - extinde QWidget sau altă clasă derivată din QWidget (QDialog, QMainWindow). Aici putem adăuga sloturi și semnale noi
- main.cpp

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ProductRep w;
    w.show();
    return a.exec();
}
```

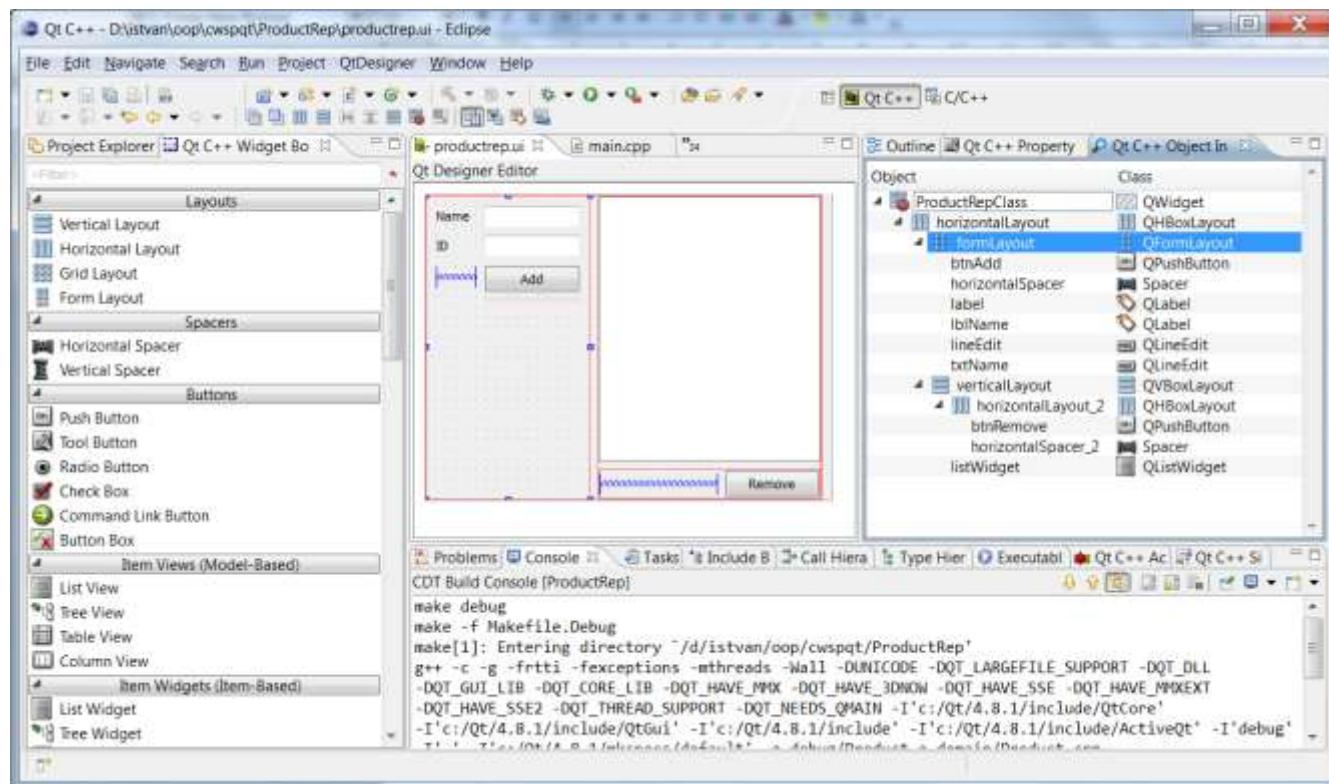
Creare de interfețe grafice vizual (folosind drag & drop)

Qt Designer permite creare de interfețe grafice în mod vizual (fără să scriem cod)

- nu este neobișnuit pentru un programator Qt să creeze aplicații exclusiv scriind cod
 - dar, varianta vizuală poate fi mai rapidă în anumite situații
 - permite experimentarea rapidă cu diferite variante de interfață grafică

Qt editor/views:

- Qt Designer editor – permite crearea de GUI (aranjare componente grafice)
 - Qt C++ Widget Box - expune componente Qt care se pot adăuga pe fereastră
 - Qt C++ Object Inspector – prezintă organizarea componentelor (componente fii)
 - Qt C++ Property Editor – editare de proprietăți pentru componentele adăugate pe fereastră



Curs 12

- **Organizarea interfețelor utilizator**
 - Separate Presentation
 - Observer – push/pull
 - diagrame UML de secvență
- **Şabloane de proiectare**
 - adapter
 - strategy
 - composite

Separate presentation

Un şablon arhitectural – o forma de structurare a aplicaţiilor

Ideea – Separarea codului legate de prezentare (presentation code) de logica aplicaţiei (domain code)

Presentation Code:

- La aplicaţii cu interfeţe grafice GUI: manipulează componente grafice, aranjează componentele
- La aplicaţii Web: partea de HTML si manipulerea headerelor HTTP
- La aplicaţii tip consola: prelucrează argumente din linia de comanda, citeşte comenzi de la tastatura, tipăreşte informaţii

Aplicaţia este separată în două părţi, module logice: partea de interfaţă cu utilizatorul și restul aplicaţiei (restul aplicaţiei în general la rândul lui este structurat pe straturi: Service Layer, Business logic Layer, Persistence Layer)

Straturile (**layers**) în general sunt doar straturi logice, controlează dependentele (stratul de sus depinde de stratul imediat următor, nici un strat nu depinde de un strat superior)

Straturile pot fi separate și fizic (**tiers** – pe mai multe calculatoare sau procese)

Stratul de prezentare poate apela stratul de domeniu (depinde de domeniu) dar startul de domeniu nu accesează niciodată startul de prezentare

Obiectele din stratul de domeniu pot folosi şablonul **Observer** pentru a notifica stratul prezentare de schimbările apărute.

Şabloane de proiectare

- Şabloanele de proiectare descriu obiecte, clase și interacțiuni/relații între ele. Un şablon reprezintă o soluție comună a unei probleme într-un anumit context
- Sunt soluții generale, reutilizabile pentru probleme ce apar frecvent într-un context dat
- Christopher Alexander: "Fiecare şablon descrie o problemă care apare mereu în domeniul nostru de activitate și indică esența soluției acelei probleme într-un mod care permite utilizarea soluției de nenumărate ori în contexte diferite"
- Design Patterns: Elements of Reusable Object-Oriented Software – 1994
- Gang of Four (GoF)- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- Introduce şabloanele de proiectare și oferă un catalog de şabloane

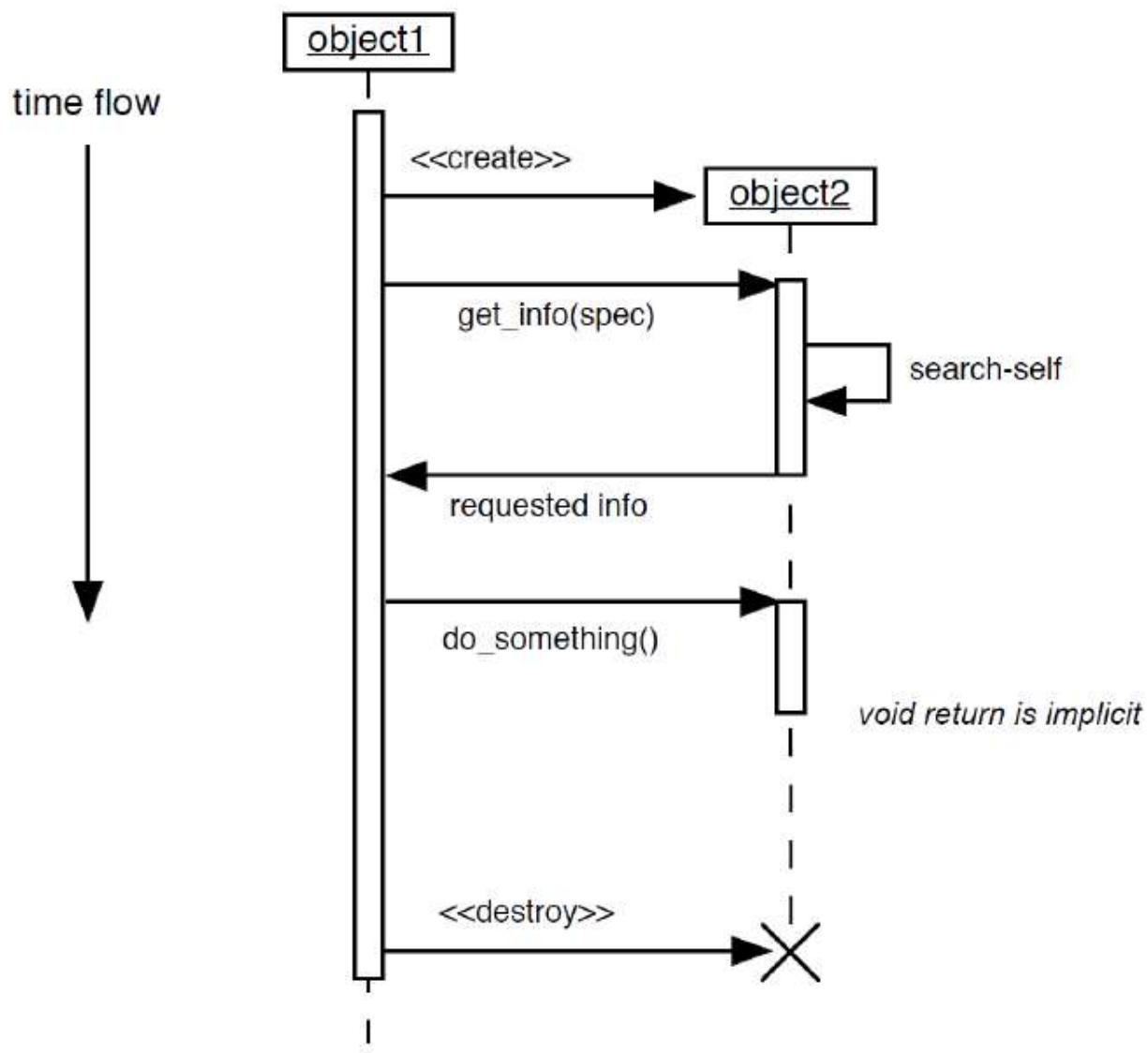
Tipuri de şabloane de proiectare (după scop):

- **Creaționale**
 - descriu modul de creare a obiectelor
 - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **Structurale**
 - se refere la compoziția claselor sau a obiectelor
 - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- **Comportamentale**
 - descriu modul în care obiectele și clasele interacționează și modul în care distribuim responsabilitățile
 - Chain of responsibility, Command Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Elemente ce descriu un şablon de proiectare

- Numele şablonului
 - descrie sintetic problema rezolvată și soluția
 - face parte din vocabularul programatorului
- Problema
 - Descrie problema și contextul în care putem aplica şablonul.
- Soluția
 - Descrie elementele soluției, relațiile între ele, responsabilitățile și modul de colaborare
 - Oferă o descriere abstractă a problemei de rezolvat, descrie modul de aranjare a elementelor (clase, obiecte) din soluție
- Consecințe
 - descrie consecințe, compromisuri legat de aplicarea şablonului de proiectare.

Diagrame UML de secvență (interacțiune)



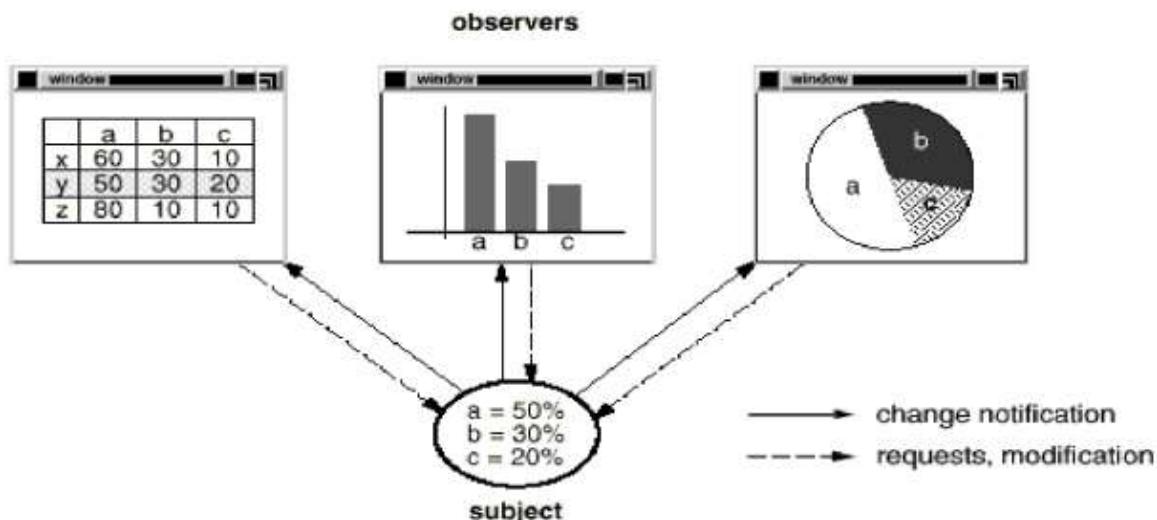
Ilustrează interacțiunea între obiecte

Sablonul Observer (Observer Design pattern)

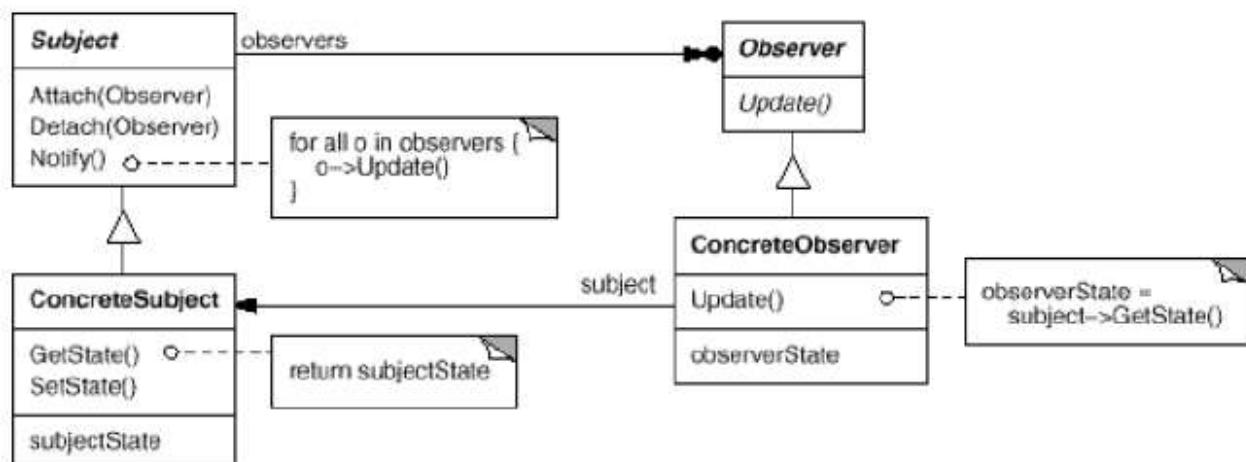
Intent : Defineste o relatie de dependenta one-to-many intre obiecte astfel incat in momentul in care obiectul schimba starea toate obiectele dependente sunt notificate automat

Also Known As: Publish-Subscribe

Motivation: O consecinta a partitionarii sistemului in clase care coopereaza este ca apare nevoia de a mentine consistenta intre obiecte. Scopul este sa meninem consistenta dar in acelasi timp sa evitam cuplarea intre obiecte (cuplarea reduce reutilizabilitatea).



Pattern class structure

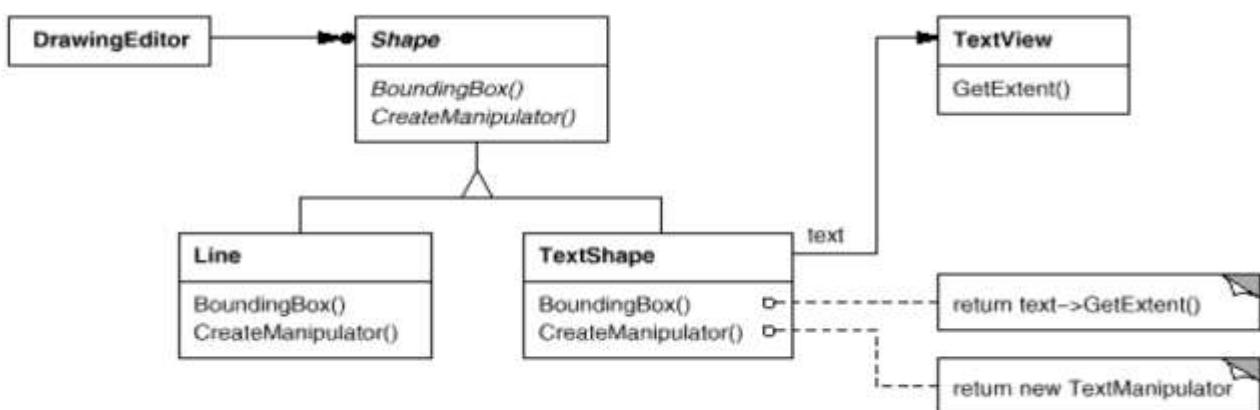


Adapter pattern (Wrapper)

Intenția: Adaptarea interfeței unei clase la o interfață potrivită pentru client. Permite claselor să inter-opereze, clase care fără convertirea interfeței nu ar putea conlucra.

Motivație: În unele cazuri avem clase din biblioteci externe care ar fi potrivite ca și funcționalitate dar nu le putem folosi pentru că este nevoie de o interfață specifică în codul existent în aplicație.

Ex. Draw Editor (Shape: lines, polygons, etc) Add TextShape. Soluția este să adaptăm clasa existentă TextView class. TexShape adaptează clasa TextView la interfața Shape

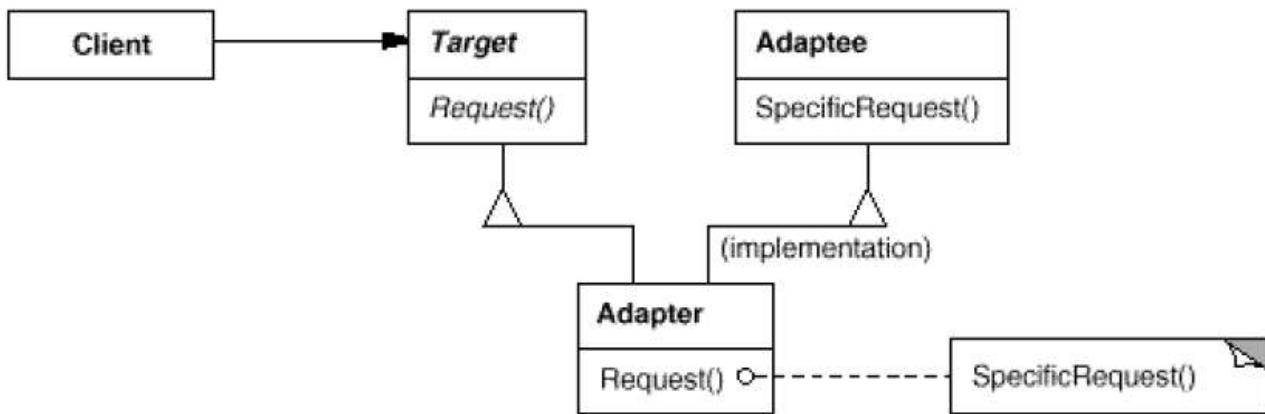


Aplicabilitate:

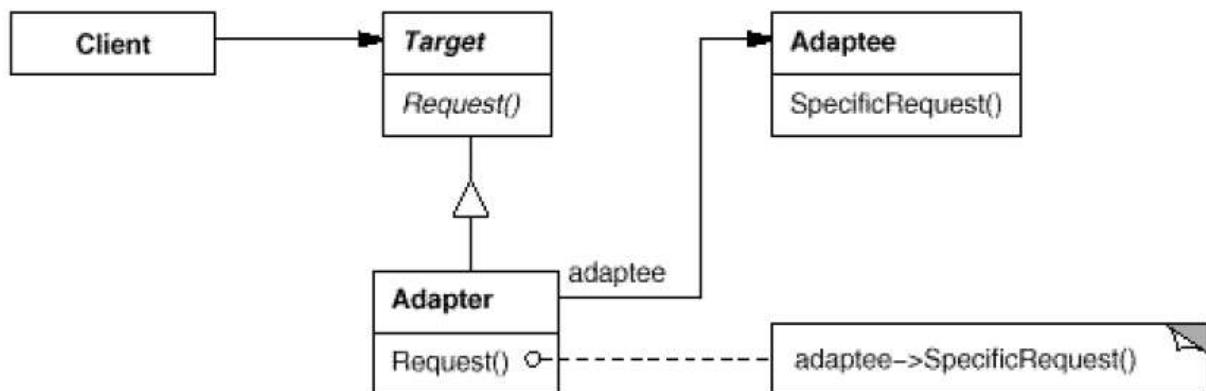
- dorim să folosim o clasă existentă dar interfața clasei nu corespunde cu ceea ce este nevoie
- creare de clase reutilizabile care cooperează cu alte clase (dar ele nu au interfețe compatibile)

Adapter - structură

Class adapter – folosește moștenire multiplă



Object adapter folosește compozиie



Participants:

- Target: definește interfața de care este nevoie.
- Client: colaborează, folosește obiecte cu interfață Target.
- Adaptee: este clasa care trebuie adaptată. Are interfață diferită de ceea ce e are nevoie Client
- Adapter: adaptează Adaptee la interfața Target.

Adapter

Colaborare:

- Clientul apelează metode ale lui Adapter. Clasa adapter folosește metode de la clasa Adaptee pentru a efectua operația dorită de Client.

Consecințe:

Class adapter:

- Nu putem folosi dacă dorim să adaptăm clasa și toate clasele derivate
- Permite clasei Adapter să suprascrie anumite metode a clasei Adaptee
- Introduce un singur obiect nou în sistem. Metodele din adapter apelează direct metode din Adaptee

Object adapter:

- este posibil ca un singur Adapter să folosească mai multe obiecte Adaptees.
- Este mai dificil să suprascriem metode din Adaptee (Trebuie să creăm o clasa derivată din Adaptee și să folosim această clasă derivată în clasa Adapter)

Adapter folosit în STL: Container adapters, Iterator adapters

Adaptor de containere (Container adaptors)

Sunt containere care încapsulează un container de tip secvență, și folosesc acest obiect pentru a oferi funcționalități specifice containerului (stivă, coadă, coadă cu priorități).

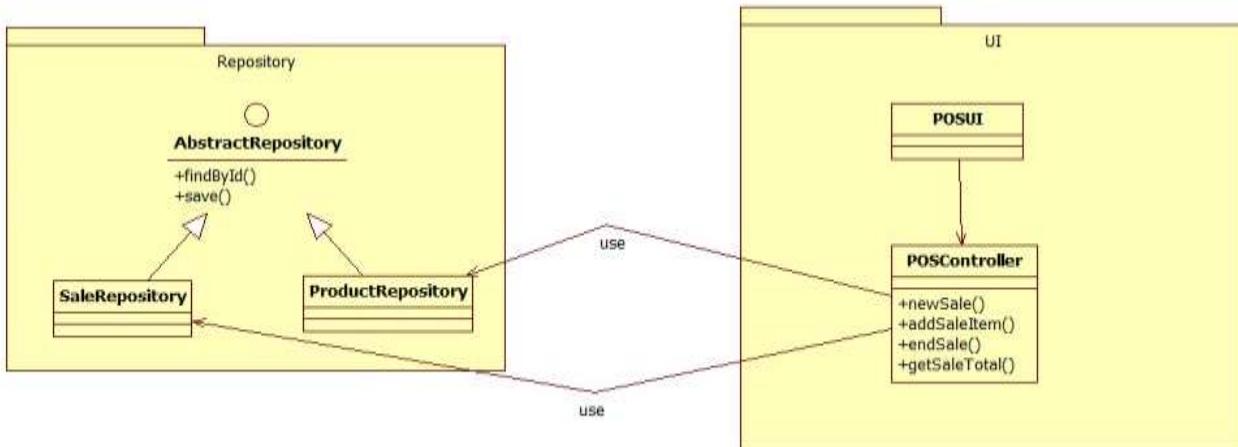
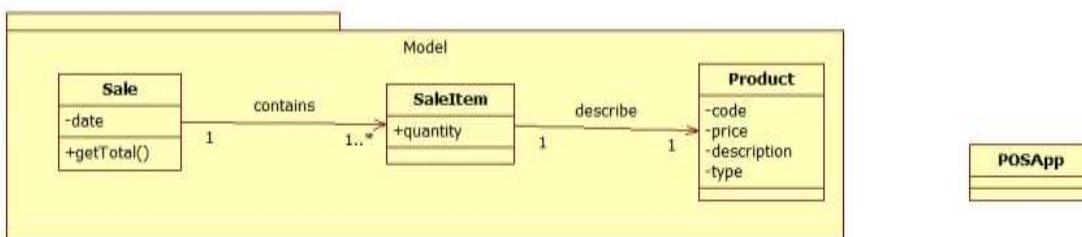
STL folosește şablonul adapter pentru: Stack, Queue, Priority Queue. Aceste clase au un template parameter de tip container de secvență, dar oferă doar operații permise pe stivă, coadă, coadă cu priorități (Stack, Queue, Priority Queue)

- Stack: strategia LIFO (last in first out) pentru adaugare/ștergere elemente
 - Elemente sunt adăugate/extrase la un capăt (din vârful stivei)
 - Operații: empty(), push(), pop(), top()
 - `template < class T, class Container = deque<T> >`
`class stack;`
 - T: tipul elementelor
 - Container: tipul containerului folosit pentru a stoca elementele din stivă
- queue: strategia FIFO (first in first out)
 - Elementele sunt adăugate (pushed) la un capăt și extrase (popped) din capătul celălalt
 - operații: empty(), front(), back(), push(), pop(), size();
 - `template < class T, class Container = deque<T> >`
`class queue;`
- priority_queue: se extrag elemente pe baza priorităților
 - operations: empty(), top(), push(), pop(), size();
 - `template < class T, class Container = vector<T>,`
`class Compare = less<typename`
`Container::value_type> >`
`class priority_queue;`

Adaptor de containere - exemple

<pre>#include <stack> void sampleStack() { stack<int> s; //stack<int,deque<int>> s; //stack<int,list<int>> s; //stack<int,vector<int>> s; s.push(3); s.push(4); s.push(1); s.push(2); while (!s.empty()) { cout<<s.top()<< " "; s.pop(); } }</pre>	<pre>#include <queue> void sampleQueue() { //queue<int> s; //queue<int,deque<int>> s; queue<int>,list<int> > s; s.push(3); s.push(4); s.push(1); s.push(2); while (!s.empty()) { cout << s.front() << " "; s.pop(); } }</pre>	<pre>#include <queue> void samplePriorQueue() { //priority_queue<int> s; //priority_queue<int,deque<int>> s; //priority_queue<int,list<int>> s; priority_queue<int,vector<int>> s; s.push(3); s.push(4); s.push(1); s.push(2); while (!s.empty()) { cout << s.top() << " "; s.pop(); } }</pre>
---	---	---

Aplicația POS (Point of service)



```
/*
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
    double total = 0;
    for (int i = 0; i < items.size(); i++) {
        SaleItem sIt = items[i];
        double price = sIt.getQuantity() * sIt.getProduct().getPrice();
        total += price;
    }
    return total;
}

void testSale() {
    Sale s;
    assert(s.getTotal()==0);

    Product p1(1, "Apple", "food", 2.0);
    s.addItem(3, p1);
    assert(s.getTotal()==6);

    Product p2(1, "TV", "electronics", 2000.0);
    s.addItem(1, p2);
    assert(s.getTotal()==2006);
}
```

Aplicația POS

Cerințe:

- 2% reducere dacă plata se face cu cardul
- Dacă se cumpără 3 bucăți sau mai multe din același produs se dă o reducere de 10%
- Luni se acordă o reducere de 5% pentru mâncare
- Reducere - Frequent buyer
- ...

```
/**  
 * Compute the total price for this sale  
 * isCard true if the payment is by credit card  
 * return the total for the items in the sale  
 */  
double Sale::getTotal(bool isCard) {  
    double total = 0;  
    for (int i = 0; i < items.size(); i++) {  
        SaleItem sIt = items[i];  
        double pPrice;  
        if (isCard) {  
            //2% discount  
            pPrice = sIt.getProduct().getPrice();  
            pPrice = pPrice - pPrice * 0.02;  
        } else {  
            pPrice = sIt.getProduct().getPrice();  
        }  
        double price = sIt.getQuantity() * pPrice;  
        total += price;  
    }  
    return total;  
}  
void testSale() {  
    Sale s;  
    assert(s.getTotal(false)==0);  
  
    Product p1(1, "Apple", "food", 2.0);  
    s.addItem(3, p1);  
  
    assert(s.getTotal(false)==6);  
  
    Product p2(1, "TV", "electronics", 2000.0);  
    s.addItem(1, p2);  
  
    assert(s.getTotal(false)==2006);  
  
    //total with discount for cars  
    assert(s.getTotal(true)==1965.88);  
}
```

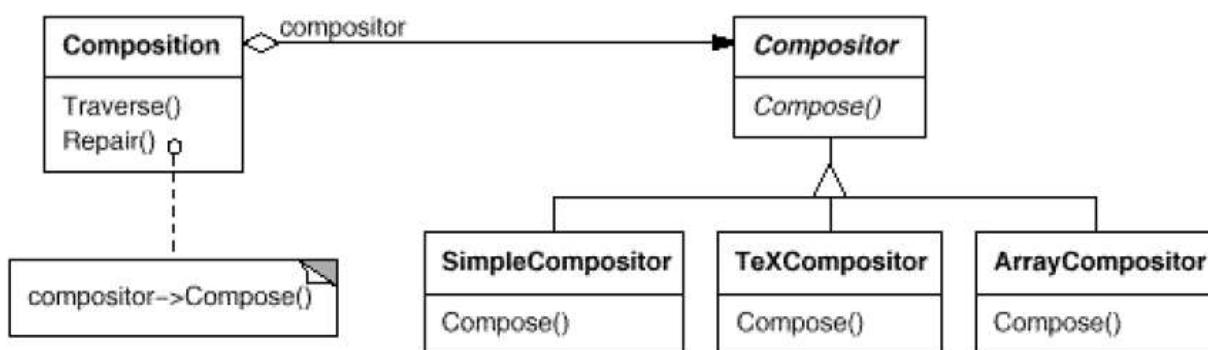
Această abordare conduce la cod complicat, calcule care sunt greu de urmărit. Cod greu de întreținut, extins, înțeles.

Şablonul de proiectare Strategy (policy)

Scop: Defineşte modul de implementare a unor familii interschimbabile de algoritmi.

Motivare:

Aplicaţia de editor de documente, are o clasă **Composition** responsabil cu menţinerea şi actualizarea aranjării textului (line-breaks). Există diferiţi algoritmi pentru formatarea textului pe linii. În funcţie de context se folosesc diferiţi algoritmi de formatare.



Fiecare strategie de formatare este implementat separat în clase derivate din clasa abstractă **Compositor** (nu **Composition**).

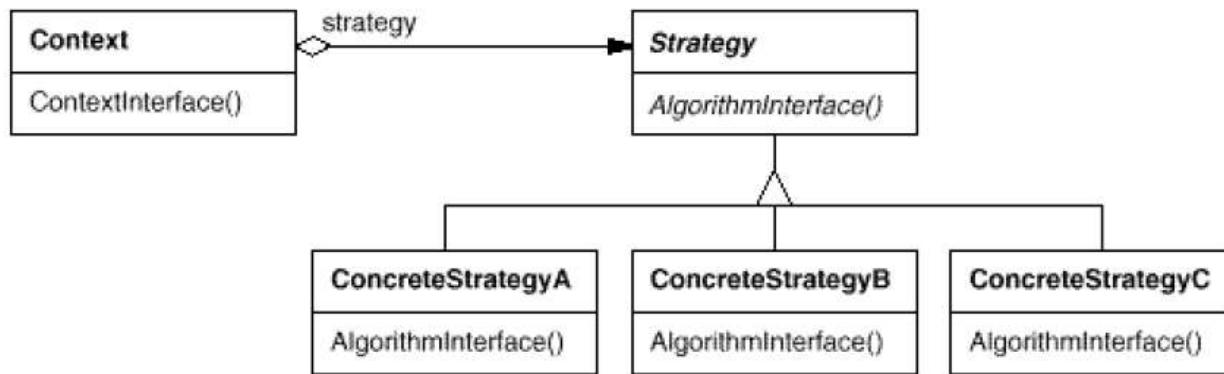
Clasele derivate din **Compositor** implementează strategii:

- **SimpleCompositor** implements strategie simplă, adaugă linie nouă una câte una.
- **TeXCompositor** implementează algoritmul TeX pentru a identifica poziția unde se adaugă linie nouă (identifică liniile global, analizând tot paragraful).
- **ArrayCompositor** formatează astfel încât pe fiecare linie există același număr de elemente (cuvinte, icoane, etc).

Strategy (Policy)

Aplicabilitate:

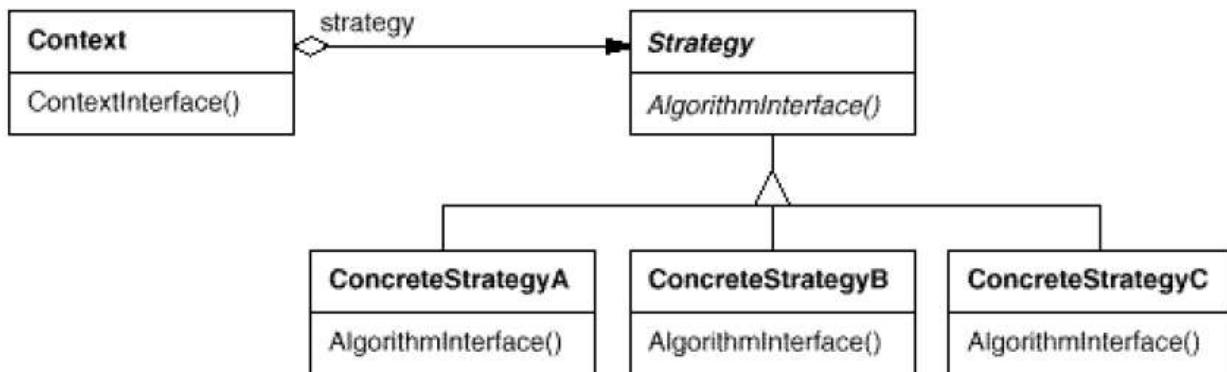
- mai multe clase sunt similare, există diferențe ca și comportament. Şablonul Strategy oferă o metodă de a configura comportamentul.
- Este nevoie de mai multe variante de algoritmi pentru o problemă.
- Un algoritm folosește date despre care clientul nu ar trebui să știe. Se poate folosi şablonul Strategy pentru a nu expune date complexe specifice algoritmului folosit.
- Avem o clasă care folosește multiple clauze if/else (sau switch) pentru a implementa o operație. Corpurile if/else, se pot transforma în clase separate și aplicat şablonul Strategy .



Participanți:

- **Strategy** (Composer): definește interfața comună pentru toți algoritmii. Context folosește această interfață pentru a apela efectiv algoritmul definit de clasa ConcreteStrategy.
- **ConcreteStrategy** (SimpleComposer, TeXComposer, ArrayComposer) implementează algoritmul.
- **Context** (Composition)
 - este configurat folosind un obiect ConcreteStrategy
 - are referință la un obiect Strategy .
 - Poate defini o interfață care permite claselor Strategy să acceseze datele membre.

Strategy



Colaborare:

- Strategy și Context interacționează pentru a implementa algoritmul ales. Context oferă toate datele necesare pentru algoritm. Alternativ, se poate transmite ca parametru chiar obiectul context când se apelează algoritmul.
- Clasa context delegă cereri de la clienți la clasele care implementează algoritmii. În general Client creează un obiect ConcreteStrategy și transmite la Context;
- Clientul interacționează doar cu context. În general există multiple versiuni de ConcreteStrategy din care clientul poate alege.

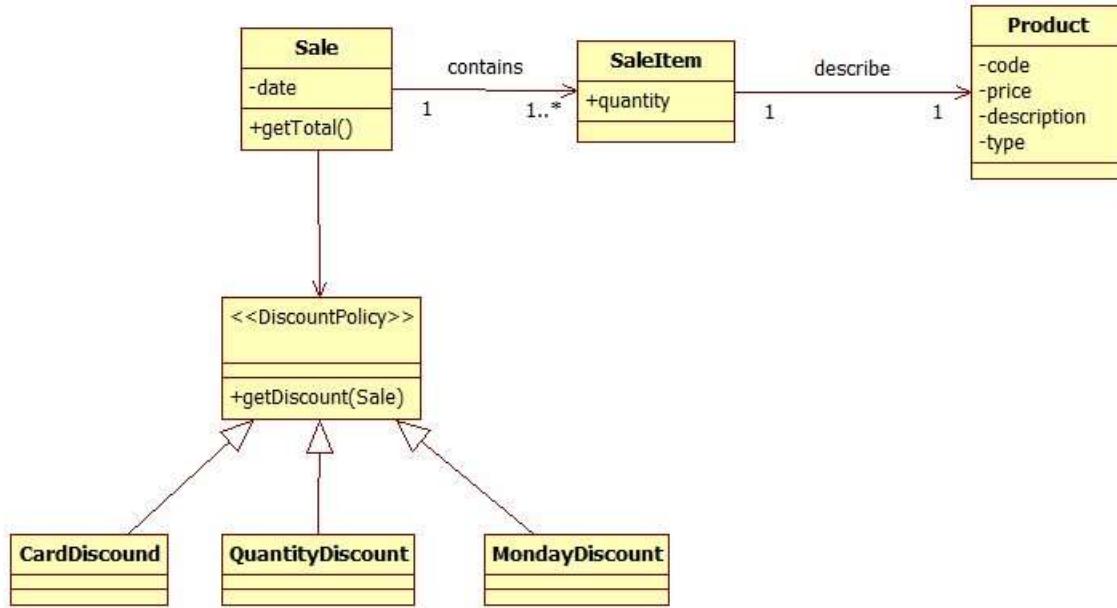
Consecințe:

- Familie de algoritmi se pot defini ca și o ierarhie de clase. Moștenirea poate ajuta să extragem părți comune.
- Se elimină if-else și switch. Şablonul Strategy poate fi o alternativă la logica condițională complicată.
- Clientul trebuie să lucreze, să cunoască faptul că există multiple variante de Strategii
- Comunicarea între Strategy și Context poate degrada performanța (se fac apeluri de metode în plus)
- Număr mare de obiecte în aplicație.

Discount Policy pentru POS

Extragem partea care variază (reducerea) în procesul (de calculare a totalului) în clase "strategy" separate.

Separăm regula de procesul de calcul al totalului, implementăm regulile conform şablonului de proiectare strategy.



Controlăm comportamentul metodei `getTotal` folosind diferite obiecte **DiscountPolicy**.

Este ușor să adăugăm reduceri noi.

Logica legată de reducere este izolat (Protected variation GRASP pattern).

Discount Policy pentru POS

```
class DiscountPolicy {
public:
    /**
     * Compute the discount for the sale item
     * s - the sale, some discount may based on all the products in te sale, or other
     attributes of the sale
     * si - the discount amount is computed for this sale item
     * return the discount amount
    */
    virtual double getDiscount(const Sale* s, SaleItem si)=0;
    virtual ~DiscountPolicy() {}

};

/**
 * Apply 2% discount
*/
class CreditCardDiscount: public DiscountPolicy {
public:
    virtual double getDiscount(const Sale* s, SaleItem si) override {
        return si.getQuantity() * si.getProduct().getPrice() * 0.02;
    }
};

/**
 * Compute the total price for this sale
 * return the total for the items in the sale
*/
double Sale::getTotal() {
    double total = 0;
    for (int i = 0; i < items.size(); i++) {
        SaleItem sIt = items[i];
        double price = sIt.getQuantity() * sIt.getProduct().getPrice();
        //apply discount
        price -= discountPolicy->getDiscount(this, sIt);
        total += price;
    }
    return total;
}

void testSale() {
    Sale s(new NoDiscount());
    Product p1(1, "Apple", "food", 2.0);
    Product p2(1, "TV", "electronics", 2000.0);
    s.addItem(3, p1);
    s.addItem(1, p2);
    assert(s.getTotal()==2006);

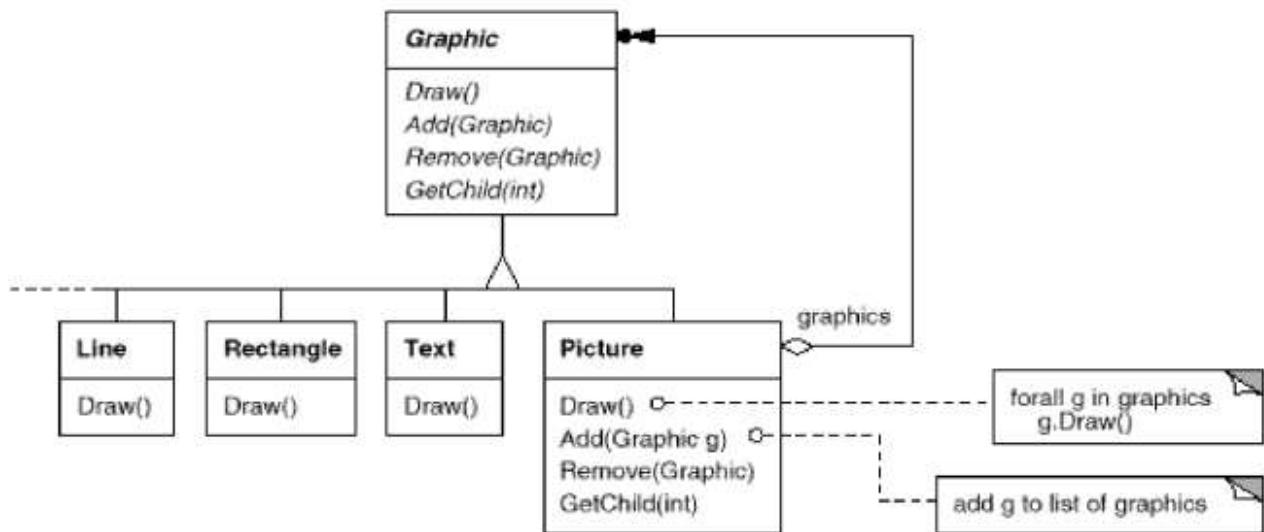
    Sale s2(new CreditCardDiscount());
    s2.addItem(3, p1);
    s2.addItem(1, p2);
    //total with discount for card
    assert(s2.getTotal()==1965.88);
}
```

Cum combinăm reducerile?

Şablonul de proiectare Composite

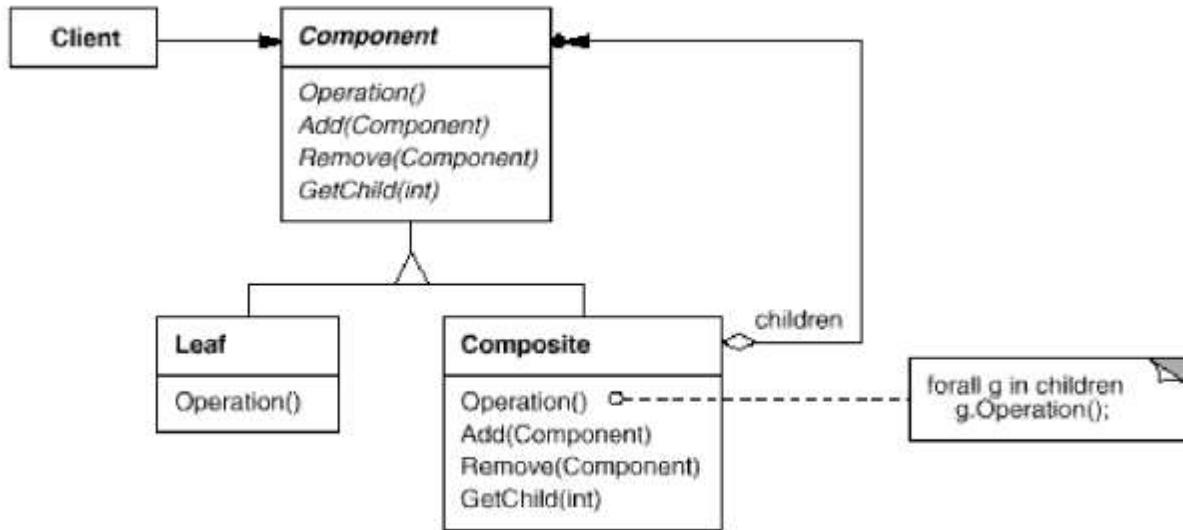
Scop: Permite compunere de obiecte într-o structură arborescentă. Clienții pot trata uniform atât obiectele individuale cat și grupuri de obiecte

Motivare: Într-o aplicație de desenat, utilizatorul poate crea obiecte simple (linii, pătrate, cercuri) și poate crea structuri mai complicate folosind obiecte grafice simple (grupează obiecte simple)



Elementul principal al şablonului Composite este clasa abstractă **Graphic**, care reprezintă atât obiecte simple cat și obiecte care sunt de fapt grupuri de obiecte simple. Acest design permite tratarea tuturor obiectelor (simple, compuse) uniform în aplicație.

Composite



Participanti:

Component:

- definește interfața obiectelor, poate oferi implementare default pentru diferite operații
- definește metode pentru a accesa elemente din interiorul compoziției

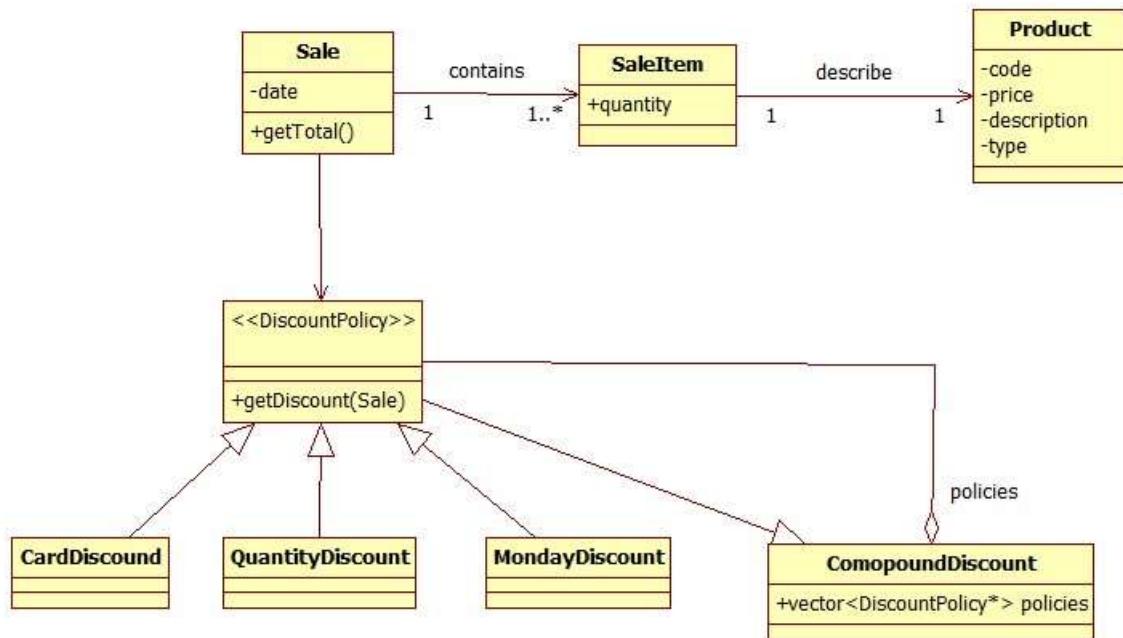
Leaf:

- reprezintă obiectele simple (frunze) din compoziție,
- definește comportamentul obiectului

Composite

- definește comportamentul componentelor compuse
- stochează componentele din care e format
- implementează operații legate de manipularea componentelor din interior

POS – Mai multe reduceri care se aplică



```

/**
 * Combine multiple discount types
 * The discounts will sum up
 */
class CompoundDiscount: public DiscountPolicy {
public:
    virtual double getDiscount(const Sale* s, SaleItem si) override;

    void addPolicy(DiscountPolicy* p) {
        policies.push_back(p);
    }
private:
    vector<DiscountPolicy*> policies;
};

/**
 * Compute the sum of all discounts
 */
double CompoundDiscount::getDiscount(const Sale* s, SaleItem si) {
    double discount = 0;
    for (int i = 0; i < policies.size(); i++) {
        discount += policies[i]->getDiscount(s, si);
    }
    return discount;
}

```

POS – Reduceri combinate

```
Sale s(new NoDiscount());
Product p1(1, "Apple", "food", 10.0);
Product p2(2, "TV", "electronics", 2000.0);
s.addItem(3, p1);
s.addItem(1, p2);
assert(s.getTotal()==2030);

CompoundDiscount* cD = new CompoundDiscount();
cD->addPolicy(new CreditCardDiscount());
cD->addPolicy(new QuantityDiscount());

Sale s2(cD);
s2.addItem(3, p1);
s2.addItem(4, p2);
//total with discount for card
assert(s2.getTotal()==7066.4);
```

Cum putem exprima reguli de genul:

Reducerea “Frequent buyer” și reducerea de luni pe mâncare nu poate fi combinată, se aplică doar una dintre ele (reducerea mai mare)

Curs 13

- Modern C++ (C++ 11/14)
- Resource management – RAII
- Thread
- ASM - optimizari

Curs 12 - Organizarea interfețelor utilizator

- Separate Presentation
- Observer – push/pull
- diagrame UML de secvență
- Șabloane de proiectare: adapter, strategy; composite

New features in C++ 11/14

Evoluția C++

- versiune curentă C++17 (înainte: C++14, C++11, C++98, C++ with classes, C)
- finalizare specificații C++ 20
- isocpp.com - website C++ ISO standard

C++11/14/17 design goals:

Make simple things simple, make hard things possible

More abstractions without sacrificing efficiency

C++ is "expert friendly" But is not just a language for "experts"

Elementele noi ar trebui să faciliteze un stil mai bun de programare.

Însă orice facilitate se poate folosi greșit (Any feature can be overused, there are many "dark corners" in the language

Majoritatea compilatoarelor implementează funcționalitățile noi din limbaj, din standardul C++ curent și oferă standard library (STL) conform specificațiilor din standard

- gcc (mingw)
- clang
- microsoft compiler

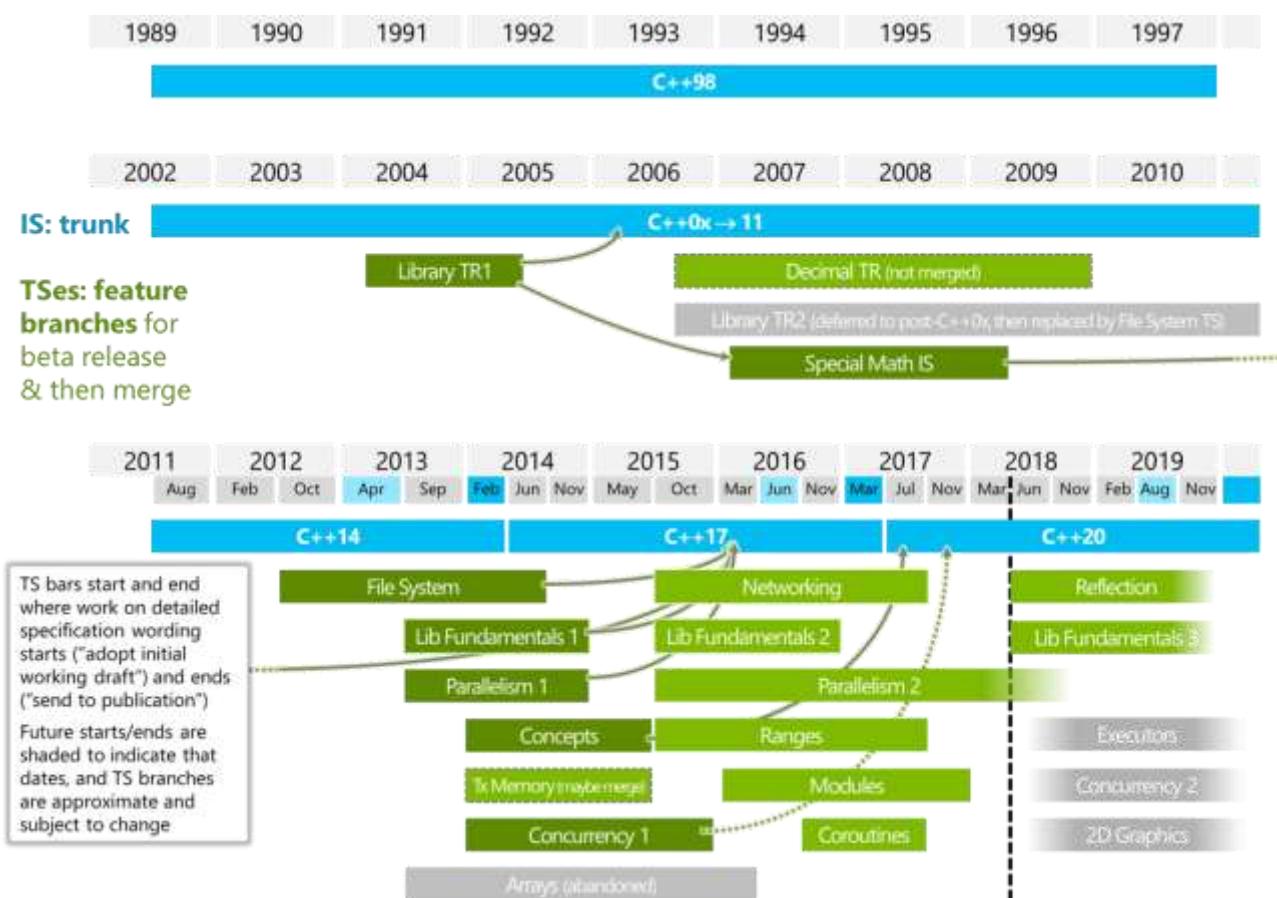
Bjarne Stroustrup:

- www.stroustrup.com/
- Talks: C++11 Style, The essence of c++, etc

Herb Sutter

- Talks: One C++, Back to basics, Why C++ , etc

Evolutie C++



Noutăți in c++ 17

Noutăți in biblioteca standard:

std::variant, std::optional, std::any

std::string_view, std::filesystem, std::byte

variante cu execuție paralele pentru algoritmi (copy, find, sort)

Facilitați noi în limbaj:

structured bindings, constexpr if, folding expressions

Noutăți in c++ 20

Noutăți in biblioteca standard:

concepts library, std::span

Facilitați noi în limbaj:

module – creare de cod izolat în module, compilare separată, separare interfață/implementare

ranges – descrie secvențe de elemente

coroutine – funcții care permit suspendarea/reîncadrarea

concepts - pentru template metaprograming, predicate verificate la compilare

Evolutia codului C++ (exemplu)

```
C (old C++ code)
int cmpInt(const void *a, const void *b) {
    int aa = *(int *) a;
    int bb = *(int *) b;
    return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}
void sortC() {
    const int size = 3;
    int array[size];
    array[0]=2; array[1]=1; array[2]=0;
    qsort(array, size, sizeof(int), cmpInt);
    int i;
    for (i=0;i<size;i++){
        printf("%d ",array[i]);
    }
}
```

```
C++98
bool cmpF(int a, int b) {
    return b < a;
}
void sortC98() {
    vector<int> v;
    v.push_back(3); v.push_back(2); v.push_back(1);
    sort(v.begin(), v.end(), cmpF);
    for (vector<int>::iterator it = v.begin(); it!=v.end(); ++it)
    {
        cout << (*it) << endl;
    }
}
```

```
C++14
void sortC14() {
    vector<int> v { 3, 2, 1 };
    sort(v.begin(), v.end(), [] (int a, int b) {
        return b < a;
    });
    for (auto elem : v) {
        cout << elem << endl;
    }
}
```

C++11: Mai rapid, mai general, type safe – make simple things simple

Uniform initialization

- Sintaxa comună pentru inițializare de obiecte în C++11.
- Permite inițializarea ușoara pentru std::vector or std::map (sau orice container) la un set de valori.

```
vector<int> v { 3, 2, 1 }           v.push_back(3);  
                                      v.push_back(2);  
                                      v.push_back(1);  
  
struct SampleStruct {  
    int a, b;  
};  
SampleStruct s { 1, 2 };  
SampleStruct sarray[] = { { 1, 2 }, { 2, 5 }, { 3, 4 } };  
  
class MyClass {  
public:  
    MyClass(int a);  
....  
MyClass obj{2};  
MyClass array[] = {{1},{7}};
```

- Implicit type narrowing

```
class MyClass {  
public:  
    MyClass(int a);  
....  
MyClass obj { 2 }; //fine  
MyClass obj3 { 2.3 }; //error:  
narrowing conversion of '2.3'
```

```
class MyClass {  
public:  
    MyClass(int a);  
...  
MyClass obj2(2.3); //works  
silently convert to 2 (maybe a  
compile warning)
```

- resolve "most vexing type"

```
class MyClass {  
public:  
    MyClass();  
....  
MyClass obj{};  
obj.f(); //fine
```

```
class MyClass {  
public:  
    MyClass();  
....  
MyClass obj(); //is this a variable?  
obj.f(); //compile error... is of  
non-class type 'MyClass()'
```

- Minimize redundant typenames

```
MyClass f() {  
    return {3};  
}
```

```
MyClass typeNameRedundancy() {  
    return MyClass(3);  
}
```

Uniform initialization mechanics - initializer lists in clasele proprii

Exista un nou tip de date: std::initializer_list , acesta poate fi folosit ca si orice colectie

Pentru a suporta initializer list in clasele proprii:

- se definește un constructor care primește ca parametru std::initializer_list
- apoi se poate inițializa folosind același sintaxă ca și la vector de ex.

```
class MyClassWithUI{
public:
    MyClassWithUI(initializer_list<int> l){
        for_each(l.begin(),l.end(),[&](int a){
            elems.push_back(a);
        });
    }
    vector<int> getElems(){
        return elems;
    }
private:
    vector<int> elems;
};
```

Obs. Constructorul cu `initializer_list` are prioritate (fata de alți constructori definiți în clasa)

```
vector<int> v2 { 10 };
cout<<v2.size()<<endl;//print 1, 1 element (10) in the vector

vector<int> v3(10);
cout<<v3.size()<<endl;//print 10, elems: 0,0,0,0,0,0,0,0,0,0
```

Auto

Tipul variabilei se deduce automat de compilator pe baza expresiei care inițializează variabila .

Poate fi folosit si pentru tipul de return al unei funcții (se deduce din expresia de la return) (since C++14).

```
 MyClass someFunction() {
    return MyClass(3);
}
auto someFct()->int{
    return 7;
}
void sampleAuto() {
    int a = 6;
    auto b = 6; //b is int
    auto c = someFunction(); //c is MyClass, the return type
    auto d = someFct(); //d is int
}
```

Permite scriere de cod generic (in general la templaturi dar nu numai)

```
void doThings(vector<int> v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}

void doThingsA(vector<int> v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}
```

Schimbam metoda doThings – transmitem prin referință sa evitam copierea, punem const fiindcă nu schimbam v

```
void doThings(const vector<int>& v) {
    for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}

void doThingsA(vector<int> v) {
    for (auto it = v.begin(); it != v.end(); ++it) {
        cout << (*it);
    }
}
```

Functii Lambda

Functii anonte care sunt capabile sa captureze variabile din domeniul exterior de vizibilitate

```
sort(v.begin(), v.end(), [](int a, int b) {
    return b<a;
});

for_each(v.begin(), v.end(),[](int a) {cout<<a;});
```

Compilatorul creeaza functorul in locul nostru. Practic creeaza o clasa care defineste operatorul ()

Variabile vizibile in interiorul unei lambda:

```
vector<Person> all{{3, "Ion"},{5, "Pop"}};
int someId=5;
auto p =find_if(all.begin(),all.end(),[&](const Person& p){
    return p.id==someId;
});
if (p!=all.end()){
    return *p
}
```

Sintaxa generală

[capture-list](params)->ret{body}

capture-list – specify variables that are visible inside the *body*

capture-list examples:

- [] captures nothing
- [&] captures all by reference
- [=] captures all by value
- [a,&b] capture a by value, b by reference

ret – return type of the lambda

if omitted: if the body is just a return statement - ret is the type of the returned expression else is void

Resource management – RAII

Orice resursa ar trebui sa fie proprietatea unui "handler":

- încapsularea resursei (memorie, fișier) este o abstractizare folositoare(vector, string, iostream, file, thread,etc)
- Handler – este o clasa cu responsabilitatea de a crea/gestiona/elibera resursa
- In constructor se obține/initializează resursa iar în destructor se eliberează resursa

```
class Vector{  
public:  
    Vector(initializer_list<int> l); //acquire memory  
    ~Vector(); //release memory  
};  
  
int fct(){  
    vector<int> v{1,2,3,4};  
    ...  
    //end of scope, destructor automatically invoked (memory released)  
}
```

Orice resursa (memory, file handler, thread, ...) ar trebui să aibă un (scoped) handler

Orice responsabilitate ar trebui încapsulat într-o clasa handler (ex. Desenare obiect geometric, ștergerea de pe ecran)

Handler – trebuie alocat pe stack

- scoped – se dealoca automat când execuția părăsește domeniul de vizibilitate a variabilei

RAII – Resource acquisition is initialization

- achiziție în timpul construcției (constructor)
- eliberare la distrugere (destructor)

Folosind idea de handlers:

- este mult mai ușor să gestionam memoria (fara resource leak)
- ascundem complexitatea (anumite resurse pot fi ne-trivial de alocat/eliberați)
- mult mai ușor să scriem exception safe code (chiar dacă metoda arunca excepție destructorul variabilelor locale se apelează)

De ce folosim pointeri – resursa (memorie) care nu e gestionata de un obiect handler

Guideline: Now raw pointers

```
void pointerNotNeeded(){
    MyClass* c = new MyClass(7);
    //do some computation...
    if (c->getA()<0){
        throw runtime_error("Negative not allowed");//!!leak
    }
    //do some computation...
    if (c->getA()>10){
        return;//!!!leak
    }
    delete c; //only if we are lucky
}

void noPointer(){
    MyClass c{7};
    //do computation
    if (c.getA()<0){
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (c.getA()>10){
        return;
    }
}
```

Exista motive legitime de a folosi pointeri (pointerul nu e un lucru rău):

- este o abstractizare buna a memoriei din calculator
- poate fi folosit in interiorul claselor de tip handler:
 - vector are un pointer la array-ul de elemente
 - pentru a implementa structura de tree sau lista înlăntuita
 - poate fi folosit pentru a reprezenta o poziție
- putem folosi pentru a facilita polimorfismul

Guideline updatat: No owning raw pointers

- nu folosiți pointer pentru obiecte pe care trebuie sa le distrugeți (reprezintă idea de ownership). Pointer este owning daca este responsabil cu memoria referita (trebuie sa eliberezi memoria folosind pointerul)

Owning pointers: Smart pointers

In cazul in care totusi avem nevoie de owning pointer:

- Ex. folosim o functie care returneaza un owning pointer (owning pointer => trebuie sa eliberez memoria dupa ce am folosit obiectul)

Ar trebui sa existe o clasa handler pentru el.

Mai bine folosim una dintre clasele handler existente din libraria standard (**unique_ptr, shared_ptr, weak_ptr** – header `<memory>`)

Smart pointers se comporta ca si un pointer normal (ofera operatiile uzuale de pointer `*, ++, etc`) dar are in plus o functionalitate: gestiune automata a memoriei, range checking, etc.

```
void sampleNastyAPI() {
    MyClass* rez = someAPI();
    //do computation
    if (rez->getA() < 0) {
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (rez->getA() == 0) {
        return;
    }
    delete rez;
}

void handleWithSmartPointers() {
    unique_ptr<MyClass> rez(someAPI());
    //do computation
    if (rez->getA() < 0) {
        throw runtime_error("Negative not allowed");
    }
    //computation
    if (rez->getA() == 0) {
        return;
    }
}
```

`std::unique_ptr` este un smart pointer care preia responsabilitatea de delocare. Cand `unique_ptr` este distrus acesta distruge si obiectul referit (apeleaza destructor).

Acelasi pointer poate fi inglobat doar de o singura instanta de `unique_ptr` (non copyable).

Smart pointers

```
void smartPointerSample() {
    MyClass* rez = someAPI(); //we leak rez
    unique_ptr<MyClass> smartP{someAPI()};

    vector<MyClass*> v;
    //when v is destroyed we leak 3 MyClass objects
    v.push_back(new MyClass());
    v.push_back(new MyClass());
    v.push_back(new MyClass());

    vector<unique_ptr<MyClass>> smartV;
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
    smartV.push_back(unique_ptr<MyClass>(new MyClass()));
}
```

shared_ptr : similar cu unique_ptr (încapsulează un pointer și gestionează dealocarea), dar implementează reference counting:

- pot exista multiple instanțe de **shared_ptr** care referă același pointer, **shared_ptr** tine numărul de instanțe **shared_ptr** existente pentru același pointer (reference count)
- când numărul de referințe ajunge la 0 obiectul referit de pointer este dealocat

Move semantic &&

Why do we have to deal with (owning) pointers

- We are working with legacy code
- We are using a library (some old library)
- Somebody in the project don't know the guideline: No owning raw pointers :)
- Return a pointer from a function to avoid the unnecessary copy of large objects (starting from c++11 there is a better alternative)

Returning large object from a function

class LargeMatrix. We want to implement sum of 2 matrices

Option1 – Problem: who does the delete; No raw pointer

```
LargeMatrix* sum(const LargeMatrix& a,const LargeMatrix&b){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return rez;  
}  
...  
LargeMatrix* rez = sum(a,b);
```

Option2 – Problem: who does the delete; Caller not even know that he may need to delete

```
LargeMatrix& sum2(const LargeMatrix& a,const LargeMatrix&b){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return *rez;  
}  
...  
LargeMatrix& rez = sum(a,b);
```

Option3 – Problem: ugly interface. Function with side effect Consider operator +;

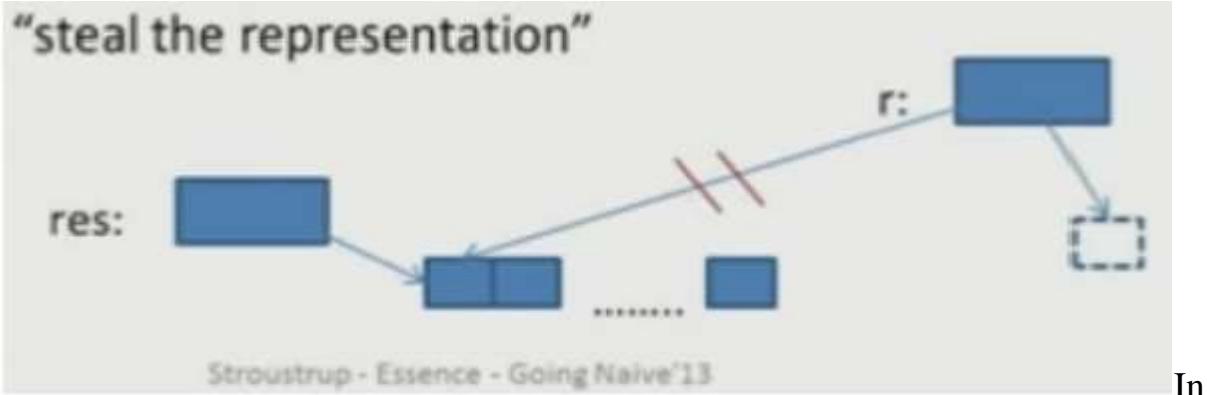
```
void sum2(const LargeMatrix& a,const LargeMatrix&b,){  
    LargeMatrix* rez = new LargeMatrix();  
    //rez=a+b  
    return *rez;  
}  
...  
LargeMatrix rez;  
sum(a,b,rez);
```

Returning large object from a function

Best option: return an handler.

```
LargeMatrix sum3(const LargeMatrix& a, const LargeMatrix& b){  
    LargeMatrix rez;  
    //rez=a+b  
    return rez;  
}  
...  
LargeMatrix res = sum3(a,b);
```

Copy may be expensive (compiler invoke copy constructor)



C++11 a new kind of constructor: move constructor used for move operation
don't copy, steal the representation and leave an empty object behind

```
class LargeMatrix {  
public:  
    LargeMatrix(LargeMatrix&& ot) { //move constructor  
        elems = ot.elems;  
        ot.elems= {};  
    }  
private:  
    int *elems; //large array of numbers  
};
```

C++11 style guidelines

Style

- No naked pointers
 - Keep them inside functions and classes
 - Keep arrays out of interfaces (prefer containers)
 - Pointers are implementation-level artifacts
 - A pointer in a function should not represent ownership
 - Always consider `std::unique_ptr` and sometimes `std::shared_ptr`
- No naked `new` or `delete`
 - They belong in implementations and as arguments to resource handles
- Return objects “by-value” (using move rather than copy)
 - Don’t fiddle with pointer, references, or reference arguments for return values



Stroustrup - C++11 Style - Feb'12

*R*All and move semantics

All the standard library containers provide move semantics:

- vector, list, map, set, unordered_map

You can return a local vector object by value from a function

- no copy just a move operation (2-3 assignments even if the number of elements is very large)

Other standard resource handlers are providing move semantics:

- threads, locks
- istream, ostream
- unique_ptr, shared_ptr

Time

Modulul `<chrono>` din standard library oferă funcționalități de măsurare a timpului (in namespaceul `using namespace std::chrono`).

Tipuri de date de interes in namespaceul chrono:

`std::chrono::system_clock` - the system-wide real time wall clock.

`std::chrono::high_resolution_clock` - the clock with the smallest tick period provided by the implementation

`time_point` - point in time, `duration` – period of time

folosind metoda `duration_cast` putem calcula timpul scurs intre doua `time_point` uri.

```
#include <chrono>
using namespace std::chrono;

void measureElapsedTIme() {
    auto t0 = high_resolution_clock::now();
    fct_weWantToMeasure();
    auto t1 = high_resolution_clock::now();
    auto elapsed = duration_cast<seconds>(t1 - t0);
    cout << elapsed.count() << "sec\n";
}

void fct_weWantToMeasure() {
...
}
```

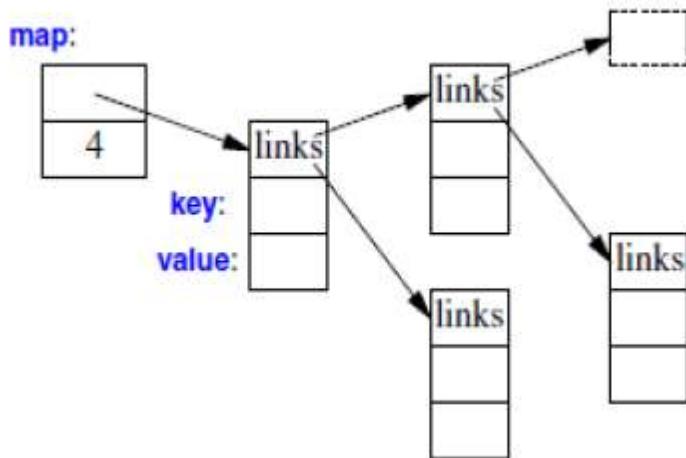
Când vorbim despre performanță, este foarte important să măsurăm. Doar măsurarea poate fi punctul de pornire pentru optimizarea codului.

Exercițiu: îmbunătățire funcție genereazaPet, măsurători pentru diferite variante de implementare

Containere STL – dicționare

std::map

- în headerul <map>
- dicționar implementat cu o structură de arbore de căutare (red/black tree)
- este un container optimizat pentru lookup: $O(\log_2(n))$
- elementele sunt perechi (Pair p.first, p.second)



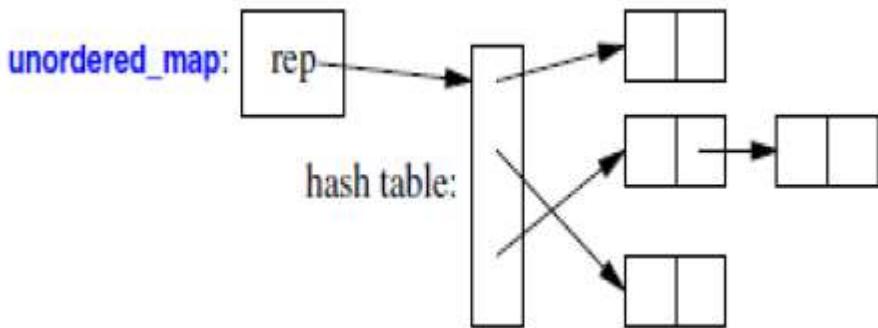
la constructor se poate da funcția de comparare a cheilor (folosit în construirea/parcursarea structurii de tree)

```
void testMaps() {
    std::map<string, int>
phoneBook{{"Ion",1234}, {"Vasile",32432} ,{ "Maria",765432 } };
    //add a new entry
    phoneBook["Iulia"] = 111145;
    //lookup key: iterator find( const Key& key );
    auto val = phoneBook.find("Ion");
    cout << (*val).first << (*val).second << "\n";
    //not found
    if (phoneBook.find("0") == phoneBook.end()) {
        cout << "Not found\n";
    }
    //stergere - prin iterator/prin element/prin range
    phoneBook.erase("Ion");
    if (phoneBook.find("Ion") == phoneBook.end()) {
        cout << "Ion erased\n";
    }
    phoneBook.erase(phoneBook.find("Iulia"));
    phoneBook.erase(phoneBook.find("Iulia"), phoneBook.end());
}
```

Unordered_map

std::unordered_map

- in headerul `<unordered_map>`
- implementat folosind o tabela de dispersie
- optimizat pentru lookup: O(1) daca avem un hashing function ideal
- interfață (metodele publice) este identic cu map



In libraria standard exista definite functii de hashing pentru tipurile predefinite

In cazul in care dorim sa avem ca si chei obiecte de ale noastre (user defined), trebuie sa cream o fuctie de hashing custom

```
struct MyClass {
    string name;
    int phoneNr;
    bool operator==(const MyClass& a) const{
        return name == a.name;
    }
};

struct MyHash {
    //acesta se va folosi pentru hashing
    size_t operator()(const MyClass& m) const {
        return std::hash<string>()(m.name) ^ std::hash<int>()(m.phoneNr);
    }
};

void testHashMaps() {
    std::unordered_map<MyClass, int, MyHash> phoneBook{{{"Ion",1234},1234},
                                                       {{ "Vasile",32432 },32432},
                                                       {{ "Maria",765432 },765432}};
    //transformam intr-o lista
    std::vector<MyClass> l;
    for (auto pair : phoneBook) {
        l.push_back(pair.first);
    }

    for (auto o : l) {
        cout << o.name << "\n";
    }
}
```

Exercițiu: modificat repository de Pet sa folosească map in loc de vector

Fire de execuție – thread

Paralel - execuția simultană a mai multor taskuri (metode)

Concurrent = paralel + acces simultan la același resurse

Este folosit pentru a mari volumul de procesare efectuat, calculatoarele/laptopurile curente au mai multe procesoare, sunt capabile să execute concurrent mai multe taskuri.

Fiecare task se executa in același proces, dar pe un fir de execuție propriu

C++ oferă posibilitatea de a executa taskuri in paralel folosind clasa std::thread din modulul <thread>

```
#include <thread>

void func1(int n) {
    for (int i = 0; i < n; i++) {
        cout << i << "\n";
    }
}

void testThreads() {
    //executam func1 cu parametrul 1000 pe un fir de executie separat
    std::thread t1{func1,100};

    //executam func2 cu parametrul 1000 pe un fir de executie separat
    std::thread t2{ func1,100 };

    //asteptam pana se termina
    t1.join();
    t2.join();
}
```

Exercițiu: La generarea pet să folosim un thread pentru a nu bloca interfața grafică în timp ce se executa generația.

Destructorul de la clasa thread distrugerea threadului (apelează terminate), trebuie apelat metoda detach() de la thread dacă dorim ca firul de execuție să continue chiar și după ce se distrug obiectul thread.

```
QObject::connect(btnGenereaza10000, &QPushButton::clicked, [this]() {
    std::thread t{ [this]{
        this->btnGenereaza10000->setEnabled(false);
        ctr.genereazaAleatorAnimale(1000);
        QMessageBox::information(nullptr, "Info", "Am terminat generarea...");
        this->btnGenereaza10000->setEnabled(true);
    } };
    t.detach();
});
```

Compiler explorer

Oferă posibilitatea de a vedea codul ASM generat de compilator

Se poate experimenta cu diferite variante compilatoare, opțiuni de compilare

<https://godbolt.org/>

The screenshot shows the Compiler Explorer interface with two assembly outputs side-by-side.

Left Window (Editor #1): Contains the C++ source code:1
2 int f(int a , int b, int c){
3 return a+b+c;
4 }
5
6
7 int main(){
8 int b = 9;
9 return f(2,5,b);
10 }

Right Window (Editor #1, Compiler #1): Shows assembly output for x86-64 clang 4.0.0 with -std=c++14. The assembly is:11010 .LX0: .text ## Intel A- C- D- E- F-
1 f(int, int, int):
2 push rbp
3 mov rbp, rsp
4 mov dword ptr [rbp - 4], edi
5 mov dword ptr [rbp - 8], esi
6 mov dword ptr [rbp - 12], edx
7
8 ret
9
10 main: # @main
11 push rbp
12 mov rbp, rsp
13 sub rsp, 16
14 mov edi, 2
15 mov esi, 5
16 mov dword ptr [rbp - 4], 0
17 mov dword ptr [rbp - 8], 9
18 mov edx, dword ptr [rbp - 8]
19 call f(int, int, int)
20 add rsp, 16
21
22
23
24

A tooltip for the 'ret' instruction in 'main' states: "Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction."

Bottom Left Window (Compiler #2): Shows the compiler exited with result code 0.

Bottom Right Window (Editor #2, Compiler #2): Shows assembly output for x86-64 clang 4.0.0 with -std=c++14 -O3. The assembly is:11010 .LX0: .text ## Intel A- C- D- E- F-
1 f(int, int, int):
2 lea eax, [rdi + rsi]
3 add eax, edx
4 ret
5
6 main: # @main
7 mov eax, 16
8 ret

CPP Insights

Poate fi folosit pentru a înțelege mai bine anumite concepții din C++ : conversii implicite, rangefor, funcții lambda, template, type deduction etc.

Afișează codul C++ după ce codul scris de programator trece printr-o transformare inițială de către compilator.

<https://cppinsights.io/>

The screenshot shows the CPP Insights interface. On the left, the 'Source' tab displays the original C++ code:

```
1 #include <array>
2
3 int main()
4 {
5     int sum = 0;
6     std::array v{1,2,3,4,5,1,3};
7     for (auto el:v){
8         sum+=el;
9     }
10
11     auto l = [sum](){return sum;};
12     return 0;
13 }
```

On the right, the 'Insight' tab shows the transformed code, which includes generated code for the lambda expression and range-based for loop:

```
1 #include <array>
2
3 int main()
4 {
5     int sum = 0;
6     std::array<int, 7> v = {{1, 2, 3, 4, 5, 1, 3}};
7     std::array<int, 7> __range1 = v;
8     int * __begin1 = __range1.begin();
9     int * __end1 = __range1.end();
10    for(; __begin1 != __end1; ++__begin1)
11    {
12        int el = *__begin1;
13        sum += el;
14    }
15
16 }
17
18 class __lambda_11_13
19 {
20 public:
21     inline /*constexpr */ int operator()() const
22     {
23         return sum;
24     }
25 }
```

At the bottom, the 'Console' tab shows the output: 'Insights exited with result code: 0'.

Programare orientată obiect

Obiective

- **Cunoașterea și înțelegerea conceptelor specifice programării orientate obiect**
- **Abilități de programare în limbajele de programare C și C++**

Obiectivele specifice:

- Scrierea de programe de scară mică/mijlocie cu interfețe grafice utilizator folosind C++ și QT.
- Proiectarea orientată obiect pentru programe de scară mică/mijlocie
- Explicarea/Înțelegerea structurilor de tip clasă ca fiind componente fundamentale în construirea aplicațiilor.
- Înțelegerea rolului moștenirii, polimorfismului, legării dinamice și a structurilor generice în realizarea codului reutilizabil.
- Utilizarea claselor/modulelor scrise de alții programatori în dezvoltarea sistemelor proprii
- Folosirea de instrumente: debugger, memory leak detector, code coverage, static code analyser, profiler

1 Elemente de bază ale limbajului C.

- Elemente de bază ale limbajului C. Elemente lexicale. Operatori. Conversii.
- Tipuri de date. Variabile. Constante. Domeniul de vizibilitate și durata de viață
- Declararea și definirea funcțiilor.

2-3. Programare modulară în C/C++. Tipuri de date

- Funcții. Parametri.
- Fișiere header. Biblioteci. Implementarea modulară a TAD-urilor.
- Tipuri de date derivate și tipuri definite de utilizator, alocare dinamică în C++.
- Tipuri de date: vectori și structuri, pointeri și referințe.
- Gestiona memoria in C/C++. Pointeri la funcții și pointeri spre void.

4. Programare orientată obiect în C++.

- Clase și obiecte. Membrii unei clase. Modificatori de acces. Constructori/destructori.
- Gestiona memoria in C++ (RAII)
- Implementarea TAD-urilor in C++
- Diagrama UML pentru clase (membri, acces).

5. Elemente de programare generică

- Functii/clase parametrizate. Mecanismul de template din C++
- Implementare TAD-uri folosind clase parametrizate
- Containere și iteratori – biblioteca STL

6-8 Moștenire / Polimorfism / Ierarhii de clase

- Moștenire simplă. Clase derivate. Principiul substituției.
- Supraîncărcarea metodelor. Moștenire multiplă. Relații de specializare/generalizare
- Moștenire, polimorfism
- Stream-uri I/O. Ierarhia de clase I/O. Formatare. Manipulatori. Fișiere text.
- Exceptii, spatii de nume. Fișiere text.

9-11 Interfețe grafice utilizator / Elemente de programare bazată pe evenimente

- QT Toolkit: instalare, instrumente și module Qt. Componente grafice utilizator. Layout management. Proiectare GUI.
- Evenimente: Semnale și sloturi Qt. Callback/Observer
- Componente grafice cu modele. Şablonul MVC.
- Studiu de caz. Detalii comenzi – Produse.

12-13 Şabloane de proiectare.

- Şablonul Observer
- Şabloane de proiectare Façade, Strategy.
- Şablonul de proiectare Composite

Note/Reguli

Calcului notei:

Notă Laborator 30% (media notelor de laborator)

Notă Simulare 10%

Notă examen scris 30%

Notă examen practic 30%

Pentru promovare este nevoie de **minim 5** la fiecare notă în afară de simulare.

Laborator/prezențe

Cei care nu satisfac criteriu cu numărul de prezență la laborator/seminar nu participă la examen (au picat materia).

Cei care nu au nota 5 la laborator nu pot intra în examenul din sesiune. Pot veni la examenul din sesiunea de restanță.

În restanță se predau toate laboratoare (cei care nu au luat minim 5 în timpul semestrului). Nota maxima pentru nota pe activitatea de laborator este 5 în acest caz.

Restanță

În sesiunea de restanță se poate da examenul scris, examenul practic sau ambele. Valabil atât pentru cei care au picat examenul (examenele) cât și pentru cei care vin la mărire de notă.

Reguli / desfăşurare

Sa aveți la voi un act de identitate (carnet de student, buletin, pașaport)

Sa aveți toate taxele plătite la zi (daca este cazul)

Se da examenul scris apoi examen practic cu o pauza intre ele

Sa va prezentați la examen înainte cu 15 minute de ora stabilita.

Nu copiați. Daca observați orice tentativa de fraudare / nereguli sa le semnalati in timpul examenului.

Sugestii

Veniți odihniți la examen.

Aveți la voi o sticla de apa pentru hidratare.

Sa mâncați ceva înainte de examen sau in pauza intre scris si practic.

Nu va stresați excesiv. Tot timpul este o alta șansa sa mai dai examenul. Nu se termina lumea cu un examen ratat.

Examenul scris

Examenul scris se da în timpul sesiunii.

Durata aproximativ 1.5

Tipuri de probleme:

- sintaxă C++ , algoritmi
 - Se dă o funcție C++ - se cere specificare și testare pentru funcția dată
 - se dă specificația sau niște funcții de test - se cere implementarea c++
- concepte din C/C++ și programarea orientată obiect : alocare dinamică, constructor, destructor, moștenire, polimorfism, metode virtuale, clase abstracte, suprascriere, supraâncarcare, excepții, streamul IO, containere, iteratori, algoritmi STL, etc.
 - Se dă un cod c++ în care se folosesc anumite concepte - se cere rezultatul rulării, identificarea/explorarea erorilor,
- Code guidelines: memory leak, dangling pointer, const correctness, exception safe code.
 - Se dă codul c++ trebuie să identificați problemele și să scrieți codul echivalent care elimina ne-ajunsurile
- UML – C++ - řabloane de proiectare
 - se dă un cod c++ - se cere diagrama UML de clasă
 - Se dă o diagramă UML de clase – se cere codul c++
 - Se dă o descriere a unor clase și relațiile între clase – se cere codul c++
- Qt
 - se dă codul sursă Qt – se cere o schiță a interfeței grafice, explicații despre ce își propune codul dat
 - se dă o schiță a interfeței grafice – se cere codul Qt care construiește interfața utilizator conform schiței

Exemple de probleme examen scris

Specificati si testati functia:

```
vector<int> f(int a) {
    if (a < 0)
        throw MyException("Illegal argument");
    vector<int> rez;
    for (int i = 1; i <= a; i++) {
        if (a % i == 0) {
            rez.push_back(i);
        }
    }
    return rez;
}
```

Definiți o clasa *grades* ce reprezintă notele obținute de un student, astfel încât următoarea secvență C++ sa fie corecta sintactic si sa efectueze ceea ce indica comentariile.

```
#include <iostream>
#include <vector>
int main() {
    grades<int> myg;
    myg = myg + 10; // adaugam nota 10 la OOP
    myg = myg + 9; //adaugam nota 9 la FP
    double avg = 0.0;
    for (auto g:myg) { //iteram toate notele
        avg+=g;
    }
    return avg/myg.getNRGrades(); //compute average
}
```

Indicați rezultatul execuției pentru următorul program c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
#include <iostream>
using namespace std;
class A {
public:
    A() {cout << "A" << endl;}
    ~A() {cout << "~A" << endl;    }
    void print() {cout << "print" << endl;}
};
void f() {
    A a[2];
    a[1].print();
}
int main() {
    f();
}
```

Code guidelines – se da un cod C++ care funcționează (compilează) dar care nu respectă stilul promovat de noi sau are probleme cu: memoria, copieri ne-necesare, const correctness, etc.

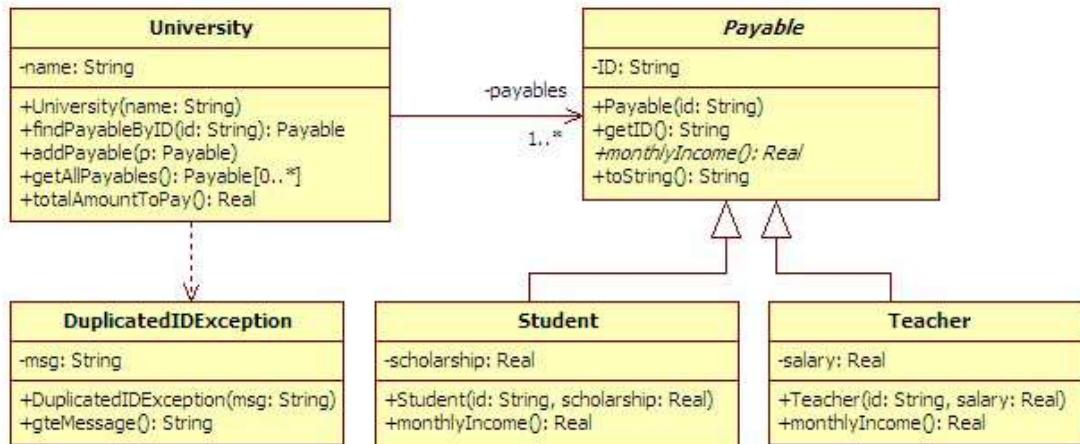
Identificați problemele în codul C++ de mai jos. Scrieți o funcție echivalent funcțional care rezolvă problemele identificate.

```
int function(vector<Point> points) {
    Point* aux = new Point{ 0,0 };
    for (auto p: points) {
        aux->x += calcul(p.x);
        aux->y += calcul(p.y);
    }
    if (aux->x > 10 || aux->y > 10) {
        return;
    }
    int rez = aux->x + aux->y;
    delete aux;
    return rez;
}
```

```
//const correctness - se face o copie la vector care nu e necesara
int function(vector<Point> points) {
    //Now owning raw pointers - de evitat pointerii cand nu e necesar
    Point* aux = new Point{ 0,0 };
    //const correctness - se face o copie la fiecare point
    for (auto p : points) {
        //exception safe code - daca calcul arunca exceptie avem memory leak
        aux->x += calcul(p.x);
        aux->y += calcul(p.y);
    }
    if (aux->x > 10 || aux->y > 10) {
        return;//memory leak
    }
    int rez = aux->x + aux->y;
    delete aux;
    return rez;
}

int myfunction(const vector<Point>& points) {
    Point aux{ 0,0 };
    for (const auto& p : points) {
        aux.x += calcul(p.x);
        aux.y += calcul(p.y);
    }
    if (aux.x > 10 || aux.y > 10) {
        return;
    }
    return aux.x + aux.y;
}
```

Scrieți codul C++ ce corespunde diagramei de clase UML.



- Universitatea are două tipuri de entități: studenți și profesori.
- Pentru studenți universitatea platește burse lunare, pentru profesor trebuie să platească salarii. Metoda `monthlyIncome()` returnează suma datorată pentru fiecare entitate (valoarea bursei pentru student, respectiv salarul pentru profesor).
- Metoda `toString()` din clasa **Payable** tipărește id-ul, urmat de suma de platit.
- Metoda `getAmountToPay()` din clasa **University** calculează suma totală de platit (atât burse cat și salarii).
- Metoda `addPlayables` arunca excepție dacă se adaugă un **Playable** cu un id care mai există.

Scrieți un program care creează o instanță de Universitate, adăugă mai multe entități (atât studenți cat și profesori) și tipărește toate entitățile (folosind metoda `toString`) și suma totală ce trebuie plătit de către universitate (`getAmountToPay()`).

Implementați corect gestiunea memoriei. Exemplificați aruncarea și tratarea excepțiilor în C++ prin adăugări a două entități cu același id.

Examen practic

În aceeași zi cu examenul scris

Durată: 2.5 - 3 ore

Se cere o aplicație cu interfață grafică utilizator (QT).

Aplicația se dezvoltă pornind de la un proiect gol, nu se pot folosi coduri surse externe (existente)

Se poate folosi:

- QT Assistant
- Pe o foaie A4 se pot scrie API - signaturi de metode, constante, operatori, include-uri,etc (fără algoritmi)

Aplicația de dezvoltat:

- Citește scrie date din/in fișier text
- Validează datele introduse de utilizator
- Folosește arhitectura stratificată
- Specificații și teste
- 4-6 funcționalități
 - Se punctează doar acele funcționalități care se pot demonstra executând aplicația (nu se dau puncte pentru cod sursă)

Se pot folosi laptopuri proprii la examen - orice mediu de dezvoltare

Se pot folosi calculatoarele facultății - Qt + Qt Creator

Ca și la simulare, în plus posibil:

- Mai multe ferestre, componente create dinamic
- Observer
- Componente cu modele (Model/View)
- Desenare (QPainter)

Examen

Nu copiați. Copiatul este furt/frauda.

Cei care copiază pică materia (nu sunt primiți în examenul din restante)

Pentru pregătire - faceți exerciții în condiții similare ca și la simulare. Vedeți cât timp vă ia, care sunt problemele de care vă loviți.

Construiți aplicația incremental. Pași mici, salvat-compilat-testat frecvent. Folosiți dezvoltarea bazate pe teste.

Feature driven – să va concentrați pe o singura cetești și implementați minimul necesar pentru respectiva funcționalitate

Adăugați câte o metodă odată să puteți reveni ușor la o versiune anterioară care funcționa.

Nu ignorați erorile, rezolvați problema înainte să treceți mai departe. Nu treceți mai departe dacă nu știți dacă ultimul lucru adăugat funcționează.

Nu ignorați warningurile, ele pot indica erori în program (ex. pointeri)

Folosiți containere, algoritmi STL.

Nu implementați lucruri care nu se cer, ele nu se vor puncta (Ex nu implementați ștergere dacă nu se cere în problemă). Nu se dau puncte pe volumul de cod sursă.

Dacă ceva nu va ieșe, încercați să treceți peste, implementați o variantă banală (Ex. În loc să citească din fișier returnează niște obiecte hardcodate) și reveniți ulterior pentru a rezolva problema.

Dacă folosiți laptopul propriu asigurați-vă că funcționează (încărător, softurile necesare instalate și funcționale).