

Programare orientată obiect

Limbajul C++

Evoluția limbajelor de programare

Cod mașină

- programul în format binar, executat direct de processor

Limbaj de asamblare

- instrucțiuni în format binar înlocuit cu mnemonice, etichete simbolice pentru access la memorie

Procedural

- descompune programul în proceduri/funcții

Modular

- descompune programul în module

Orientat obiect

- descompune programul într-o multime de obiecte care interacționează

Paradigma de programare orientată-obiect

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 data (clasă)

Un obiect este o entitate care:

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

Poate fi privit ca și o combinație de:

- date (atribute)
- metode

Concepte:

- obiect
- clasă
- metodă (mesaj)

Proprietăți:

- abstractizare
- încapsulare
- moștenire
- polimorfism

Limbajul de programare C/C++

De ce C/C++:

- Folosit la scară largă atât în mediul academic cât și în industrie
- limbaj hibrid , implementează toate conceptele necesare pentru programare orientat obiect (C++)
- multe limbaje de programare au evoluat din C/C++ (Java, C#). Este mult mai ușor să înveți aceste limbaje dacă știi C/C++

Avantaje:

- **Concis, cod lizibil**
- **Performanță:** programele scrise în c/c++ în general sunt mai rapide decât cele scrise în alte limbaje de programare
- **Întreținere:** codul fiind concis programatorul are de întreținut un volum mai mic de cod
- **Portabil:** se pot scrie aplicații pentru orice fel de procesor, sistem de operare
- **Productivitate:** C++ a fost creat cu scopul principal de a mări productivitatea (față de C).

Limbaje compilate. Procesul de compilare

Programul C/C++ trebuie compilat pentru a putea fi executat.

Programul scris în fișiere text (fișiere sursă) trebuie transformat în cod binar ce poate fi executat de procesor:

Fișiere sursa - fișiere text, conțin programul scris într-un limbaj de programare

| - compilatorul, analizează fișierele și crează fișiere obiect

Fișiere Obiect - fișiere intermediare, conține bucați incomplete din programul final

| - linker, combină mai multe fișiere obiect și crează programul care poate fi executat de calculator

Executabil

| - sistemul de operare, încarcă fișierul executabil în memorie și execută programul

|

Program in memorie

În C/C++, pașii sunt executate înaintea rulării programului .

În unele limbaje transformarea se execută în timpul rulării. Acesta este una dintre motivele pentru care C/C++ în general are o performanță mai bună decât alte limbaje mai recente.

Python (Interpretat) vs C/C++ (compilat)

Mediu de dezvoltare pentru C/C++ (IDE - Integrated Development Environment)

Compiler

MinGW - - Minimalist GNU for Windows, implementare nativă Windows pentru compilatorul GNU Compiler Collection (GCC)

MinSYS – colecție de utilitare GNU (bash, make, gawk, grep)

Eclipse IDE

Eclipse CDC – mediu de dezvoltare integrat pentru C/C++ bazat pe platforma Eclipse

Colecție de pluginuri care ofera instrumentele necesare pentru a dezvolta aplicații în C/C++

Project C - Hello Word

Elemente de bază

C/C++ este case sensitive (a <> A)

Identificator:

- Secvență de litere și cifre, începe cu o literă sau “_”(underline).
- Nume pentru elemente din program (nume variabile, funcții, tipuri, etc)
- Ex. i, myFunction, rez,

Cuvinte rezervate (Keywords):

- Identificatori cu semantica specială pentru compilator
- int, if, for, etc.

Literals:

- Constante specificate direct în codul sursă
- Ex. “Hello”, 72, 4.6, ‘c’

Operatori:

- aritmetici, pe biti, relaționali, etc
- +, -, <<

Separatori:

- Semne de punctuație folosite pentru a defini structura programului
- ; { }, ()

Whitespace:

- Caractere ignorate de compilator
- space, tab, linie nouă

Comentarii:

- // this is a single line comment
- /* This is a
* multiline comment
*/
- sunt ignorate de compilator

Tipuri de date

Un tip de date definește domeniul de valori și operațiile ce sunt definite pentru valorile din domeniu

Tipuri de date (Built in):

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

- operații: +, -, *, /, %
- relații: <, >, <=, >=, ==, !=
- operațiile se pot executa doar dacă operanzii sunt de tipuri compatibile
- precedența operatorilor dictează ordinea de execuție

Vectori

Daca **T** e un tip de dată:

- $T[n]$ – este un vector cu n elemente de tip T
- indicii sunt de la 0 la $n-1$
- operatorul de indexare: $[]$
- vector multidimensional : $t[n][m]$

```
#include <stdio.h>

int main() {
    int a[5]; // create an array with 5 elements
    a[0] = 1; //index start from 0
    a[1] = 2;
    printf("a[0]=%d \n", a[0]);
    printf("a[1]=%d \n", a[1]);
    //!!! a[2] uninitialised
    printf("a[2]=%d \n", a[2]);

    int b[] = { 1, 2, 3, 5, 7 };
    printf("b[0]=%d \n", b[0]);
    b[0] = 10;
    printf("b[0]=%d \n", b[0]);
    return 0;
}
```

Structuri (Record)

- este o colectie de elemente de tipuri diferite
- permite gruparea diferitelor tipuri de date simple într-o structură

<pre>struct name{ type1 field1; type2 field2 }</pre>	<pre>struct car{ int year; int nrKm; }</pre>	<pre>car c; c.year = 2010 c.nrKm = 30000;</pre>
---	---	---

```
#include <stdio.h>  
  
//introduce a new struct called Car  
typedef struct {  
    int year;  
    int km;  
} Car;  
  
int main() {  
    Car car, car2;  
  
    //initialise fields  
    car.year = 2001;  
    car.km = 20000;  
  
    printf("Car 1 fabricated:%d Km:%d \n", car.year, car.km);  
  
    //!!! car2 fields are uninitialised  
    printf("Car 1 fabricated:%d Km:%d \n", car2.year, car2.km);  
    return 0;  
}
```

Declarații de variabile

- Introduce un nume în program, asociază numele cu un tip
- Se alocă memorie conform tipului variabilei
- Tipul variabilei este folosit de compilator pentru a decide care sunt valorile și operațiile posibile
- Valoarea este nedefinită după declarare.
- Este recomandat să combinăm declarația cu inițializarea
- Trebuie ales nume sugestiv pentru orice variabilă din program
- Variabila trebuie inițializată după declarație cu o valoare

<type> <identifier>

Ex. int i

long rez

int j=7

Constante

- numerice: 1, 12, 23.5
- sir de caractere: "Hello World"
- caracter: 'c'

Referințe și pointeri

- Pointer este un tip de date special, folosit pentru a lucra cu adrese de memorie
- poate stoca adresa unei variabile, adres unei locații de memorie

Declarare

- ca și orice variabilă de alt tip doar ca se pune '*' înaintea numelui variabilei.
- Ex: int *a; long *a, char *a

Operatori

- *address of '&'*: - returneaza adresa de memorie unde este stocat valoarea dintr-o variabilă
- *dereferencing '*'* - returneaza valoarea stocata în locația de memorie specificată

```
#include <stdio.h>

int main() {
    int a = 7;
    int *pa;

    printf("Value of a:%d address of a:%p \n", a, &a);
    //assign the address of a to pa
    pa = &a;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);

    //a and pa refers to the same memory location
    a = 10;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);
    return 0;
}
```

Instrucțiuni

Toate instrucțiunile se termina cu: ; (excepție instrucțiunea compusă)

Expresii: `a = b+c; i++; a==b`

Instrucțiune vidă: ;

Instrucțiunea compusă:

```
{  
//multiple statements here  
}
```

Atribuire

- Operator de atribuire: =
- Atribuire o valoare la o variabilă (inițializează sau scimba valoare variabilei)

if, if-else, else if

```
if (condition){  
    //statements executed only if the condition is true  
}
```

```
if (condition){  
    //statements executed if the condition is true  
} else {  
    //statements executed only if the condition is not true  
}
```

```
if (condition1){  
    //statements executed if the condition1 is true  
} else if (condition2){  
    //statements executed only if condition1 is not true and the condition2 is true  
}
```

- *condition*, *condition1*, *condition2* - sunt expresii
- orice expresie are o valoare
- valoarea 0 înseamnă fals, orice altă valoare înseamnă adevărat

switch-case

```
switch(expression)
{
  case constant1:
    statementA1
    statementA2
    ...
    break;
  case constant2:
    statementB1
    statementB2
    ...
    break;
  ...
  default:
    statementZ1
    statementZ2
    ...
}
```

Se evalueaza expresia, daca valoarea este egal cu constant1 atunci tot ce e pe ramura **case** constant1: se execută până la primul **break**;

Dacă nu e egal cu constant1 se verifică cu constant2, 3...

Dacă nici o constantă nu este egală cu rezultatul expresiei atunci se executa ramura **default**

Instrucțiuni repetitive

Execută repetat un set de instrucțiuni

while , do-while

```
while(condition)
{
    statement1
    statement2
    ...
}
```

Cât timp condiția este adevărată (!=0) se execută corpul de instrucțiuni

```
do
{
    statement1
    statement2
    ...
}
while(condition);
```


for

```
for(initialization; condition; incrementation)  
{  
    //body  
}
```

initialization – inițializează una sau mai multe variabile

incrementation – se execută la fiecare iterație

condition – se verifică la fiecare iterație, cât timp e adevărat corpul de instrucțiuni se execută

<pre>for(<i>initialization</i>; <i>condition</i>; <i>incrementation</i>) { statement1 statement2 ... }</pre>	<pre><i>initialization</i> while(<i>condition</i>) { statement1 statement2 ... <i>incrementation</i> }</pre>
---	---

Citire/Scriere

printf() - tipărește în consola (la ieșirea standard)

```
#include <stdio.h>

int main() {
    int nr = 5;
    float nrf = 3.14;
    char c = 's';
    char str[] = "abc";

    printf("%d %f %c %s", nr, nrf , c, str);
    return 0;
}
```

scanf() - citește de la tastatura

```
int main() {
    int nr;
    float f;
    printf("Enter a decimal number:");
    //read from the command line and store the value in nr
    scanf("%d", &nr);
    printf("The number is:%d \n", nr);

    printf("Enter a float:");
    if (scanf("%f", &f) == 0) {
        printf("Error: Not a float:");
    } else {
        printf("The number is:%f", f);
    }
    //wait until user enters 'e'
    while(getchar()!='e');
    return 0;
}
```

Calculator numere raționale

Problem statement

Profesorul are nevoie de un program care permite elevilor să învețe despre numere raționale. Programul ajută studenții să efectueze operații aritmetice cu numere raționale

```
#include <stdio.h>

int main() {
    int totalM = 0;
    int totalN = 1;
    int m;
    int n;
    while (1) {
        printf("Enter m, then n to add\n");
        scanf("%d", &m);
        scanf("%d", &n);
        totalM = totalM * n + m * totalN;
        totalN = totalN * n;
        printf("Total: %d/%d\n", totalM, totalN);
    }
    return 0;
}
```

Funcții

Funcția main este executat cand lansam in execuție un program C/C++

Declarare

<result type> name (<parameter list>);

<result-type> - tipul rezultatului, poate fi orice tip sau *void* *daca funcția nu returnează nimic*

<name> - numele funcției

<parameter-list> - parametrii formali

Corpul funcției nu face parte din declarare

Definiție

```
<result type> name(<parameter list>){  
//statements - the body of the function  
}
```

- **return** <exp> rezultatul expresiei se returnează, execuția funcției se termina
- o funcție care nu este void trebuie neapărat sa returneze o valoare prin expresia ce urmează dupa **return**
- declararea trebuie sa corespunda cu definiția (numele parametrilor poate fi diferit)

Specificații

- Nume sugestiv
- O scurtă descriere a funcției (ce face)
- Semnificația parametrilor
- condiții asupra parametrilor (precondiții)
- ce se returnează
- relația dintre parametri și rezultat (post condiții)

```
/*  
* Verify if a number is prime  
* nr - a number, nr>0  
* return true if the number is prime (1 and nr are the only dividers)  
*/  
bool isPrime(int nr);
```

precondiții - sunt condiții care trebuie să fie satisfăcute de parametrii actuali înainte de a executa corpul funcției

postcondiții - condiții care sunt satisfăcute după execuția funcției

Apelul de funcții

name (<parameter list>);

- Toate expresiile date ca parametru sunt evaluate înainte de execuția funcției
- Parametrii actuali trebuie să corespundă cu parametri formali (număr, poziție, tip)
- declarația trebuie să apară înainte de apel

Supraîncărcare (Overloading)

- Pot exista mai multe funcții cu același nume (dar parametrii formali diferiți)
- La apel se va executa funcția care corespunde parametrilor actuali

Vizibilitate (scope)

Locul unde declarăm variabila determină vizibilitate lui (unde este variabila accesibilă).

Variabile, funcții declarate în interiorul unei instrucțiuni compuse ({ }) sunt vizibile doar în interiorul instrucțiunii compuse

Funcțiile au domeniul lor de vizibilitate - variabilele definite în interiorul funcțiilor sunt accesibile doar în funcție, ele sunt distruse după apelul funcției. (**variabile locale**)

Variabilele definite în afara funcțiilor sunt accesibile în orice funcție (**variabile globale**).

Transmiterea parametrilor : prin valoare sau prin referință

Transmitere prin valoare:

La apelul funcției se face o copie a parametrilor.

Schimbările făcute în interiorul funcției nu afectează variabilele exterioare.

Este mecanismul implicit de transmitere a parametrilor în C

Transmitere prin referință:

La apelul funcției se transmite adresa locației de memorie unde se află valoarea variabilei.

Modificările din interiorul funcției sunt vizibile și în afară.

```
void byValue(int a) {
    a = a + 1;
}

void byRef(int* a) {
    *a = *a + 1;
}

int main() {
    int a = 10;
    byValue(a);
    printf("Value remain unchanged a=%d \n", a);
    byRef(&a);
    printf("Value changed a=%d \n", a);
    return 0;
}
```

Vectorul este transmis prin referință

Calculator

```
/*
 * Return the greatest common divisor of two natural numbers.
 * Pre: a, b >= 0, a*a + b*b != 0
 */
int gcd(int a, int b) {
    if (a == 0) {
        return b;
    } else if (b == 0) {
        return a;
    } else {
        while (a != b) {
            if (a > b) {
                a = a - b;
            } else {
                b = b - a;
            }
        }
        return a;
    }
}
/*
 * Add (m, n) to (toM, toN) - operation on rational numbers
 * Pre: toN != 0 and n != 0
 */
void add(int* toM, int* toN, int m, int n) {
    *toM = *toM * n + *toN * m;
    *toN = *toN * n;
    int gcdTo = gcd(abs(*toM), abs(*toN));
    *toM = *toM / gcdTo;
    *toN = *toN / gcdTo;
}
//global variables. Store the current total
int totalM = 0;
int totalN = 1;
int main() {
    int m,n;
    while (1) {
        printf("Enter m, then n to add\n");
        scanf("%d", &m);scanf("%d", &n);
        add(&totalM, &totalN, m, n);
        printf("Total: %d/%d\n", totalM, totalN);
    }
    return 0;
}
```


Curs 1

- Introducere - POO
- Limbajul C
 - sintaxă
 - tipuri de date
 - instructiuni
 - funcții

Funcții

Declarare

<result type> name (<parameter list>);

<result-type> - tipul rezultatului, poate fi orice tip sau *void* *daca funcția nu returnează nimic*

<name> - numele funcției

<parameter-list> - parametrii formali

Corpul funcției nu face parte din declarare

Definiție

```
<result type> name(<parameter list>){  
//statements - the body of the function  
}
```

- **return** <exp> rezultatul expresiei se returnează, execuția funcției se termina
- o funcție care nu este void trebuie neapărat sa returneze o valoare prin expresia ce urmează dupa **return**
- declararea trebuie sa corespunda cu definiția (numele parametrilor poate fi diferit)

Funcția *main* este executat cand lansam in execuție un program C/C++

Specificații

- Nume sugestiv
- O scurtă descriere a funcției (ce face)
- Semnificația parametrilor
- condiții asupra parametrilor (precondiții)
- ce se returnează
- relația dintre parametri și rezultat (post condiții)

```
/*  
* Verify if a number is prime  
* nr - a number, nr>0  
* return true if the number is prime (1 and nr are the only dividers)  
*/  
bool isPrime(int nr);
```

precondiții - sunt condiții care trebuie să fie satisfăcute de parametrii actuali înainte de a executa corpul funcției

postcondiții - condiții care sunt satisfăcute după execuția funcției

Apelul de funcții

name (<parameter list>);

- Toate expresiile date ca parametru sunt evaluate înainte de execuția funcției
- Parametrii actuali trebuie să corespundă cu parametri formal (număr, poziție, tip)
- declarația trebuie să apară înainte de apel

Supraîncărcare (Overloading)

- Pot exista mai multe funcții cu același nume (dar parametrii formali diferiți)
- La apel se va executa funcția care corespunde parametrilor actuali

Vizibilitate (scope)

Locul unde declarăm variabila determină vizibilitate lui (unde este variabila accesibilă).

Variabile locale

- variabila este vizibilă doar în interiorul instrucțiunii compuse (`{ }`) unde a fost declarată
- variabilele declarate în interiorul funcției sunt vizibile (accesibile) doar în funcție
- instrucțiunile `if`, `if else`, `for`, `while` au domeniul propriu de vizibilitate
- Încercarea de a accesa o variabilă în afara domeniului de vizibilitate generează eroare la compilare.
- Ciclul de viață a unei variabile începe de la declararea lui și se termină când execuția iese din domeniul de vizibilitate a variabilei (variabila se distruge, memoria ocupată se eliberează)

Global Variables

- Variabilele definite în afara funcțiilor sunt accesibile în orice funcție, domeniul lor de vizibilitate este întreg aplicația
- Se recomandă evitarea utilizării variabilelor globale (există soluții mai bune care nu necesită variabile globale)

Transmiterea parametrilor : prin valoare sau prin referință

Transmitere prin valoare:

La apelul funcției se face o copie a parametrilor.

Schimbările făcute în interiorul funcției nu afectează variabilele exterioare.

Este mecanismul implicit de transmitere a parametrilor în C

Transmitere prin referință:

La apelul funcției se transmite adresa locației de memorie unde se află valoarea variabilei.

Modificările din interiorul funcției sunt vizibile și în afară.

```
void byValue(int a) {
    a = a + 1;
}

void byRef(int* a) {
    *a = *a + 1;
}

int main() {
    int a = 10;
    byValue(a);
    printf("Value remain unchanged a=%d \n", a);
    byRef(&a);
    printf("Value changed a=%d \n", a);
    return 0;
}
```

Vectorul este transmis prin referință

Calculator

```
/*
 * Return the greatest common divisor of two natural numbers.
 * Pre: a, b >= 0, a*a + b*b != 0
 */
int gcd(int a, int b) {
    if (a == 0) {
        return b;
    } else if (b == 0) {
        return a;
    } else {
        while (a != b) {
            if (a > b) {
                a = a - b;
            } else {
                b = b - a;
            }
        }
        return a;
    }
}
/*
 * Add (m, n) to (toM, toN) - operation on rational numbers
 * Pre: toN != 0 and n != 0
 */
void add(int* toM, int* toN, int m, int n) {
    *toM = *toM * n + *toN * m;
    *toN = *toN * n;
    int gcdTo = gcd(abs(*toM), abs(*toN));
    *toM = *toM / gcdTo;
    *toN = *toN / gcdTo;
}
//global variables. Store the current total
int totalM = 0;
int totalN = 1;
int main() {
    int m,n;
    while (1) {
        printf("Enter m, then n to add\n");
        scanf("%d", &m);scanf("%d", &n);
        add(&totalM, &totalN, m, n);
        printf("Total: %d/%d\n", totalM, totalN);
    }
    return 0;
}
```

Funcții de test

Assert

```
#include <assert.h>
void assert (int expr);
```

expr – Se evaluează expresia. Dacă e fals (=0) metoda assert generează o eroare și se termină execuția aplicației

Mesajul de eroare depinde de compilator (pot fi diferențe în funcție de compilator), conține informații despre locul unde a apărut eroarea (fișierul, linia), expresia care a generat eroare.

Vom folosi instrucțiunea assert pentru a crea teste automate.

<pre>#include <assert.h> /* * greatest common divisor . * Pre: a, b >= 0, a*a + b*b != 0 * return gcd */ int gcd(int a, int b) { a = abs(a); b = abs(b); if (a == 0) { return b; } if (b == 0) { return a; } while (a != b) { if (a > b) { a = a - b; } else { b = b - a; } } return a; }</pre>	<pre>/** * Test function for gcd */ void test_gcd() { assert(gcd(2, 4) == 2); assert(gcd(3, 27) == 3); assert(gcd(7, 27) == 1); assert(gcd(7, -27) == 1); }</pre>
---	---

Review: Calculator – varianta procedurală

Profesorul are nevoie de un program care permite elevilor să învețe despre numere raționale. Programul ajută studenții să efectueze operații aritmetice cu numere raționale

```
/**
 * Test function for gcd
 */
void test_gcd() {
    assert(gcd(2, 4) == 2);
    assert(gcd(3, 27) == 3);
    assert(gcd(7, 27) == 1);
    assert(gcd(7, -27) == 1);
}

/**
 * Add (m, n) to (toM, toN) - operation on rational numbers
 * Pre: toN != 0 and n != 0
 */
void add(int* toM, int* toN, int m, int n) {
    *toM = *toM * n + *toN * m;
    *toN = *toN * n;
    int gcdTo = gcd(abs(*toM), abs(*toN));
    *toM = *toM / gcdTo;
    *toN = *toN / gcdTo;
}

int main() {
    test_gcd();
    int totalM = 0, totalN = 1;
    int m, n;
    while (1) {
        printf("Enter m, then n to add\n");
        scanf("%d", &m);
        scanf("%d", &n);
        add(&totalM, &totalN, m, n);
        printf("Total: %d/%d\n", totalM, totalN);
    }
    return 0;
}
```


Principii de proiectare pentru funcții

- **Fiecare funcție sa aibă o singură responsabilitate (Single responsibility principle)**
- **Folosiți nume sugestive (nume funcție, nume parametri, variabile)**
- **Folosiți reguli de denumire (adauga_rational, adaugaRational, CONSTANTA), consistent în toată aplicația**
- **Specificați fiecare funcție din aplicație**
- **Creați teste automate pentru funcții**
- **Funcția trebuie sa fie usor de testat, (re)folosit, înțeles și modificat**
- **Folosiți comentarii în cod (includeți explicații pentru lucruri care nu sunt evidente în cod)**
- **Evitați (pe cât posibil) funcțiile cu efect secundar**

Modular programming in C++.

Modulul este o colecție de funcții și variabile care oferă o funcționalitate bine definită.

Fișiere Header .

Declarațiile de funcții sunt grupate într-un fișier separat – fișier header (.h).

Implementarea (definițiile pentru funcții) într-un fișier separat (.c/.cpp)

Scop

Separarea interfeței (ce oferă modulul) de implementare (cum sunt implementate funcțiile)

Separare specificații, declarații de implementare

Modulele sunt distribuite în general prin: fișierul header + fișierul binar cu implementările (.dll,.so)

- Nu e nevoie să dezvălui codul sursă (.c/.cpp)

Cei care folosesc modulul au nevoie doar de declarațiile de funcții (fișierul header) nu și de implementări (codul din fișierele .c/.cpp)

Directive de preprocessare

Preprocesarea are loc înainte de compilare.

cod sursă – preprocessare – compilare – linkeditare – executabil

Permite printre altele: includere de fișiere header, definire de macrouri, compilare condiționată

Directiva Include

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

Pentru a avea acces la funcțiile declarate într-un modul (bibliotecă de funcții) se folosește directiva **#include**

Preprocessorul include fișierul referit în fișierul sursă în locul unde apare directiva

avem două variante pentru a referi un modul: < > sau ""

```
#include "local.h" //caută fișierul header relativ la directorul current al aplicației
```

```
#include <header> //caută fișierul header între bibliotecile system (standard compiler include paths )
```

Aplicații modulare C/C++

Codul este împărțit în mai multe fișiere header (.h) și implementare (.c)

- fișierele **.h** conțin declarații (interfața)
- **.c** conține definiția (implementarea) funcțiilor

se grupează funcții în module astfel încât modulul să ofere o funcționalitate bine definită (puternic coeziv)

- Când un fișier .h se modifică este nevoie de **recompilarea** tuturor modulelor care îl referă (direct sau indirect)
- Fișierele .c se pot compila separat, modificarea implementării nu afectează modulele care folosesc (ele referă doar definițiile din header)

Headerul este un **contract** între cel care dezvoltă modulul și cel care folosește modulul.

Detaliile de implementare sunt ascunse în fișierul .h

Review: Calculator versiune modulară

Module:

- **calculatorui.c – interfața utilizator**
- **calculator.h, calculator.c - TAD Calculator, operatii cu calculator**
- **rational.h, rational.c - TAD rational, operatii cu numere rationale**
- **util.h, util.c - funcții utile de operații cu numere (gcd)**

Declarație multiplă – directivele #ifndef și #define

Într-un program mai complex este posibil ca un fișier header să fie inclus de mai multe ori. Asta ar conduce la declarații multiple pentru funcții

Soluție: se folosesc directivele de preprocesare

#ifndef, #ifdef, #define, #endif

Se poate verifica dacă modulul a fost deja inclus, respectiv să marcăm când un modul a fost inclus (prin definirea unei etichete)

```
#ifndef RATIONAL_H_    /* verify if RATIONAL_H_ is already defined, the rest
                       (until the #endif will be processed only if RATIONAL_H_ is
                       not defined*/
#define RATIONAL_H_    /* define RATIONAL_H_ so next time the preprocessor will not
                       include this */

/**
 * New data type to store rational numbers
 */
typedef struct {
    int a, b;
} Rational;

/**
 * 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 sum(Rational nr1, Rational nr2, Rational &rez);

#endif /* RATIONAL_H_ */
```

Principii de proiectare pentru module

- **Separați interfața de implementare**
 - Headerul conține doar declarații, implementările în fișierul .c
- **Includeți la începutul fișierului header un comentariu, o scurta descriere a modulului**
- **Creați module puternic coezive**
 - fiecare modul o singură funcționalitate, are o singură responsabilitate
- **Șablonul - Arhitectură stratificată**
 - Straturi: ui, control, model, validation, repository
 - Controlul dependențelor - Fiecare nivel depinde doar de nivelul următor
- **Tip abstract de date – TAD**
 - operațiile definite in header (interfață) /implementarea in .c
 - ascundere detalii de implementare
 - specificații abstracte (independent de implementare, ce face nu cum)

Biblioteci standard

`#include <stdio.h>`

Operatii de intrare/iesire

`#include <math.h>`

Funcții matematice – abs, sqrt, sin, exp, etc

`#include <string.h>`

sirul de caractere in C - vector de char care se termina cu caracterul `'\0'`

`strncpy` - copiează string

`strcat` - concatenează string

`strcmp` - compară stringuri

`strlen` - lungimea stringului

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

int main(void) {
    char arr[4]; // for accommodating 3 characters and one null '\0' byte.
    char *ptr = "abc"; //a string containing 'a', 'b', 'c', '\0'

    memset(arr, '\0', sizeof(arr)); //reset all
    strncpy(arr, ptr, sizeof("abc")); // Copy the string

    printf("\n %s \n", arr);

    arr[0] = 'p';

    printf("\n %s \n", arr);
    return 0;
}
```


Pointeri

Pointer este un tip de date , folosit pentru a lucra cu adrese de memorie - poate stoca adresa unei variabile, adres unei locații de memorie

Operatori: '&', '*'

```
#include <stdio.h>

int main() {
    int a = 7;
    int *pa;

    printf("Value of a:%d address of a:%p \n", a, &a);
    //assign the address of a to pa
    pa = &a;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);

    //a and pa refers to the same memory location
    a = 10;
    printf("Value of pa:%d address of pa:%p \n", *pa, pa);
    return 0;
}
```

Null pointer

- valoare specială (0) pentru a indica faptul ca pointerul nu referă o memorie validă

Pointer invalid (Dangling pointer)

Adresa refertă de pointer e invalid

<pre>#include <stdio.h> int main() { //init to null int *pa1 = NULL; int *pa2; //!!! pa2 refers to an unknown address *pa2 = 6; if (pa1==NULL){ printf("pa1 is NULL"); } return 0; }</pre>	<pre>#include <stdio.h> int* f() { int localVar = 7; printf("%d\n", localVar); return &localVar; } int main() { int* badP = f(); //!!! *badP refera o adresa de memorie //care a fost deja eliberata printf("%d\n", *badP); }</pre>
--	--

Vectori / pointeri - Aritmetica pointerilor

O variabila de tip vector - un pointer la primul element al vectorului

- vectorul este transmis prin referinta (se transmite adresa de memorie al primului element din vector – nu se face o copie).
- Indicele porneste de la 0 – primul element este la distanță 0 față de începutul vectorului.
- Expresia `array[3]` – compilatorul calculează care este locația de memorie la distanță 3 față de începutul vectorului.
- Cu funcția `sizeof(var)` se poate afla numarul de bytes ocupat de valoarea din `var` (depinde de tipul lui `var`)

Aritmetica pointerilor

Folosirea de operații adăugare/scadere pentru a naviga în memorie (adrese de memorie)

```
#include <stdio.h>

int main() {
    int t[3] = { 10, 20, 30 };
    int *p = t;
    //print the first elem
    printf("val=%d adr=%p\n", *p, p);

    //move to the next memory location (next int)
    p++;
    //print the element (20)
    printf("val=%d adr=%p\n", *p, p);
    return 0;
}
```

`p++` în funcție de tipul valorii referite de pointer, compilatorul calculează următoarea adresa de memorie.

Gestiunea memoriei

Pentru variabilele declarate intr-o aplicație, compilatorul alocă memorie pe **stivă** (o zonă de memorie gestionat de compilator)

```
int f(int a) {
    if (a>0){
        int x = 10; //memory for x is allocated on the stack
    }
    //here x is out of scope and the memory allocated for x is no longer reserved
    //the memory can be reused
    return 0;
}
```

```
int f(int a) {
    int *p;
    if (a>0){
        int x = 10;
        p = &x;
    }
    //here p will point to a memory location that is no longer reserved
    *p = 5; //!!! undefined behavior, the program may crash
    return 0;
}
```

Memoria este automat eliberată de compilator în momentul în care execuția părăsește domeniul de vizibilitate a variabilei.

La iesire dintr-o funcție memoria alocată pentru variabile locale este eliberată automat de compilator

Alocare dinamică

Folosind funcțiile **malloc**(size) și **free**(pointer) programatorul poate alocă memorie pe Heap – zonă de memorie gestionat 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) {
    char* newStr;
    int len;
    len = strlen(str) + 1; // +1 for the '\0'
    newStr = malloc(sizeof(char) * len); // allocate memory
    strcpy(newStr, str); // copy string
    return newStr;
}
```

Programatorul este responsabil sa dealoce memoria

Memory leak

Programul alocă memorie dar nu dealocă 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;
}
```

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 folosi **void*** pentru a crea structuri de date care funcționează cu orice tip de elemente

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

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 distruge(VectorDinamic *v) {
    int i;
    for (i = 0; i < v->lg; i++) {
        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);

/**
 * Aloca memorie aditionala 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)(); // a pointer to a function
void *funcPtr();  // a function that returns a pointer
```

```
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
}
```

Putem folosi pointer la funcții în structurile de date generice

```
typedef elem (*copyPtr)(elem&, elem);

typedef int (*equalsPtr)(elem, elem);
```

Limbajul de programare C++

Urmașul limbajului C apărut în anii 80 (C cu clase), dezvoltat inițial de Bjarne Stroustrup

Bibliografie: B. Stroustrup, The C++ Programming Language, Addison Wesley

Limbajul C++

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

Tipul de date mulțime - modular, implementat în C++

```
/*set.h*/
struct Mult;
typedef Mult* Set;
/**
 * Create an empty set. Need to be invoked before we can use the set
 * n - the maximum number of element in the set; m - the set
 */
void createSet(Set &m, int n);
/**
 * Add an element to the set
 * m - the set; e the element to be added
 * return 0 if we can not add the element
 */
int add(Set m, Element e);
```

```
/*set.cpp*/
#include "set.h"
struct Mult {
    Element* e;
    int c;
    int max;
};
/**
 * Release the memory allocated for this set
 */
void destroyM(Set& m) {
    for (int i = 0; i < m->c; i++)
        destroyE(m->e[i]);
    delete[] m->e;
    delete m;
}
/**
 * Add an element to the set
 * m - the set; e the element to be added
 * return 0 if we can not add the element
 */
int add(Set m, Element e) {
    if (m->c == m->max)
        return 0;
    if (!(contains(m, e))) {
        (m->c)++;
        atrib(e, m->e[(m->c) - 1]);
    }
    return 1;
}
```

Tipul bool

domeniu de valori: adevărat (**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 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;
}
```

Const Pointer

1 `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";  
//change the memory address (valid)  
p2 = &i;  
cout << *p2 << "\n";  
//change the value (compiler error)  
*p2 = 7;  
cout << *p2 << "\n";
```

2 `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";
```

3 `const type* const`

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

Atât adresa cât și valoarea sunt constante

Paradigma de programare orientată-obiect

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 data (clasă)

Un obiect este o entitate care:

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

Poate fi privit ca și o combinație de:

- date (atribute)
- metode

Concepte:

- 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 instanța 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 parametri)
- 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() << endl;  
cout << r2.toFloat() << endl;  
cout << r3.toFloat() << endl;
```

Acces la attribute (câmpuri)

În interiorul clasei

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

Când implementăm metodele avem acces direct la attribute

```
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 attributele 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 crează instanțe noi (se declara o variabilă locală, se crează un obiect folosind **new**)

Numele coincide cu numele clasei, nu are tip returnat

Constructorul alocă memorie pentru datele membre, inițializează attributele

```
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 crează – 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 parametri.

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

Constructor de copiere

Constructor folosit când se face o copie a obiectului

- la atribuire
- la transmitere de parametrii (prin valoare)
- când se returnează o valoare dintr-o metodă

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

Există un constructor de copiere implicit (chiar dacă nu se declară în clasă) acesta copiează câmpurile obiectului, dar nu este potrivit mai ales în cazul în care avem attribute alocate dinamic

Alocare dinamică de obiecte

operatorul new se foloseste pentru alocarea de memorie pe heap pentru obiecte

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


Destructor

Destructorul este apelat de fiecare data cand se dealocă un obiect

- dacă am alocat pe heap (new), se apeleaza la delete
- dacă e variabilă statică, se dealoca în momentul în care nu mai e vizibil (out of scope)

<pre>DynamicArray::DynamicArray() { capacity = 10; elems = new Rational[capacity]; size = 0; }</pre>	<pre>DynamicArray::~~DynamicArray() { delete[] elems; }</pre>
--	---

Obiecte ca parametri 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**

```
/**                                Rational::Rational(const Rational &ot) {
 * Copy constructor                a = ot.a;
 *                                b = ot.b;
 */                                }
Rational(const Rational &ot);
```

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 încercăm 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;
}
```

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[],

Tipuri abstracte de date implementate folosind clase obiecte

– Tip abstract de date

- sparare interfață de implementare
- specificare abstractă (independent de reprezentare/implementare)
- ascunderea detaliilor de implementare

– Clasă

- header : conține doar declarații (interfața). Implementarea în fisier separat (cpp)
- fiecare metodă specificată
- folosind modificatori de acces se poate controla accesul la attribute (metode,variabile membre) , reprezentarea este protejată (nu poate fi accesat din afara clasei)

Listă – implementată secvențial pe vector dinamic (Dynamic Array)

Vector dinamic - tablou unidimensional, lungimea se modifică în timp

```
typedef int TElem;
/**
 * List implemented using a dynamic array data structure
 */
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * r - is a rational number
     */
    void addE(TElem r);
    /**
     * Delete the element from the given position
     * poz - the position of the elem to be deleted, poz>=0;poz<size
     * return the deleted element
     */
    TElem deleteElem(int poz);

    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    TElem get(int poz);
    /**
     * Give the size of the array
     * return the number of elements in the array
     */
    int getSize();
private:
    TElem *elems;
    int capacity;
    int size;
    /**
     * Create enough space to hold nrElem elements
     * nrElems - the number of elements that we need to store
     */
    void ensureCapacity(int nrElems);
```

Regula celor trei (rule of tree)

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

Daca o clasa gestioneaza (este responsabil de) o resursa (memorie heap, fisiere, etc) trebuie sa defineasca:

- constructor de copiere

```
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];
    }
}
```

- operatorul de atribuire

```
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;
}
```

Iterator

- utilizați pentru a parcurge un container de obiecte
- oferă un mecanism generic (abstract) pentru accesul la elemente

Iteratorul va contine

- referință spre containerul pe care-l iterează
- referință spre elementul curent din iteratie (cursor).

Avantaj

permite accesul la elemente fără a expune reprezentarea internă

Clase/Metode prietene (friends)

- Clasa B este clasa prieten cu clasa A dacă B are acces la membrii privați din clasa A
- Util dacă avem nevoie într-o clasa la acces la elementele private dintr-o alta clasă
- Similar este posibil sa avem o funcție prieten cu o clasă.
- Funcția prieten are acces la metode/variabile membre private

Clasă prieten

```
class ItLista {  
public:  
    friend class Lista;
```

Metodă prieten

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


Implementare iterator

```
/**
 * Itarotor over the DynamicArray
 */
class Iterator {
public:
    void urmator() {
        pozCurrent++;
    }
    int valid() {
        return pozCurrent < l->getSize();
    }
    TElem element() {
        return l->elems[pozCurrent];
    }

    Iterator(DynamicArray* _l) {
        this->l = _l;
        this->pozCurrent = 0;
    }
private:
    DynamicArray* l;
    //current element in the iteration
    int pozCurrent;
};

void testIterator() {
    DynamicArray ar1;
    ar1.addE(2);
    ar1.addE(3);
    ar1.addE(4);
    Iterator* it = ar1.begin();
    while (it->valid()) {
        cout << it->element() << " ";
        it->urmator();
    }
    delete it;
    cout << "\n";
}

class DynamicArray {
    friend class Iterator;
    ...
}
```

Iterator – suprascriere operatori

```
DynamicArray ar1; //for like
ar1.addE(2);      for (Iterator i = ar1.begin(); i != ar1.end();i++)
ar1.addE(3);      {
ar1.addE(4);      cout << (*i) << " ";
Iterator it = ar1.begin(); }
while (it != ar1.end()) { //backward
    cout << (*it) << " "; it = ar1.end();
    it++;                 --it;
}                          --it;
cout << "\n";            cout << (*it) << " ";
```

Operatori

```
/**
 * Overload dereferencing operator
 */
TElem& operator*() {
    return l->elems[pozCurrent];
}
/**
 * Overload ++ (prefixed version)
 */
Iterator& operator ++() {
    pozCurrent++;
    return *this;
}
/**
 * Overload ++
 * Postfix version use a dummy param
 */
Iterator& operator ++(int dummy) {
    pozCurrent++;
    return *this;
}

/**
 * Overload !=
 */
bool operator!=(const Iterator& ot) {
    return ot.pozCurrent != pozCurrent;
}
/**
 * Overload -- (prefixed version)
 */
Iterator& operator --() {
    pozCurrent--;
    return *this;
}
```

Listă – implementată înlănțuit

```
class Nod;                                     /**
typedef Nod *PNod;                             * Lista , implementare folosim reprezentare
/**                                             inlantuita
* Reprezinta un nod din inlantuire          */
*/
class Nod {                                     class Lista {
public:                                          public:
    friend class Lista;

    Nod(E e, PNod urm = 0) {                   friend class ItLista;
        this->e = e;
        this->urm = urm;
    }

    E element() {
        return e;
    }

    PNod urmator() {
        return urm;
    }

private:
    //elementul curent
    E e;
    //referinta la nodul urmator
    PNod urm;
};

~Lista();
/**
* Adauga la sfarsitul listei
*/
void adaugaSfarsit(E e);
/**
* Adauga dupa pozitia data de iterator
*/
void adaugaDupa(ItLista i, E e);
/**
* Iterator
*/
ItLista* iterator();

private:
    PNod prim;
};
```

Iterator – Lista implementată înlănțuit

```
/**
 * Iterator pentru lista inlantuita
 */
class ItLista {
public:

    friend class Lista;
    void urmator() {
        curent = curent->urmator();
    }
    int valid() {
        return curent != 0;
    }
    E element() {
        return curent->element();
    }

private:
    ItLista(Lista& _l) :
        l(_l), curent(l.prim) {
    }
    Lista& l;
    PNode curent;
};

void tiparire(Lista& l) {
    ItLista *i = l.iterator();

    while (i->valid()) {
        cout << i->element() << " ";
        i->urmator();
    }
    cout << "\n";
    delete i;
}
```

Lista generica (funcționează cu orice tip de elemente)

- `typedef Telem = <type name>` Ex. `typedef int TElem;`
 - nu putem avea liste cu elemente de tipuri diferite în același program
- Folosim `void*` `typedef void* TElem2;`
 - nu putem adauga constante
 - trebuie sa folosim operatorul cast cand luam elementele

Șabloane (Template)

<pre>int sum(int a, int b) { return a + b; } sum(2,3);</pre>	<pre>double sum(double a, double b) { return a + b; } sum(2.6,3.121);</pre>
---	--

- creare de funcții / clase care folosesc același cod sursă pentru diferite tipuri de date
- în loc sa rescriem funcția / clasa pentru fiecare tip de dată putem folosi mecanismul de șabloane pentru a folosi același cod pentru tipuri diferite
- o modalitate de a refolosi codul

Funcții:

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

or

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

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

- T este un parametru pentru șablon, cand folosim
- instanțiere de șablon + procesul de generare a funcției pentru un tip de date

```
int sum = sumTemp<int>(1, 2);
```

Class template:

Un macro (șablon, skeleton) care descrie o mulțime de clase similare.

Clasa template indică compilatorului ca definiția clasei poate acomoda unul sau mai multe tipuri de date care se vor specifica la momentul utilizării (creare de instanțe)

La momentul utilizării compilatorul crează o clasa actuală înlocuind parametrii cu tipul actual furnizat.

```
template<typename Element>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE(Element 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
     */
    Element deleteElem(int poz);

    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    Element 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:
    Element *elems;
    int capacity;
    int size;
};
```

Lista implementată folosind șabloane (template)

```
template<typename Element>
class DynamicArray3 {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element*/
    void addE(Element 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*/
    Element deleteElem(int poz);
    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)*/
    Element get(int poz);
    ....
private:
    Element *elems;
    int capacity;
    int size;
}
/**
 * Add an element to the dynamic array
 * r - is a rational number
 */
template<typename Element>
void DynamicArray3<Element>::addE(Element r) {
    ensureCapacity(size + 1);
    elems[size] = r;
    size++;
}
/**
 * Access the element from a given position
 * poz - the position (poz>=0;poz<size)
 */
template<typename Element>
Element DynamicArray3<Element>::get(int poz) {
    return elems[poz];
}
template<typename Element>
void DynamicArray3<Element>::set(int poz, Element el) {
    elems[poz] = el;
}
}
```


Instanțiere DynamicArray pentru diferite tipuri de date

```
void testAddDouble() {  
  
    DynamicArray3<double> ar1;  
    ar1.addE(1.3);  
    double elem = ar1.get(0);  
    assert(elem==1.3);  
    assert(ar1.getSize()==1);  
  
    ar1.addE(2.5);  
    elem = ar1.get(1);  
    assert(elem==2.5);  
    assert(ar1.getSize()==2);  
}  
  
void testAddIntParam2() {  
  
    DynamicArray3<int> ar1;  
    assert(ar1.getSize()==0);  
    ar1.addE(1);  
    int elem = ar1.get(0);  
    assert(elem==1);  
    assert(ar1.getSize()==1);  
  
    ar1.addE(2);  
    elem = ar1.get(1);  
    assert(elem==2);  
    assert(ar1.getSize()==2);  
}  
  
void testRationalAdd() {  
    DynamicArray3<Rational> ar1;  
    Rational r1(1, 1);  
    ar1.addE(r1);  
    Rational elem = ar1.get(0);  
    assert(elem.getUp()==1);  
    assert(elem.getDown()==1);  
    assert(ar1.getSize()==1);  
  
    Rational r2(2, 3);  
    ar1.addE(r2);  
    elem = ar1.get(1);  
    assert(elem.getUp()==2);  
    assert(elem.getDown()==3);  
    assert(ar1.getSize()==2);  
}
```

Elemente statice (metode și variabile membre)

Atributele declarate **static** aparțin clasei nu instanțelor

Ele descriu caracteristici ale clasei nu fac parte din starea obiectelor

Pot fi privite ca și variabile globale definite în interiorul clasei

- Pot fi accesate de toate instanțele
- pot fi accesate folosind operatorul scope `::`

```
/**                                     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;
};
```

Diagrame UML .

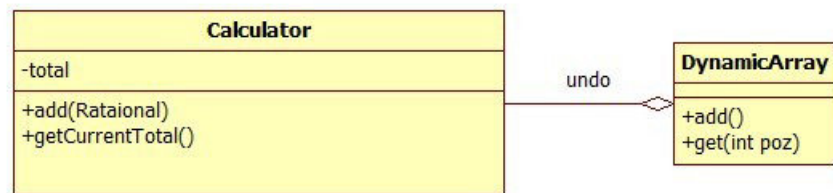
- UML (Unified Modeling Language)
- Standard folosit la scară largă pentru a specifica, vizualiza, construi, documenta sisteme software
- Este independent de limbajul de programare
- Permite modelarea sistemelor soft, oferă un limbaj comun

UML Diagrame de clase

descrie clasele din program și relațiile între ele

Conține:

- numele clasei
- variable membre – (nume + tip)
- metode - (nume + parametri + tipul returnat)
- modificatori de acces
 - membrii privați cu “-”,
 - membrii publici cu “+”



Moștenire

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

Dacă A și B sunt două clase unde B moșteneste 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 membrii (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, descendent 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 in clasa B

```
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 in clasa A și în clase derivate din A).

Modificatori de access

Definesc reguli de access 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. Să ne asigurăm că obiectul este inițializat corect.
- Similar și pentru destructor. Trebuie să ne asigurăm că 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";  
}
```

Initializare.

Cand definim constructorul putem initializa 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 memre din clasa de bază
- se alocă memorie pentru variabile membre noi din clasa derivată
- se apelează constructorul clasei de bază pentru a inițializa attributele 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 bidirecțional

Tipuri de relații de asociere

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

Are (has a):

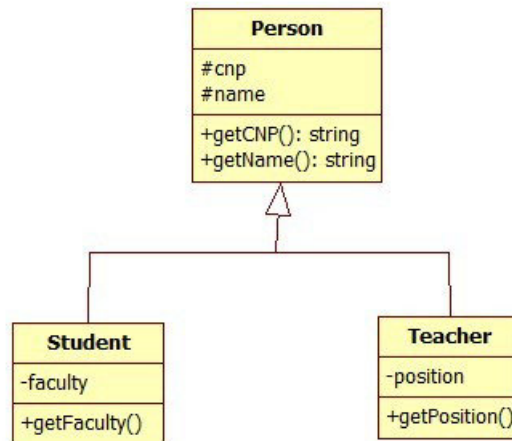
- Orice obiect de tip A are un obiect B.
- Saleltem are un Product. Persoana are nume (string)
- in cod apare ca și o variabilă membră

Este ca și (is a ,is like a):

- Orice instanța 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 derivata descriem ce este specific clasei derivate, ce diferă față de clasa de bază

Suprascriere (overwrite) \neq Supraîncărcare (overload)

<pre>string Person::toString() { return "Person:" + cnp + " " + name; }</pre>
<pre>string Person::toString(string prefix) { return prefix + cnp + " " + name; }</pre>

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 diferi de tipul declarat

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

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

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

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

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

```

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 sa 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) numit 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 dinamică în C++ funcționează doar pentru referințe și pointeri
- În C++ doar metodele virtuale folosesc legarea dinamică

Metode virtuale.

Legarea dinamică în c++: Folsind 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 sa știm tipul exact
- Destructorul poate fi virtual (este chiar recomandat sa fie când avem hierarhii 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 moștenească de la mai multe clase

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

Clasa moștenește din toate clasele de bază toate atributele.

Moștenirea multiplă 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

Funcții pur virtuale

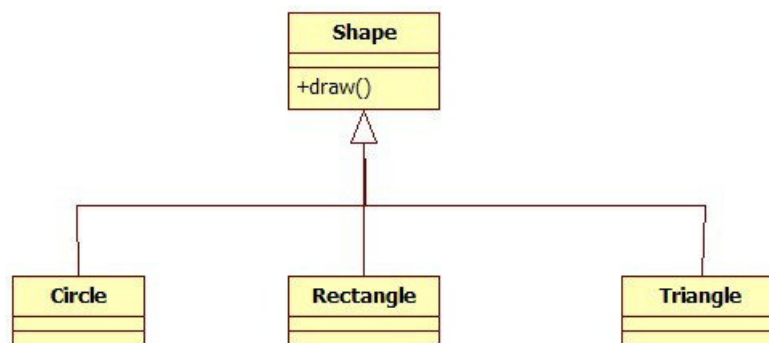
Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei). Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o să definească metoda.

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

=0 indică faptul ca nu există implementare pentru această metodă în clasă.

Clasele care au metode pur virtuale nu se pot instanția

Shape este o clasă abstractă - definește 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 atribute 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 mosteneste interfața publică a clasei abstracte
- clasa suprascrive metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstractă
- putem avea instanțe

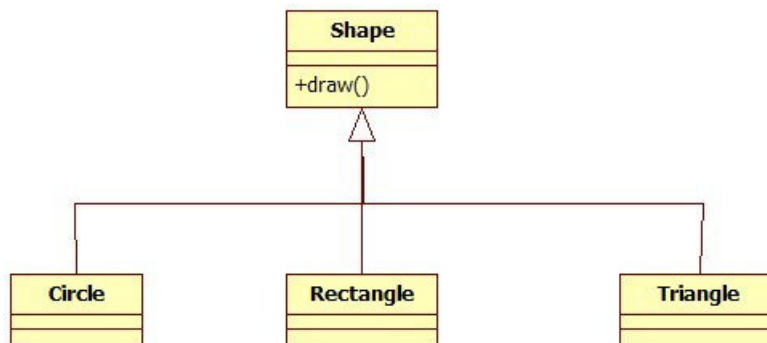
Funcții pur virtuale

Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei). Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o să definească metoda.

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

=0 indică faptul ca nu există implementare pentru această metodă în clasă. Clasele care au metode pur virtuale nu se pot instanția

Shape este o clasă abstractă - definește 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 atribute 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

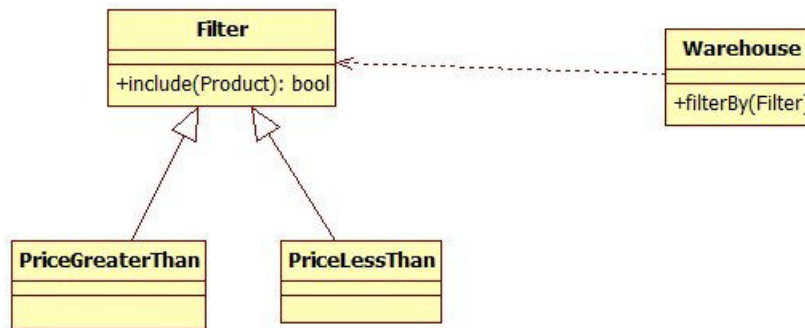
- O clasă derivată dintr-o clasă abstractă mosteneste interfața publică a clasei abstracte
- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstracta
- putem avea instanțe

Moștenire. Polimorfism

Avantaje:

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

Exemplu : Filrare produse



```
DynamicArray3<Product*>* Warehouse::filterByPrice(double price) {
    DynamicArray3<Product*>* rez = new DynamicArray3<Product*>();

    DynamicArray3<Product*>* all = repo->getAllProds();

    for (int i = 0; i < all->getSize(); i++) {
        Product* p = all->get(i);
        if (p->getPrice() > price) {
            //make a copy of the product
            rez->addE(new Product(*p));
        }
    }
    return rez;
}
```

```
DynamicArray3<Product*>* Warehouse::filterByPriceLT(double price) {
    SmallerThanPrice* f = new SmallerThanPrice(price);
    return filterBy(f);
}
```

```
DynamicArray3<Product*>* Warehouse::filterByPriceGT(double price) {
    GreaterThanPrice* f = new GreaterThanPrice(price);
    return filterBy(f);
}
```

```
DynamicArray3<Product*>* Warehouse::filterBy(Filter* filter) {
    DynamicArray3<Product*>* rez = new DynamicArray3<Product*>();
    DynamicArray3<Product*>* all = repo->getAllProds();
    for (int i = 0; i < all->getSize(); i++) {
        Product* p = all->get(i);
        //polimorphic method invocation
        if (!filter->include(p)) {
            //make a copy of the product
            rez->addE(new Product(*p));
        }
    }
    return rez;
}
```


Operații de intrare/ieșire

IO (Input/Output) în C

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

- Funcțiile 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 diferi (ANSI standard)
- pentru fiecare clasă nouă, ar trebui să adăugăm o versiune nouă (supraîncărcare) de funcții **printf()** and **scanf()** și variantele pentru lucru cu fișiere, șiruri de caractere
- Metodele supraîncărcate au același nume dar o listă de parametri 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 creată 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 în 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 în C++ pentru operații IO

Biblioteca standard de intrări/ieșiri (iostream) conține:

Clase template

O hierarhie 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 **streamsize** (reprezintă poziții, offset, dimensiuni)

Manipulatori

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

Flux - Stream

Noțiunea de flux este o abstractizare, reprezintă 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

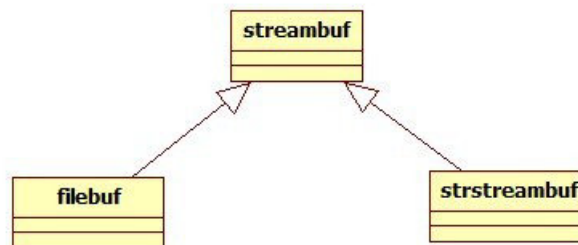
buffer este o zonă de memorie care este un intermediar între stream și dispozitiv.

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

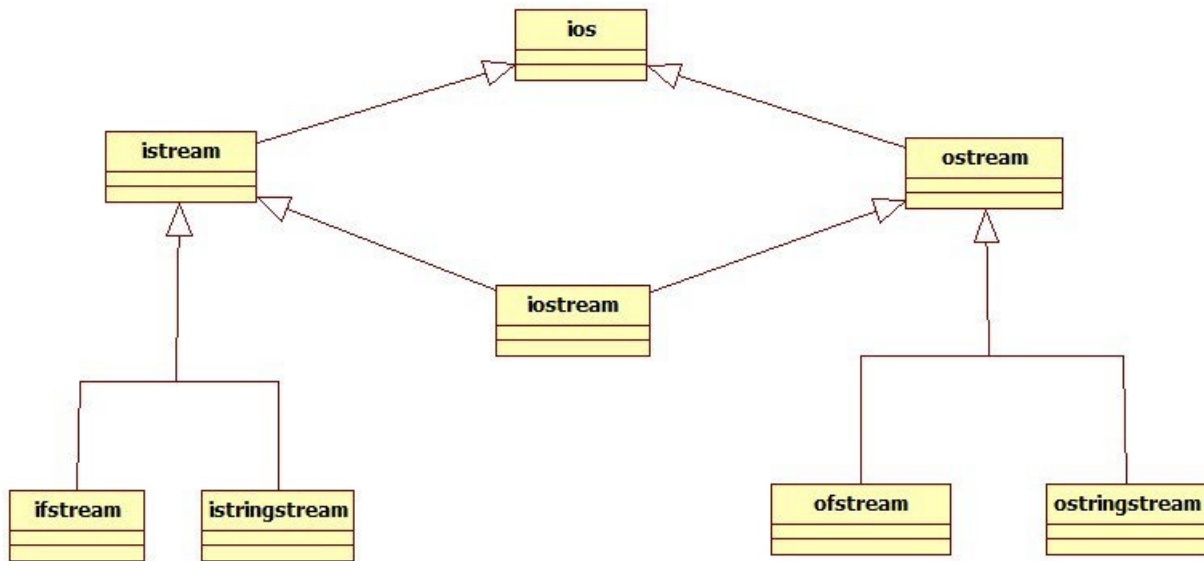
Când bufferul este golit (flush) , toate datele se trimit la dispozitiv (daca era un strim de ieșire). Procesul prin care conținutul bufferului este trimis la dispozitiv se numeste sincronizare și se întâmplă dacă:

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

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



Hierarhie de clase din biblioteca standard IO C++



Fișiere header din IOStream

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

- <ios> formatare , streambuffer.
- <istream> 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.

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 sa avem un obiect de tip ostream (sau derivat din ostream). Pentru a scrie pe ieșire standard (consolă) se folosește **cout** (declarat in modulul iostream)
- pe dreapta putem avea o expresie.
- Operatorul este supraîncărcat pentru tipurile standard, pentru tipurile noi programatorul trebuie sa 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 înlănțui operații de inserție, evaluarea se face în ordinea inversă scrierii. Înlănțuirea funcționează fiindca 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;
}
```

Supraîncărcare 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).

```
class Product {
public:
    Product(int code, string desc,
double price)
    ~Product();
    double getCode() const {
        return code;
    }
    double getPrice() const {
        return price;
    }
    const string& getDescription()
const {
        return description;
    }

    friend ostream& operator<<
(ostream& stream, const Product&
prod);

private:
    int code;
    string description;
    double price;
};
```

```
ostream& operator<<(ostream&
stream, const Product& prod) {
    stream << prod.code << " ";
    stream << prod.description << "
";
    stream << prod.price;
    return stream;
}
```

```
void testStandardOutputUserType() {
    Product p = Product(1, "prod", 21.1);
    cout << p << "\n";
    Product p2 = Product(2, "prod2", 2.4);
    cout << p2 << "\n";
}
```

Formatare scriere

<code>width(int x)</code>	Numarul minim de caractere pentru scrierea următoare
<code>fill(char x)</code>	Caracter folosit pentru a umple spațiu dacă e nevoie să completeze cu caractere (lungime mai mică decât cel setat folond <code>width</code>).
<code>precision(int x)</code>	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;
}
```

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);  
}
```

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);
```

Fișiere

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

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

<fstream.h>

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

Putem atașa fișierul de stream folosind constructorul sau metoda **open**

După ce am terminat operațiile de IO pe fișier trebuie sa închidem (deasociem) 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 daca 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. Conținutul fișierului este șters la deschidere.
- Dacă nu există “test.out”: se crează, 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 implementari 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::binary
Mod binar.

ios::ate Se poziționează la sfârșitul fișierului.

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

ios::append
Toate operațiile de scriere se efectuează la sfârșitul fișierului, se adaugă la conținutul existent. Poate fi folosit doar pe stream-uri deschise pentru scriere.

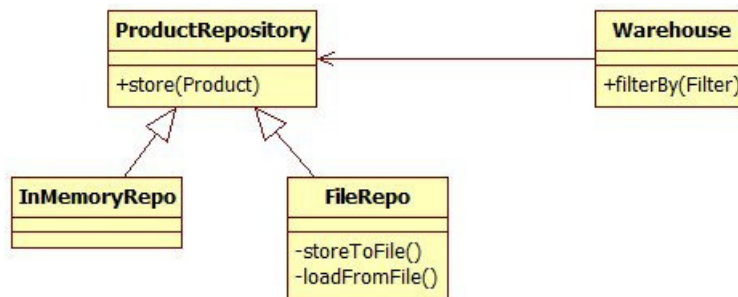
ios::trunc Sterge conținutul existent.

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

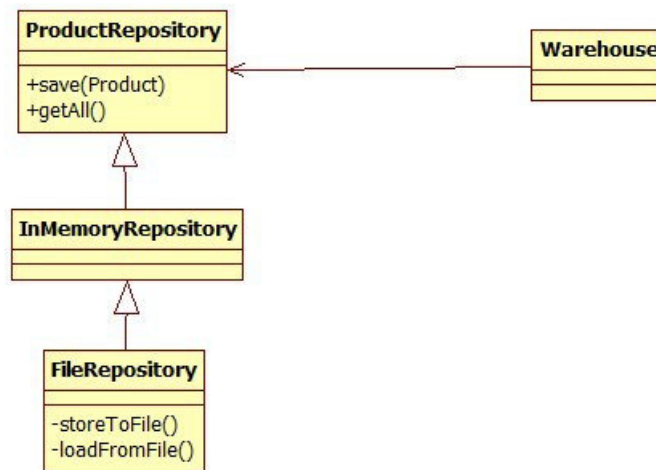
Citire/scriere obiecte

```
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



Tratarea excepțiilor în c++

excepții - situații anormale ce apar în timpul execuției

tratarea excepțiilor - mod organizat de a gestiona situațiile excepționale 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 execută în cazul în care apare o excepție (tratează excepția).
- Instrucțiunea **throw** mecanism prin care putem arunca (genera excepții) pentru a semnala codului client apariția unei probleme.

```
void testTryCatch() {  
    // some code  
    try {  
        //code that may throw an exception  
        throw 12;  
        //code  
    } catch (int error) {  
        //error handling code  
        cout << "Error ocurred." << error;  
    }  
}
```

Tratarea excepțiilor

- Codul care susceptibil de a arunca excepție se pune într-un bloc de `try` .
- Adăugăm unu sau mai multe secțiuni de **catch** . Blocul de instrucțiuni din interiorul blocului `catch` este responsabil sa trateze excepția apărută.
- Dacă codul din interiorul blocului `try` (sau orice cod apelat de acesta) aruncă excepție, se transferă execuția la clauza `catch` corespunzătoare tipului excepției apărute. (exception handler)

```
void testTryCatchFlow(bool throwEx) {
    // some code
    try {
        cout << "code before the exception" << endl;
        if (throwEx) {
            cout << "throw (raise) exception" << endl;
            throw 12;
        }
        cout << "code after the exception" << endl;
    } catch (int error) {
        cout << "Error handling code " << endl;
    }
}

testTryCatchFlow(0);
testTryCatchFlow(1);
```

- Clauza `catch` nu trebuie neapărat sa fie în același metodă unde se aruncă excepția. Excepția se propagă.
- Când se aruncă o excepție, se caută cel mai apropiat bloc de **catch** care poate trata excepția ("unwinding the stack").
- Dacă nu avem clauză **catch** in funcția în care a apărut excepția, se caută clauza **catch** în funcția care a apelat funcția .
- Căutarea continuă pe stack până se găsește o clauză **catch** potrivită. Dacă excepția nu se tratează (nu există o clauză **catch** potrivită) programul se oprește semnalând eroarea apărută.

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).
- Este recomandat să se creeze clase pentru diferite tipuri de excepții care apar în aplicație
- Obiectul excepție este transmis ca referință sau pointer (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, Product* product) {
    if (quantity < 0) {
        throw ValidationError("Quantity must be positive");
    }
    saleItems.push_back(new SaleItem(quantity, product));
}
```

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

Specificații

Putem include lista de excepții pe care o metodă le poate arunca în declarația funcției folosind specificațiile de excepții

Dacă nu includem specificațiile de excepții pentru o metoda asta înseamnă că metoda poate arunca orice tip excepție

```
//can throw any type of exceptions
void f1() {
}
```

```
//can not throw exceptions
void f3() throw (){
}
```

```
/**
 * create ,validate and store a product
 * code - product code
 * desc - product description
 * price - product price
 * throw ValidatorException if the product is invalid
 * throw Repository exception if the code is already used by a product
 */
Product Warehouse::addProduct(int code, string desc, double price)
    throw (ValidatorException, RepositoryException) {
    //create product
    Product p(code, desc, price);
    //validate product
    validator->validate(p);
    //store product
    repo->store(p);
    return p;
}
```

Excepții - Avantaje

Metode de gestiune a situațiilor excepționale:

- folosind coduri de eroare (funcția returnează un cod de eroare dacă a apărut o problemă)
- folosind flaguri de control (în caz de erori se setează flaguri, care pot fi verificate ulterior de codul apelant)
- folosind mecanismul de excepții

Avantajele mecanismului de excepții:

- putem separa codul de gestiune în caz de eroare (error handling code) de fluxul normal de execuție
- multiple tipuri de erori se pot trata cu metodă. Clauza **catch** se selectează conform tipului excepției aruncate (moștenire). Putem folosi (...) pentru a trata orice tip de excepție (nerecomandat)
- Excepțiile nu se pot ignora. Dacă nu tratăm excepția (nu există clauză catch corespunzătoare) programul se termină.
- Funcțiile au mai puține parametri, nu se pun diferite valori de retur. Funcția e mai ușor de înțeles și folosit
- Orice informații se pot transmite folosind excepțiile. Informația se propagă de la locul unde a apărut excepția până la clauza catch care tratează eroarea.
- Dacă într-o metodă nu putem trata excepția putem ignora și acesta se propagă la metoda apelantă. Se propagă până în locul unde putem lua acțiuni (să tratăm eroarea)

Când să aruncăm excepții

- Se aruncă excepție dacă metoda nu poate realiza operația promisă
- Folosim excepții pentru a semnală erori neașteptate
- Putem folosi excepții pentru a semnală încălcarea condițiilor.
 - În mod normal cel care apelează metoda este responsabil să furnizeze parametri actuali care satisfac condiția.
- Este de evitat folosirea prea frecventă (în alte scopuri decât semnalarea unei situații excepționale) de excepții fiindcă excepțiile frecvente fac codul mai greu de înțeles (execuția sare de la flux normal la codul de tratare a excepției)
- Aruncarea de excepție cu singurul scop de a schimba fluxul de execuție nu este recomandat.
- Constructorul / destructorul nu ar trebui să arunce excepții

Clauze catch

```
try {
    Product p = wh->addProduct(code, desc, price);
    cout << "product " << p.getDescription() << " added." << endl;
} catch (RepositoryException& ex) {
    cout << "Error on store:" << ex.getMsg() << endl;
} catch (ValidatorException& ex) {
    cout << "Error on validate:" << ex.getMsg() << endl;
} catch (...) {
    cout << "unknwn exception";
}
```

- Dacă se aruncă o excepție în codul din blocul **try** se executa clauza catch conform tipului excepției aruncate
- Se execută doar una dintre clauzele **catch**
- Se executa clauza catch care corespunde tipului – excepția este de același tip sau este derivat din tipul indicat în clauza **catch**
- (...) corespunde oricărui tip. Orice tip de excepție se aruncă această clauză catch corespunde

Ierarhii de clase exceptii

- Excepțiile permit izolarea codului care gestionează eroarea apărută. De asemenea cu o organizare potrivită se poate reduce volumul de cod scris pentru tratarea excepțiilor din aplicație
- Numărul de clauze catch nu ar trebui sa crească odată cu evoluția programului.
- Clasele folosite pentru excepții trebuie organizate în ierarhii de clase, asta permite reducerea numărului de clauze catch.
- Un grup de tipuri de excepții poate fi tratat uniform, daca între aceste tipuri există relația de moștenire.
- Folosind ierarhiile de excepții putem beneficia și de polimorfism

```
try {  
    Product p = wh->addProduct(code, desc, price);  
    cout << "product " << p.getDescription() << " added." << endl;  
} catch (WarehouseException& ex) {  
    cout << "Error on store:" << ex.getMsg() << endl;  
}
```

catch (WarehouseException& ex) – se executa daca apare excepția WarehouseException sau clase derivate din WarehouseException (ValidatorException, RepositoryException)

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;  
}
```

Qt Toolkit

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

Folosind QT putem crea interfețe grafice utilizator. Codul odată scris poate fi compilat 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, MSVS)
- 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 limbaje: 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: <http://qt-project.org/>

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 părți din QT
- **Qt Linguist** – suport pentru aplicații care funcționează în diferite limbi (internaționalizare)
- **qmake** – aplicație folosit în procesul de compilare
- **Qt Eclipse integration** – plugin eclipse care ajută la crearea de aplicații QT folosind eclipse

Instalare QT, eclipse plugin

Este necesar: Eclipse CDC (MinGW) funcțional

1) Download /instalare Qt Library

<http://qt-project.org/downloads> → for windows **Qt libraries 4.8.1 for Windows (minGW 4.4, 319 MB)**

Conține:

- Qt SDK (library)
- Qt Designer, Assistant, Linguist, Samples and demos

2) Download / instalare plugin eclipse pentru Qt

Oferă:

- Wizards pentru a crea proiecte Qt
- Configurare build automată
- Un editor QT integrat în eclipse (QT Designer)

OBS:

- pluginul nu funcționează cu Eclipse Indigo 64 bit (folosiți 32 bit Indigo or Helios)
- Qt installer poate da un mesaj warning la anumite versiuni de MinGW (diferit de 3.13). Dacă aveți versiune mai nouă ignorați mesajul și continuați instalarea.

Qt Hello World

Creați un QT Eclipse Project: File → New → Qt GUI Project

Adăugați în main:

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

Project → Build Project

Rulați aplicația

QApplication

Clasa QApplication gestioneaza fluxul de evenimente și setarile 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**();

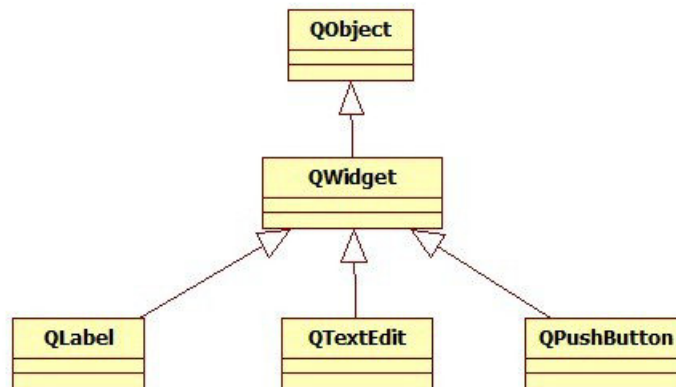
- pornește procesarea de evenimente din QApplication (event loop). În timp ce rulează aplicația evenimente sunt generate și trimise către componentele grafice.

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 defapt o fereastră separată .

Ferestrele 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. For this use 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 opțional o iconiță. 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

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

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

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

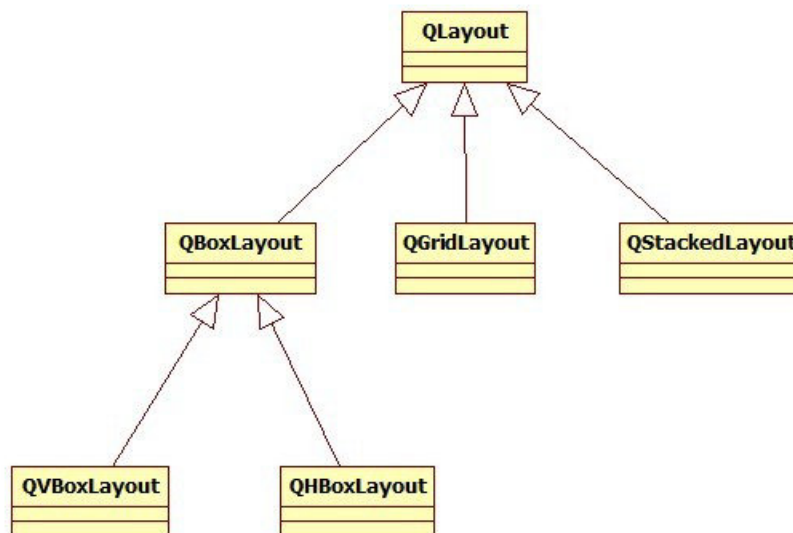
QListWidget

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

<pre>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();</pre>	<pre>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();</pre>
--	---

GUI putem compune multiple componente care folosesc diferite strategii de aranjare pentru a crea interfața 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* btnsL = new QHBoxLayout;  
btns->setLayout(btnsL);  
QPushButton* store = new QPushButton("&Store");  
btnsL->addWidget(store);  
btnsL->addStretch();  
QPushButton* close = new QPushButton("&Close");  
btnsL->addWidget(close);
```

Layout manager o să adauge spațiu între butonul Store și Close

Layout management

Cum construim interfețele grafice:

- instanțiem 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 reanjarea componentelor în caz de redimensionare a componentei
- setează valori implicite 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 adapteaza componenta pentru a evita trunchiera de text).
- Dacă adugăm/ștergem componente restul componentelor sunt rearanjate automat (similar și pentru *show()* *hide()* pentru o componentă)

Poziționare 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://qt-project.org/doc/qt-4.8/>
- Pentru orice clasă găsiți descrieri detaliate pentru metode, atribute, semnale, sloturi

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, apasăm 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 semnătura funcției, parametrii, etc. Are nevoie de referința la metoda callback).

Signal. Slot.

- Semnalul (**signal**) este emis la apariția unui eveniment (ex.: clicked())
- 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 emiterea semnalului slotul este automat executat

```
QPushButton *btn = new QPushButton("&Close");
QObject::connect(btn,SIGNAL(clicked()),&app,SLOT(quit()));
btn->show();
```

- **Slot** poate fi folosit pentru a reacționa la semnale, dar ele sunt defapt metode normale.
- 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)
- Componentel Qt's au și sloturi predefinite
- În general programatorul extinde componentele (moștenește din clasele QWidget) și adaugă sloturi noi care se conectează la semnale

Conectarea semnalelor cu sloturi

Folosind metoda `QObject::connect` și macrourile `SIGNAL` și `SLOT` 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,SIGNAL(clicked()),&a,SLOT(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 emiterii semnalului se vor apela sloturile în ordinea în care au fost conectate

Există o corespondență între semnatura semnalului și semnatura slotului (parametrii trebuie să corespundă).

Slotul poate avea mai puține parametrii, parametrii de la semnal care nu au corespondență la slot se vor ignora.

Conectarea 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,
SIGNAL(valueChanged(int)),slAge,SLOT(setValue(int)));
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge,
SIGNAL(valueChanged(int)),spAge,SLOT(setValue(int)));
```

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 schimbă (este egal cu ce s-a primit la setValue) astfel se evită ciclul infinit de semnale

Componente definite de utilizator

- Se crează clase separate pentru interfața grafică utilizator
- componentel 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 QWidget(10, 10, this);
setCentralWidget(middle);
```

- Status bar - în partea de jos

```
statusBar()->showMessage("Status Message ....");
```


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 semnale 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

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 fișer 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 sa apară la începutul definiției clasei. El este necesar în orice clasă unde vrem să adăugăm 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 semnale, 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 și sloturi definite de utilizator

Pentru a crea sloturi se folosește cuvântul rezervat *slots* (este defapt un macro). Qt (moc utility) are nevoie de această informație pentru a genera meta-informații despre sloturile disponibile ale componentei

Slotul în sine este doar o metodă obișnuită .

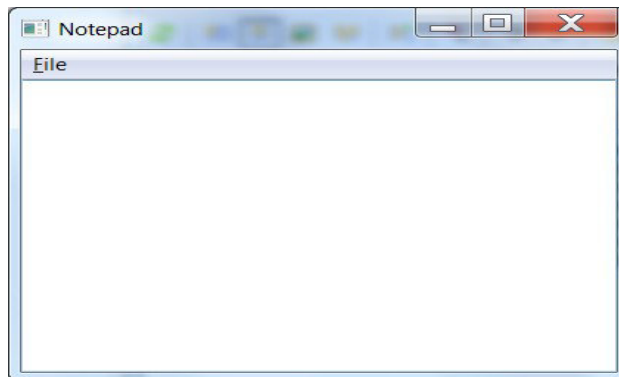
```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit();
```

```
void Notepad::save()
{
    ...
}
```

Notepad



```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit2();

    openAction = new QAction(tr("&Load"), this);
    saveAction = new QAction(tr("&Save"), this);
    exitAction = new QAction(tr("E&xit"), this);

    connect(openAction, SIGNAL(triggered()), this, SLOT(open()));
    connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));
    connect(exitAction, SIGNAL(triggered()), this, SLOT(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();
    }
}
```

Semnale proprii

Folosind macroul *signals* se pot declara semnale proprii pentru componentele pe care le creăm.

```
private signals:  
    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

QtDesigner din Eclipse

Proiect Eclipse Qt GUI

File ->New->Qt GUI Project

- generează structura proiectului qt (setează modulele incluse, directoare , etc)
- .ui – fisier ce contine descrierea interfeței grafice
 - UIC (user interface copiller) utilitar ce transformă fișierul .ui in fișier c++ care construiește interfața grafică (ui_<name>.h)
- crează o componenta 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();
}
```

Important – Versiunea curenta de Qt eclipse plugin are probleme când folosim anumite clase din biblioteca standard c++ (cout,iostream, etc). Compilatorul interactiv raportează erori în mod greșit.

Rezolvare - Deactivați analizorul din timpul editarii - Code Analysis (Project->Properties->c/c++ general->Code Analyses deselectați toate).

Erorile sunt raportate corect după ce compilați proiectul (nu în timp ce scrieți codul)

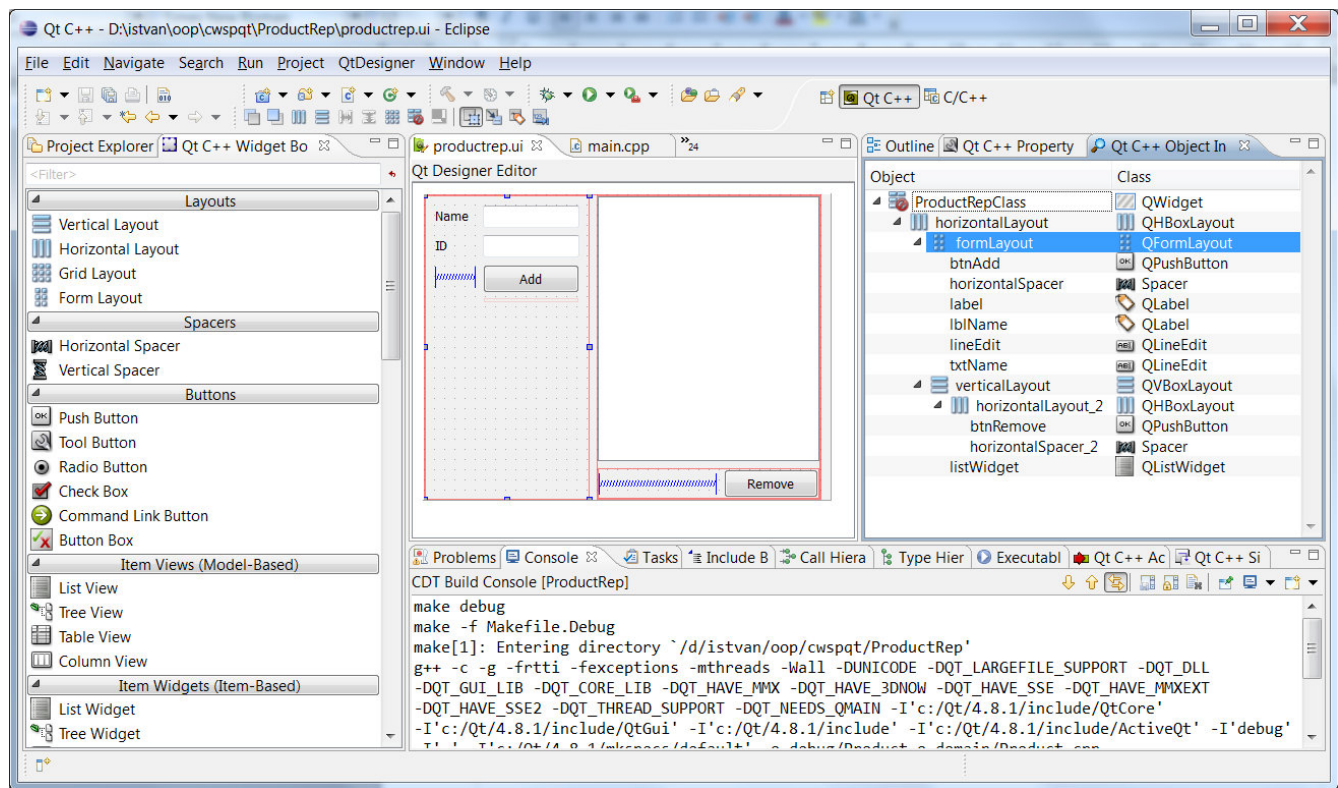
Creare de interfețe grafice vizual (folosind drag & drop)

Pluginul Eclipse pentru Qt permite crearea 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

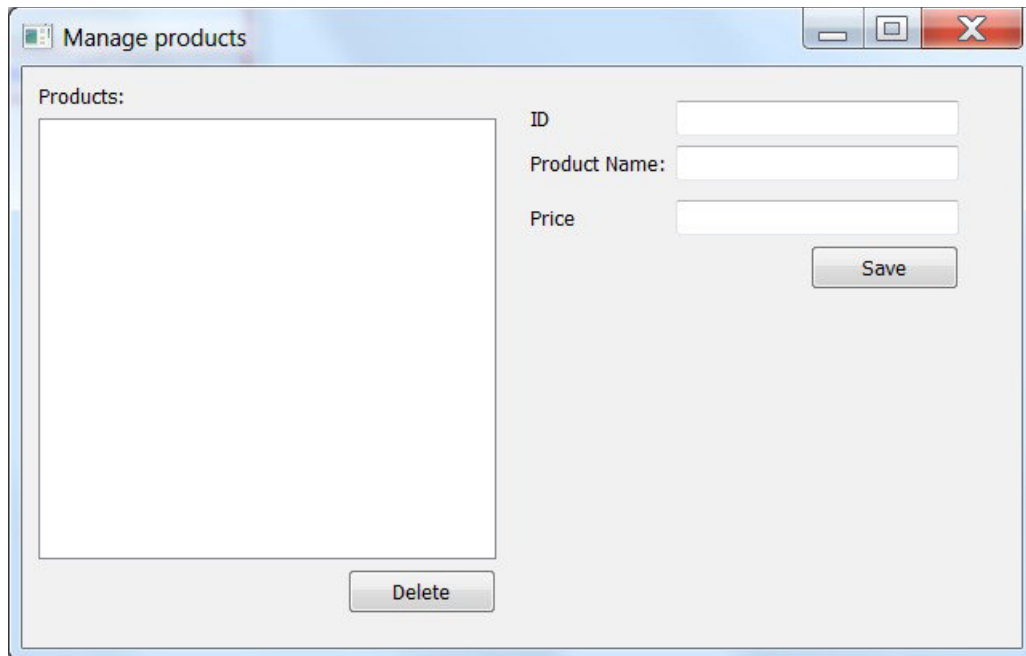
Eclipse Qt editor/views:

- 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



Master detail – Product

CRUD (Create Read Update Delete) pentru Product



The image shows a screenshot of a software application window titled "Manage products". The window has a standard Windows-style title bar with minimize, maximize, and close buttons. The main content area is divided into two sections. On the left, there is a large empty rectangular box labeled "Products:". On the right, there is a form with three input fields: "ID", "Product Name:", and "Price". Below the "Price" field is a "Save" button. At the bottom center of the window, there is a "Delete" button.

Field	Value
ID	
Product Name:	
Price	

Buttons: Save, Delete

Sloturi definite de utilizator

```
class testSlots: public
QWidget {
Q_OBJECT

public:
    testSlots(Warehouse*
wh, QWidget *parent = 0);
    ~testSlots();

private:
    Ui::testSlotsClass ui;
    Warehouse* wh;
    void connectSS();
    void reloadList();

private slots:
    void save();
    void productSelected();
};

/**
 * Save products
 */
void testSlots::save() {
    int id = ui.txtID->text().toInt();
    double price = ui.txtPrice->text().toDouble();
    string desc = ui.txtName->text().toStdString();
    try {
        wh->addProduct(id, desc, price);
        reloadList();
        QMessageBox::information(this, "Info", "Product
saved...");
    } catch (WarehouseException ex) {
        QMessageBox::critical(this, "Error",
QString::fromStdString(ex.getMsg()));
    }
}
```

QString

- Clasă pentru șiruri de caractere (Unicode), similar cu string din biblioteca standard c++
- apare frecvent când lucrăm cu componente din Qt

Create QString

```
QString s1 = "Hello";  
QString s2("World");
```

QString and string (STL)

```
string str = "Hello";  
QString qStr = QString::fromStdString(str);  
string str2 = qStr.toStdString();
```

Numbers and QString

```
QString s3 = QString::number(2);  
QString s4 = QString::number(2.5);  
QString s5 = "2";  
int i = s5.toInt();  
double d = s5.toDouble();
```

QListWidget – populare listă, semnalul itemSelectionChanged()

```
private slots:
    void save();
    void productSelected();

/**
 * Save product
 */
void testSlots::save() {
    int id = ui.txtID->text().toInt();
    double price = ui.txtPrice->text().toDouble();
    try {
        wh->addProduct(id, ui.txtName->text().toStdString(), price);
        reloadList();
        QMessageBox::information(this, "Info", "Product saved...");
    } catch (WarehouseException ex) {
        QMessageBox::critical(this, "Error",
            QString::fromStdString(ex.getMsg()));
    }
}

/**
 * Load the products into the list
 */
void testSlots::reloadList() {
    ui.lstProducts->clear();
    DynamicArray<Product*> all = wh->getAll();
    for (int i = 0; i < wh->getNrProducts(); i++) {
        string desc = all.get(i)->getDescription();
        QListWidgetItem *item = new QListWidgetItem(
            QString::fromStdString(desc), ui.lstProducts);
        item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)-
>getCode()));
    }
}
```

Clasele Qt ItemView

QListWidget, QTableWidget , QTreeWidget

Componentele se populează, adaugând toate elementele de la început (items: QListWidgetItem, QTableWidgetItem, QTreeWidgetItem).

Afișarea, cautarea, editarea sunt efectuate direct asupra datelor cu care este populat componenta

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 daca 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

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)

the **controller:** mediează între model și view

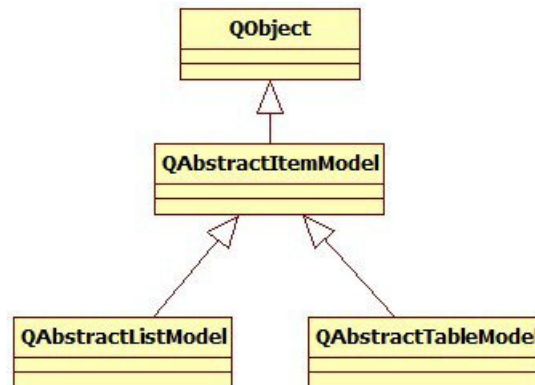
- transformă acțiunile utilizator în cereri (de navigare, de editare date)
- diferit de GRASP Controller

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 șablonul MVC (Model-View-Controller), but dar în loc de controller, Qt folosește o altă abstractizare numită **delegate**
- **delegate** – oferă control asupra modului de prezentare a datelor și asupra editării
- Qt oferă implementari 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

Creare de modele noi

- Se crează 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 atasat un numar de elemente cu roluri diferite (DisplayRole, BackgroundRole, UserRole, etc)

```
void testSlots::reloadList() {
    ui.lstProducts->clear();
    DynamicArray<Product*> all = wh->getAll();
    for (int i = 0; i < wh->getNrProducts(); i++) {
        string desc = all.get(i)->getDescription();
        QListWidgetItem *item = new QListWidgetItem(
            QString::fromStdString(desc), ui.lstProducts);
        item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)->getCode()));
    }
}

/**
 * Load the selected product into the detail panel
 */
void testSlots::productSelected() {
    QList<QListWidgetItem*> sel = ui.lstProducts->selectedItems();
    if (sel.size() == 0) {
        return;
    }
    QVariant idV = sel.first()->data(Qt::UserRole);
    int id = idV.toInt();
    const Product *p = wh->getByCode(id);
    ui.txtID->setText(QString::number(id));
    ui.txtName->setText(QString::fromStdString(p->getDescription()));
    ui.txtPrice->setText(QString::number(p->getPrice()));
    QMessageBox::information(this, "Info", "Product selected...");
}
```


Creare de modele noi

```
class MyTableModel: public QAbstractTableModel {
public:
    MyTableModel(QObject *parent);
    /**
     * number of rows
     */
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    /**
     * number of columns
     */
    int columnCount(const QModelIndex &parent = QModelIndex()) const;
    /**
     * Value at a given position
     */
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
};

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();
}
```

Putem crea modele care incarcă doar datele care sunt efectiv necesare (sunt vizibile)

Modele predefinite

Qt oferă modele predefinite:

- **QStringListModel** – Lucrează cu o listă de stringuri
- **StandardItemModel** - Date ierarhice
- **QDirModel** - System de fișiere
- **QSqlQueryModel** - SQL result set
- **QSqlTableModel** - SQL table
- **QSqlRelationalTableModel** - SQL table cu chei străine
- **QSortFilterProxyModel** - oferă sortare/filtrare

```
void createTree() {  
    QTreeView *tV = new QTreeView();  
    QDirModel *model = new QDirModel();  
    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();
}
```

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 sa emitem semnalul dataChanged și se actualizează interfața grafică

```
/**
 * Slot invoked by the timer
 */
void MyTableModel::timerTikTak() {
    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 permițând 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 schimbam modelul trebuie să emitem semnalul dataChanged (să ne asigurăm că vederile se actualizează)

Containere cu elemente generice

Creare de containere (liste, multimi, dictionar,etc) care acceptă orice tip de elemente:

- **void ***

```
typedef void* TElem;

class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void add(TElem e);
    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    TElem get(int poz);
};
```

- **Şabloane**

```
template<typename Element>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE(Element r);
    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    Element get(int poz);
};
```


Containere cu elemente generice

Dacă structura de date are nevoie de anumite funcții (egalitate, hashCode, copiere de valori, etc.):

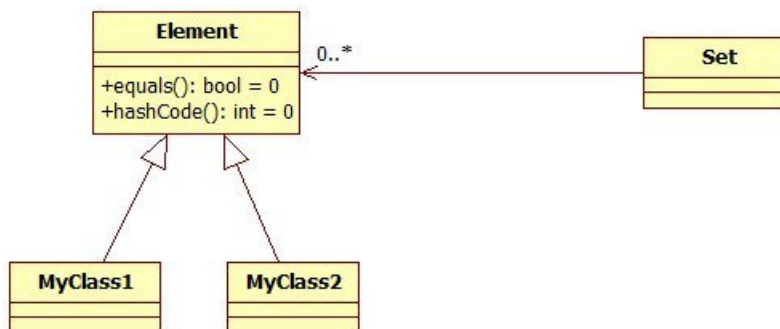
- pointer la funcții

```
typedef elem (*copyPtr)(elem&, elem);  
typedef int (*equalsPtr)(elem, elem);
```

- supraîncărcare operatori

```
class Product {  
public:  
    bool Product::operator==(const Product &other) const {  
        return code == other.code;  
    }  
  
template<typename Element>  
class Set {  
private:  
    Element* elems;  
    int size;  
public:  
    Set();  
    void add(Element el);  
    bool contains(Element el);  
    int card() {  
        return size;  
    }  
};  
template<typename Element>  
Set<Element>::Set() {  
    elems = new Element[100];  
    size = 0;  
}  
  
template<typename Element>  
void Set<Element>::add(Element el) {  
    if (contains(el)) {  
        return;  
    }  
    elems[size] = el;  
    size++;  
}  
template<typename Element>  
bool Set<Element>::contains(Element el) {  
    for (int i = 0; i < size; i++) {  
        if (el == elems[i]) {  
            return true;  
        }  
    }  
    return false;  
}
```

- creăm o clasă de bază abstractă cu metode virtuale



Șabloane de proiectare

- Șabloanele de proiectare descriu obiecte, clase și interacțiuni/relații între ele. Un șablon 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 șablon 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 șabloanele de proiectare și oferă un catalog de șabloane

Tipuri de șabloane de proiectare (după scop):

- **Creționale**
 - descriu modul de creare a obiectelor
 - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **Structurale**
 - se referă la compoziția claselor sau al 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 de descriu un șablon de proiectare

- Numele șablonului
 - descrie sintetic problema rezolvată și soluția
 - face parte din vocabularul programatorului
- Problema
 - Descrie problema și contextul în care putem aplica șablonul.
- 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 șablonului de proiectare.

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 performanța (generic programming)
- STL conține clase pentru:
 - containere
 - iteratori
 - algoritmi
 - function objects
 - allocators

Containers

Un container este o grupare de date în care se pot adauga (insera) si din care se pot sterge (extrage) obiecte. Implementările din STL folosesc șabloane 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 si 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).

Container - 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>**

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 înlănțuită
 - List are performanțe bune la:
 - Ștergere/adăugare de elemente pe orice poziție (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 / complexity

<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 at the either end

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 alocă zone mari de memorie, deque alocă multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Warehouse – folosind vector

```
/**
 * Store the products in memory (in a dynamic array)
 */
class ProductInMemoryRepository: public ProductRepository {
public:
    /**
     * Store a product
     * p - product to be stored
     * throw RepositoryException if a product with the same id already exists
     */
    void store(Product& p) throw (RepositoryException);
    /**
     * Lookup product by code
     * the code of the product
     * return the product with the given code or NULL if the product not found
     */
    const Product* getByCode(int code);
    /**
     * Count the number of products in the repository
     */
    int getNrProducts();

    ProductInMemoryRepository();
    ~ProductInMemoryRepository();
private:
    vector<Product*> prods;
};

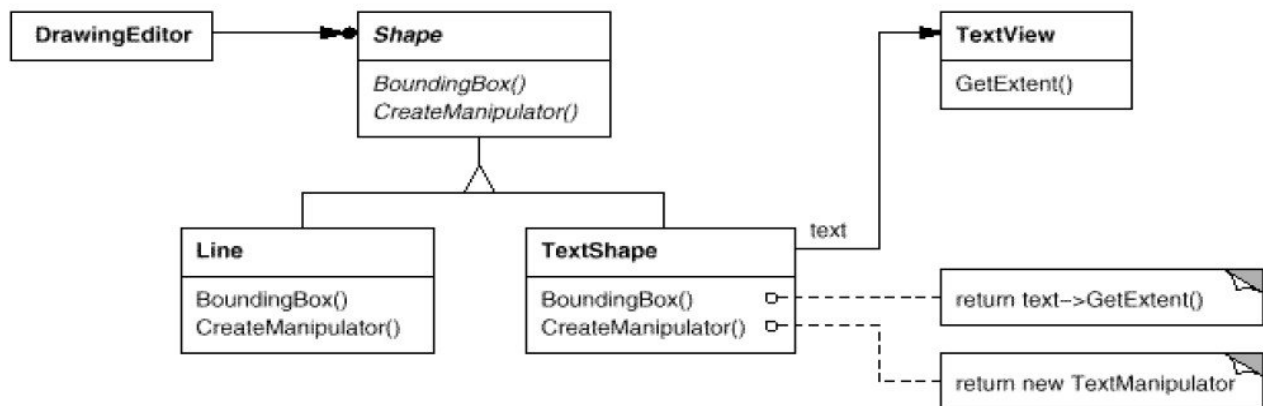
/**
 * Store a product
 * p - product to be stored
 * throw RepositoryException if a product with the same id already exists
 */
void ProductInMemoryRepository::store(Product& p) throw (RepositoryException) {
    //verify if we have a product withe same code
    const Product* aux = getByCode(p.getCode());
    if (aux != NULL) {
        throw RepositoryException("Product with the same code already exists");
    }
    prods.push_back(&p);
}
}
```


Adapter pattern (Wrapper)

Intenția: Adaptarea interfeței unei clase la o interfață potrivită pentru client. Permite claselor să interopereze 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. TextShape adaptează clasa TextView la interfața Shape

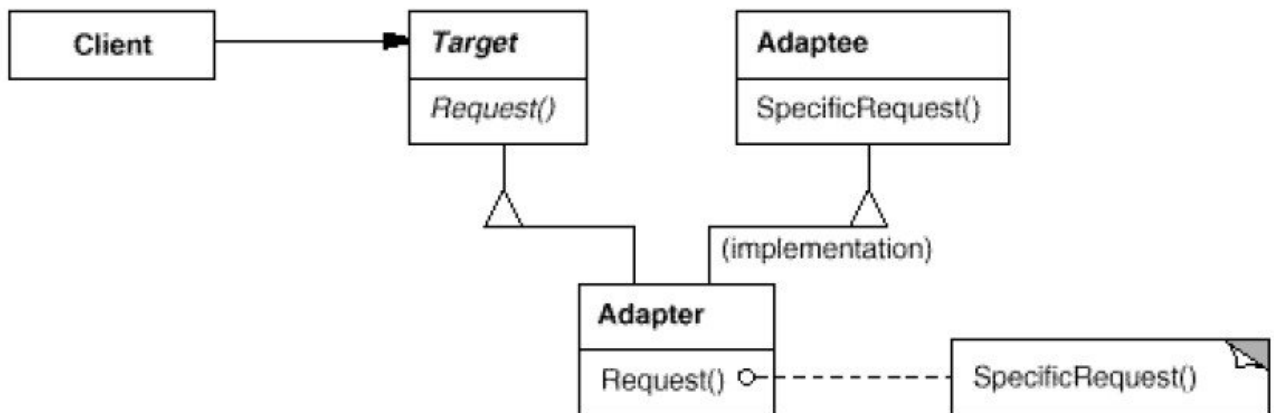


Aplicabilitate:

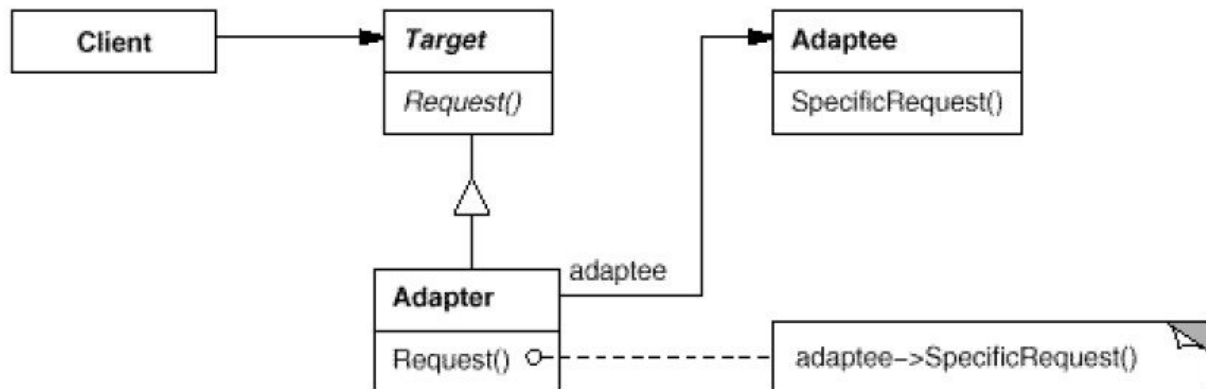
- dorim să folosim o clasă existentă dar interfața clasei nu corespunde cu ceea ce este nevoie
- crearea de clase reutilizabile care cooperează cu alte clase (dar ele nu au interfețe compatibile)

Adapter - structură

Class adapter – folosește moștenire multiplă



Object adapter folosește compoziție



Participants:

- Target: definește interfața de care este nevoie.
- Client: colaborează, folosește obiecte cu interfață Target.
- Adaptee: este clasa care trebuie adaptată. Are interfața diferită de cea ce e are nevoie Client
- Adapter: adaptează Adaptee la interfața Target.

Adapter

Colaborare:

- Clientul apelează metode al 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 clasă 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 șablonul adaptor 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

```
#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();
    }
}
```

```
#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();
    }
}
```

```
#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();
    }
}
```

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 doua elemente care sunt egale
 - se folosește arbore binar de căutare ca și reprezentare internă
- Map
 - dictionar - stochează elemente formate din cheie și valoare
 - nu putem avea chei duplicate
- Bitset
 - container special pentru a stoca biti (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;
}
```

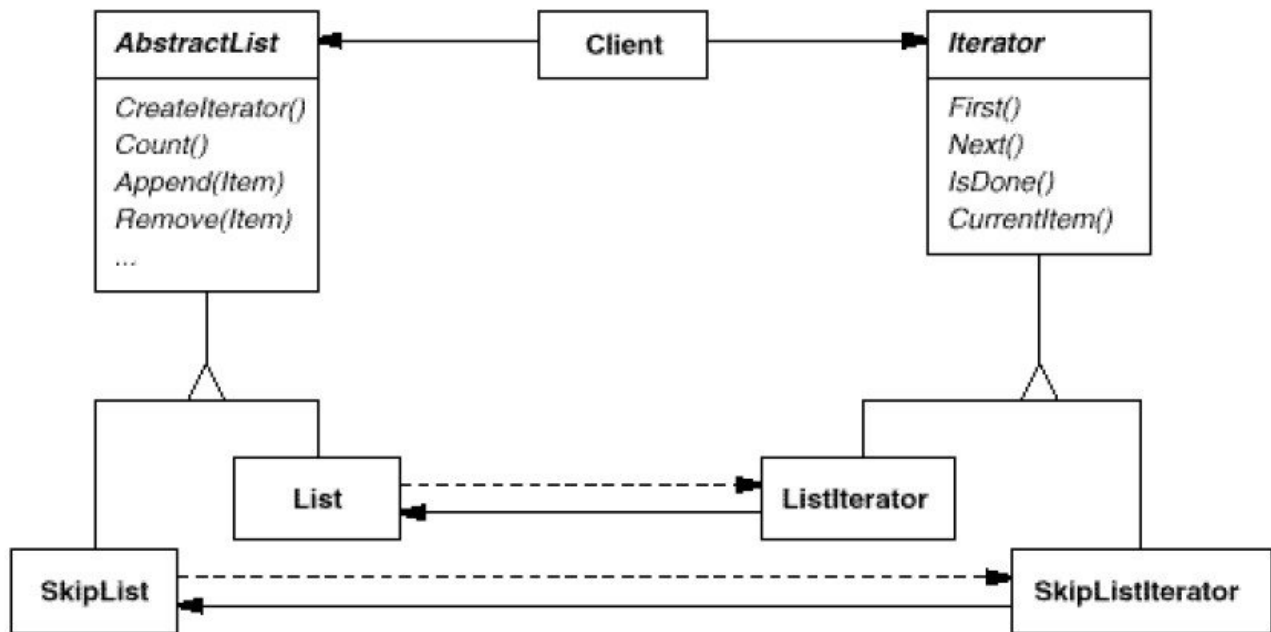
Șablonul Iterator (Cursor)

Intenție: Oferă acces secvențial la elementele unui agregat fără a expune reprezentarea internă.

Motivație:

- Un agregat (ex. listă) ar trebui să permită accesul la elemente fără a expune reprezentarea internă
- Ar trebui să permită traversarea elementelor în moduri diferite (înainte, înapoi, random)

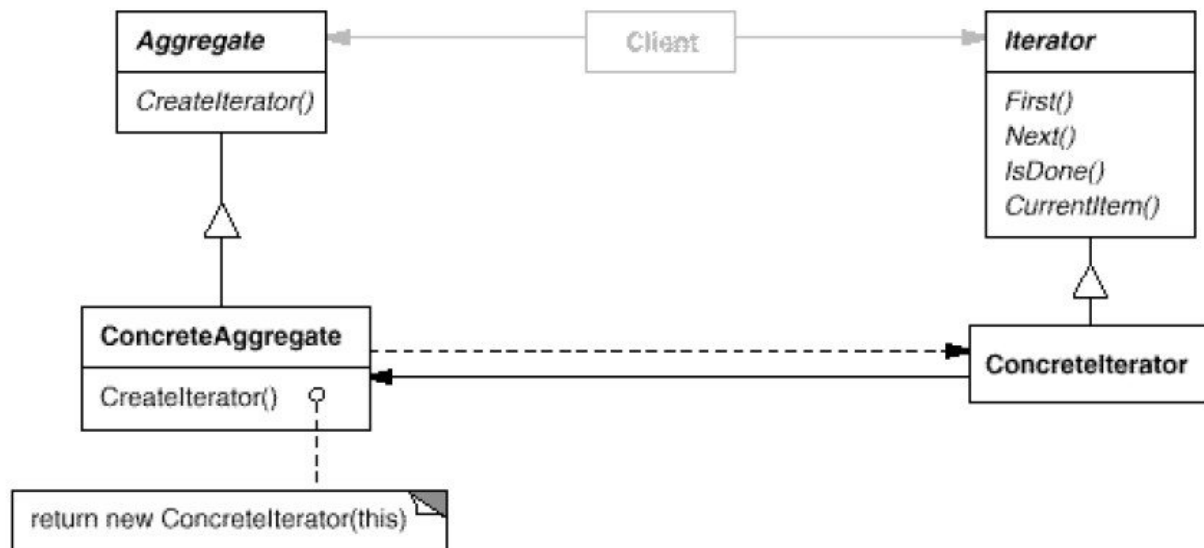
Exemplu: List, SkipList, Iterator



Aplicabilitate:

- diferite tipuri de traversari pentru un agregat
- oferă o interfață uniformă pentru accesul la elementele unui agregat

Iterator design pattern structure



Participants:

Iterator: definește interfața pentru a traversa elementele

ConcreteIterator: Implementează interfața **Iterator**, responsabil cu gestiunea poziției curente din iterație

Aggregate: definește metode pentru a crea un obiect de tip iterator

ConcreteAggregate: Implementează interfața necesară pentru crearea de **Iterator** și creează **ConcreteIterator**

Consecințe:

- suportă multiple tipuri de traversări. Agregate mai complexe au nevoie de diferite metode prin care se accesează elementele
- se simplifică interfața **Aggregate**.
- Putem avea mai multe traversări în același moment.

Iteratori in STL

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

Iteratorul este un concept fundamental in STL, este elementul central pentru algoritmi oferiți de STL.

Fiecare container STL include funcții membre begin() and end()

```
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 intre algoritmi si containere

Existe mai multe tipuri de iteratori:

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

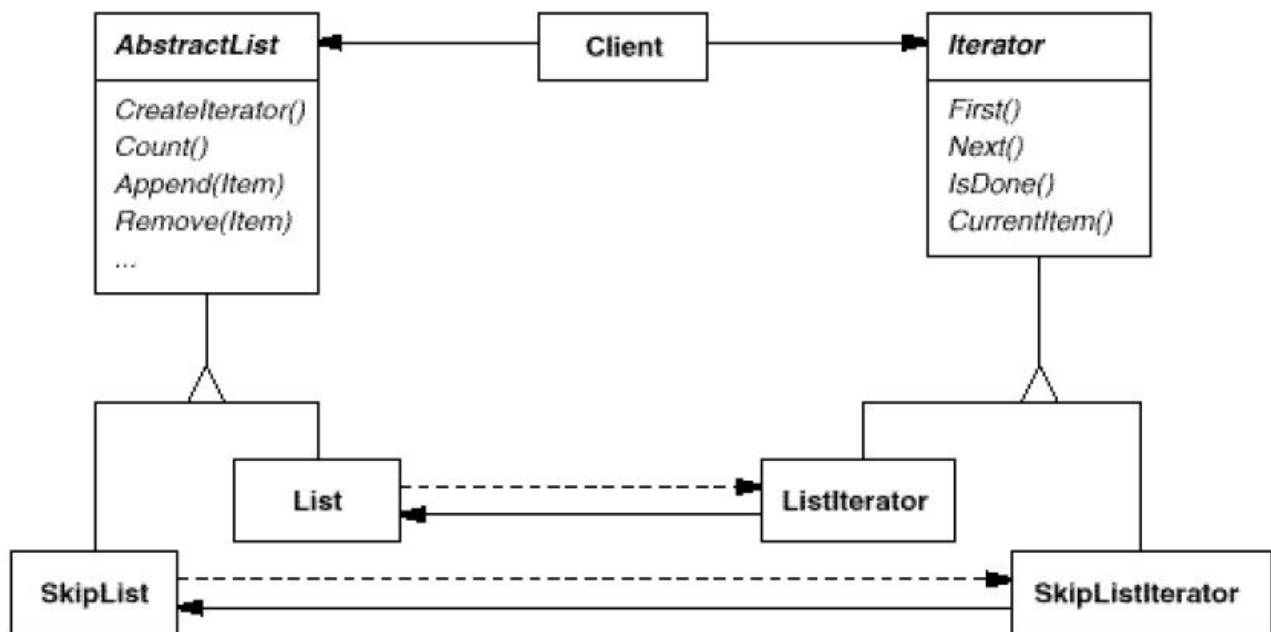
Șablonul Iterator (Cursor)

Intenție: Oferă acces secvențial la elementele unui agregat fără a expune reprezentarea internă.

Motivație:

- Un agregat (ex. listă) ar trebui să permită accesul la elemente fără a expune reprezentarea internă
- Ar trebui să permită traversarea elementelor în moduri diferite (înainte, înapoi, random)

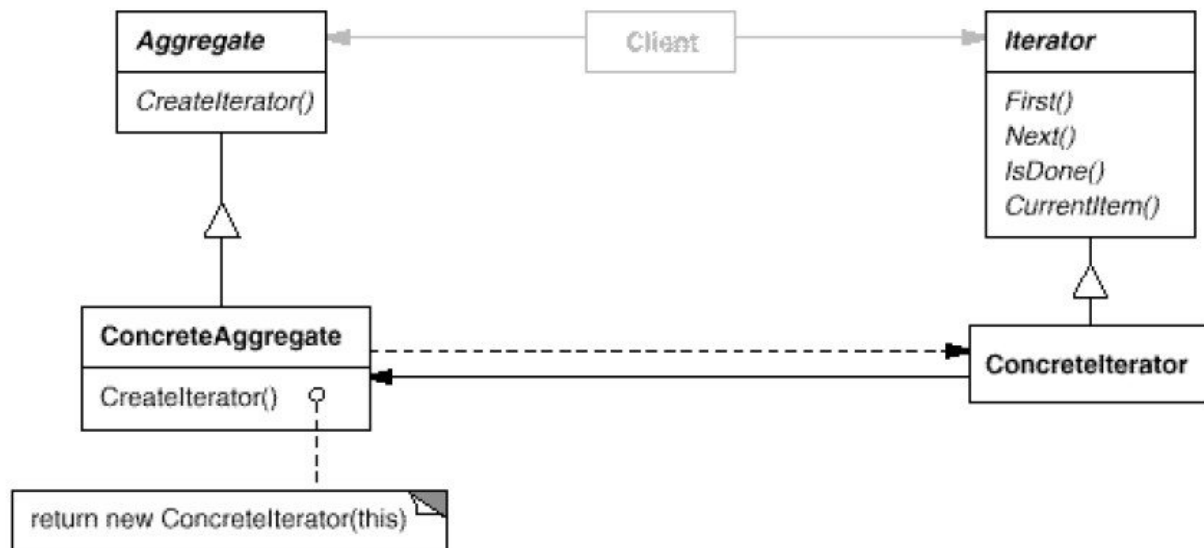
Exemplu: List, SkipList, Iterator



Aplicabilitate:

- diferite tipuri de traversări pentru un agregat
- oferă o interfață uniformă pentru accesul la elementele unui agregat

Șablonul Iterator - structură



Participanți:

Iterator: definește interfața pentru a traversa elementele

ConcreteIterator: Implementează interfața **Iterator**, responsabil cu gestiunea poziției curente din iterație

Aggregate: definește metode pentru a crea un obiect de tip **Iterator**

ConcreteAggregate: Implementează interfața necesară pentru crearea de **Iterator** și creează **ConcreteIterator**

Consecințe:

- suportă multiple tipuri de traversări. Agregate mai complexe au nevoie de diferite metode prin care se accesează elementele
- se simplifică interfața **Aggregate**.
- Putem avea mai multe traversări în același moment.

Iteratori in STL

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

Iteratorul este un concept fundamental in STL, este elementul central pentru algoritmi oferiți de STL.

Fiecare container STL include funcții membre **begin()** and **end()**

```
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

Tipuri de Iterator in STL

Iterator Type	Behavioral Description	Operations Supported
random access (most powerful)	Store and retrieve values Move forward and backward Access values randomly	* = ++ -> == != -- + - [] < > <= >= += -=
bidirectional	Store and retrieve values Move forward and backward	* = ++ -> == != --
forward	Store and retrieve values Move forward only	* = ++ -> == !=
input	Retrieve but not store values Move forward only	* = ++ -> == !=
output (least powerful)	Store but not retrieve values Move forward only	* = ++

Iteratori oferiți de containere STL

Container Class	Iterator Type	Container Category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	

- Adaptoarele de containere (container adaptors) - stack, queue, priority_queue - nu oferă iterator

Algoritmi STL

- O colecție de funcții template care pot fi folosite cu iteratori. Funcțiile operează pe un domeniu (range) definit folosind iteratori.
- Un domeniu (range) este o secvență de obiecte care pot fi accesate folosind iteratori sau pointeri

header files: <algorithm> or <numeric>.

- <numeric> conține algoritmi (funcții) ce operează în general cu numere (calculează ceva)
- <algorithm> conține algoritmi de uz general.
- Dacă aveți erori de compilare pentru că nu se găsește o funcție STL, includeți ambele header-uri.

Funcții (algoritmi) STL:

- Operații pe secvențe
 - care nu modifică sursa: **accumulate**, **count**, **find**, **count_if**, etc
 - care modifică : **copy**, **transform**, **swap**, **reverse**, **random_shuffle**, etc
- Sortări: **sort**, **stable_sort**, etc
- Pe secvențe de obiecte ordonate
 - Căutare binară : **binary_search**, etc
 - Interclasare (Merge): **merge**, **set_union**, **set_intersect**, etc
- Min/max: **min**, **max**, **min_element**, etc
- Heap: **make_heap**, **sort_heap**, etc

STL - accumulate

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);
//compute the sum of all elements in the vector
cout << accumulate(v.begin(), v.end(), 0) << endl;

//compute the sum of elements from 1 inclusive, 4 exclusive [1,4)
vector<int>::iterator start = v.begin()+1;
vector<int>::iterator end = v.begin()+4;
cout << accumulate(start, end, 0) << endl;
```

- accumulate : calculează suma elementelor

```
/**
 * @brief Accumulate values in a range.
 *
 * Accumulates the values in the range [first,last) using operator+().
The
 * initial value is @a init. The values are processed in order.
 *
 * @param first Start of range.
 * @param last End of range.
 * @param init Starting value to add other values to.
 * @return The final sum.
 */
template<typename _InputIterator, typename _Tp>
accumulate(_InputIterator __first, _InputIterator __last, _Tp __init)
```

STL – copy

Copiează elemente

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
OutputIterator result)
{
    while (first!=last) *result++ = *first++;
    return result;
}
```

Parametrii:

- first, last – definește secvența care se copiează (Input iterator) - [first,last)
- Result – Iterator de tip Output, poziționat pe primul element în destinație. (nu poate referi element din secvența sursă)

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//make sure there are enough space in the destination
//allocate space for 5 elements
vector<int> v2(5);

//copy all from v to v2
copy(v.begin(), v.end(), v2.begin());
```


STL – sort

- Sorteaza elementele din intervalul [first,last) ordine crescătoare.
- Elementele se compară folosind operatorul **operator <**

Parametrii:

- first, last : secvența de elemente ce dorim să ordonăm [first,last)
- Necesită iterator Random-Access
- Comp: funcție de comparare, relația folosită la ordonare

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//sort
sort(v.begin(), v.end());
```

Folosind o funcție de comparare:

```
bool asc(int i, int j) {
    return (i < j);
}
```

```
bool desc(int i, int j) {
    return (i > j);
}
```

```
void testSortCompare() {
    vector<int> v;
    v.push_back(3);
    v.push_back(4);
    v.push_back(2);
    v.push_back(7);
    v.push_back(17);

    //sort
    sort(v.begin(), v.end(), asc);

    //print elements
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
```

STL – for_each, transform

Aplică o funcție pe fiecare element din secvență [first,last).

```
void testForEach() {
    vector<int> v;
    v.push_back(3);
    v.push_back(4);
    v.push_back(2);
    v.push_back(7);
    v.push_back(17);

    for_each(v.begin(), v.end(), print);
    cout << endl;
}

void print(int elem) {
    cout << elem << " ";
}
```

Transform: aplică funcția pentru fiecare element din secvență ([first1,last1)) rezultatul se depune în secvența rezultat.

```
int multiply3(int el) {
    return 3 * el;
}

void testTransform() {
    vector<int> v;
    v.push_back(3);
    v.push_back(4);
    v.push_back(2);
    v.push_back(7);
    v.push_back(17);

    vector<int> v2(5);
    transform(v.begin(), v.end(), v2.begin(), multiply3);

    //print the elements
    for_each(v2.begin(), v2.end(), print);
}
```

Functor - STL Function Objects (Functors)

- Functor – Orice clasă ce are definit **operator ()**
- Față de pointer la funcție, functorul poate avea informații adiționale (variabile membre)

Dacă avem un obiect **f**, putem folosi obiectul similar ca și o funcție

Exemplu

someValue = f(arg1, arg2);

este același cu:

someValue = f.operator()(arg1, arg2);

```
class MyClass {
public:
    bool operator()(int i, int j) {
        return (i > j);
    }
};

sort(v.begin(), v.end(), MyClass());
```

Adaptor Iterator pentru inserție (insert iterator, inserters)

- Permite algoritmilor STL să opereze în modul de inserare (în loc de suprascriere)
- Rezolvă problema ce apare dacă dorim să scriem (modificăm) elemente în secvența destinație dar nu mai este suficient spațiu.

Tipuri de iteratori de inserție:

- **back_inserter**, poate fi folosit pentru containere care oferă operația **push_back()**.
- **front_inserter**, poate fi folosit pentru containere care oferă operația **push_front()**.
- **inserter**, poate fi folosit pentru containere care oferă operația **insert()**.

```
deque<int> v;  
v.push_back(4);  
v.push_back(8);  
v.push_back(12);  
  
deque<int> v2;  
// back_insert_iterator<deque<int> > dest(v2);  
front_insert_iterator<deque<int> > dest(v2);  
  
//copy elements from v to v2  
copy(v.begin(), v.end(), dest);
```

Adaptor de iterator pentru Input/Output

poate itera peste streamuri (citește sau scrie)

Un stream are funcționalitatea potrivită pentru un iterator (se pot accesa elementele secvențial) dar nu are interfața potrivită

Ex. Interfața (metodele oferite) de Cin nu este potrivit pentru a fi folosit cu algoritmi STL => STL oferă adaptor (implementarea folosește șablonul de proiectare adapter)

istream_iterator – transformă, adaptează interfața istream la interfața iterator

```
//create a istream iterator using the standard input
istream_iterator<int> start(cin);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy elements from the standard input into the vector
copy(start, end, dest);
```

Exemplu

```
const int size = 1000; // array of 1000 integers
int array[size];

cout << "Elements (Nr elements <1000):";

int elem;
int n = 0;
while (cin >> elem) {
    array[n++] = elem;
}

//sort the array
qsort(array, n, sizeof(int), cmpInt);

cout << endl;
//print the array
for (int i = 0; i < n; i++)
    cout << array[i] << " ";
cout << endl;

ifstream inFile("in.txt");
//create a istream iterator using the file
istream_iterator<int> start(inFile);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy from the standard input into the vector
copy(start, end, dest);
inFile.close();

sort(v.begin(), v.end());

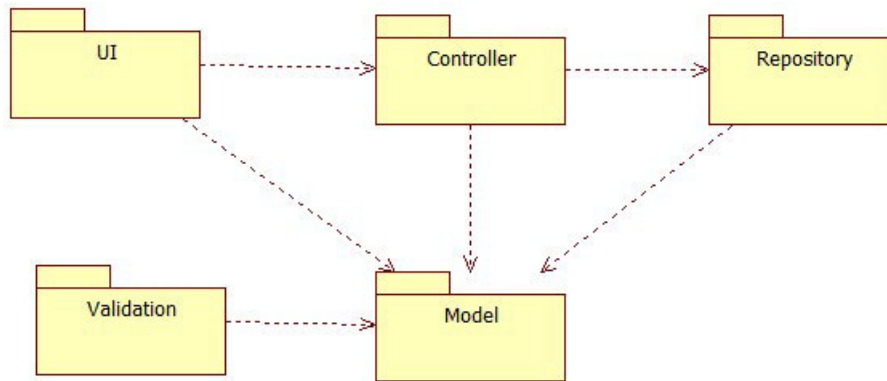
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;

//copy to file
ofstream outFile("out.txt");
copy(v.begin(), v.end(), ostream_iterator<int>(outFile, " "));
outFile.close();
```

Algoritmi STL

- simplitate: se poate folosi codul existent, nu e nevoie de implementare (sortare, copiere, etc)
- corectitudine: algoritmi din STL au fost testați, funcționează corect
- performanță: în general are performanțe mai bune decât codul scris adhoc
 - Folosește algoritmi sofisticati (e mai performant decât codul scris de un programator C)
 - Folosește tehnici mecanisme avansate (template specialization, template metaprogramming), algoritmi STL sunt optimizați.
 - Algoritmul STL poate exploata detalii interne de implementare de la containere.
- Claritate : codul este mai scurt, citind o linie se poate înțelege algoritmul
- Cod ușor de întreținut/modificat

Șablonul arhitectură stratificată



Definește arhitectura logică a sistemului

Șablonul arhitectură stratificată - Gestiunea memoriei

Obiectele se transmit între straturi:

- **valori - se fac copii la transmiterea obiectelor între nivele**
- **pointeri – cine este responsabil cu crearea/distrugerea obiectelor**

Transmitere obiecte folosind valori

```
vector<Produce> ProductFileRepository::getAll() {
    return cache;
}

vector<Produce> DepozitControler::sortByNume() {
    return sortBy(cmpName);
}

vector<Produce> DepozitControler::sortBy(bool (*cmp)(Produce p1, Produce p2)) {
    vector<Produce> all = repo->getAll();

    //crez o copie sa nu afectam cache-ul din repository
    vector<Produce> cpy(all.size());
    copy(all.begin(), all.end(), cpy.begin());

    sort(cpy.begin(), cpy.end(), cmp);
    return cpy;
}

void Console::sortByName() {
    vector<Produce> prods = ctr->sortByNume();
    showProducts(prods);
}
```

Se fac copii ale obiectelor

Există excepții: Named return value optimisation

Transmitere obiecte folosind pointeri

```
vector<Produs*> ProductFileRepository::getAll() {
    return cache;
}

vector<Produs*> DepozitControler::sortByNume() {
    return sortBy(cmpName);
}

vector<Produs*> DepozitControler::sortBy(bool (*cmp)(Produs* p1, Produs* p2)) {
    vector<Produs*> all = repo->getAll();

    //creem o copie sa nu afectam ce e in repository
    vector<Produs*> cpy(all.size());
    copy(all.begin(), all.end(), cpy.begin());

    sort(cpy.begin(), cpy.end(), cmp);
    return cpy;
}

void Console::sortByName() {
    vector<Produs*> prods = ctr->sortByNume();
    showProducts(prods);
}
```

Cine dealoca produsele din vector?

```
/**
 * The dtor only erases the elements, and note that if the
 * elements themselves are pointers, the pointed-to memory is
 * not touched in any way. Managing the pointer is the user's
 * responsibility.
 */
~vector()
{ std::_Destroy(this->_M_impl._M_start, this->_M_impl._M_finish,
               _M_get_Tp_allocator()); }
```

Eliberăm

```
ProductFileRepository::~ProductFileRepository() {
    //eliberam memoria ocupata de cache
    vector<Produs*>::iterator it = cache.begin();
    while (it != cache.end()) {
        Produs* p = *it;
        delete p;
        it++;
    }
    cache.clear();
}
```

Smart pointer

Se comportă ca și pointerii normali (permit operațiile uzuale *,++,etc) dar oferă mecanisme utile legate de gestiune memoriei.

Pot fi folosiți pentru a gestiona probleme legate de gestiunea memoriei :
memory leak, dangling pointer, null pointer etc.

Există mai multe variante: `auto_ptr`, `unique_ptr`, `shared_ptr`,
`weak_ptr`, etc.

Smart pointer: `auto_ptr`

varianta cea mai simplă de Smart pointer, se găsește în headerul `<memory>`
Încapsulează un pointer, delegă operațiile pe pointer către pointerul conținut.

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                {delete ptr;}
    T& operator*()             {return *ptr;}
    T* operator->()            {return ptr;}
    // ...
};
```

Oferă în plus (smart): eliberează memoria alocată de pointer în destructor.

<pre>void foo() { MyClass* p = new MyClass(); p->DoSomething(); delete p; }</pre>	<pre>#include <memory> void foo2() { auto_ptr<MyClass> p(new MyClass); p->DoSomething(); }</pre>
--	---

Smart pointer: `auto_ptr`

Exception safety

```
void foo() {
    MyClass* p = new MyClass();
    p->DoSomething();
    delete p;
}

void DoSomething() {
    throw 3;
}

#include <memory>
void foo2() {
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

`auto_ptr` preia responsabilitatea de a șterge obiectul

la copiere se transferă responsabilitatea la obiectul destinație și sursa pierde referința (strict ownership)

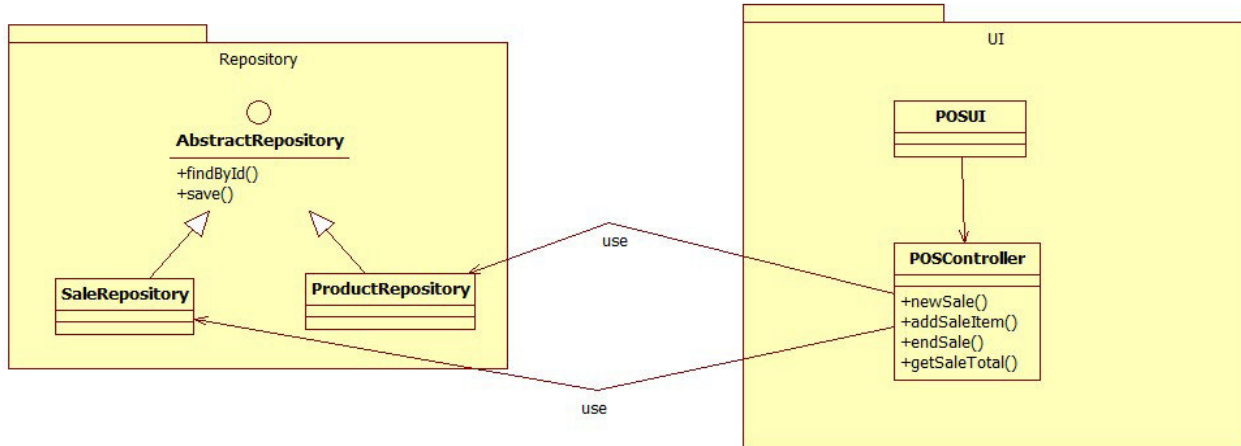
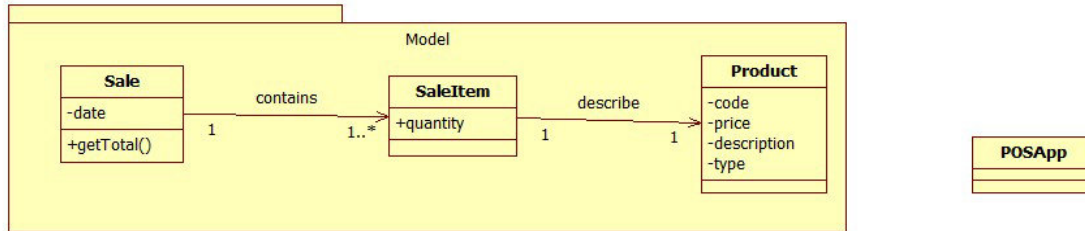
```
void assignment() {
    auto_ptr<MyClass> x(new MyClass());
    auto_ptr<MyClass> y;

    y = x; //se transfera la x

    cout << x.get() << endl; // Print NULL
    cout << y.get() << endl; // Print non-NULL address
}
```

Există și alte strategii: Ex. `shared_ptr` (disponibil standard in c++ 11) implementează reference counting

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);
}

```

POS – application

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);
}
```

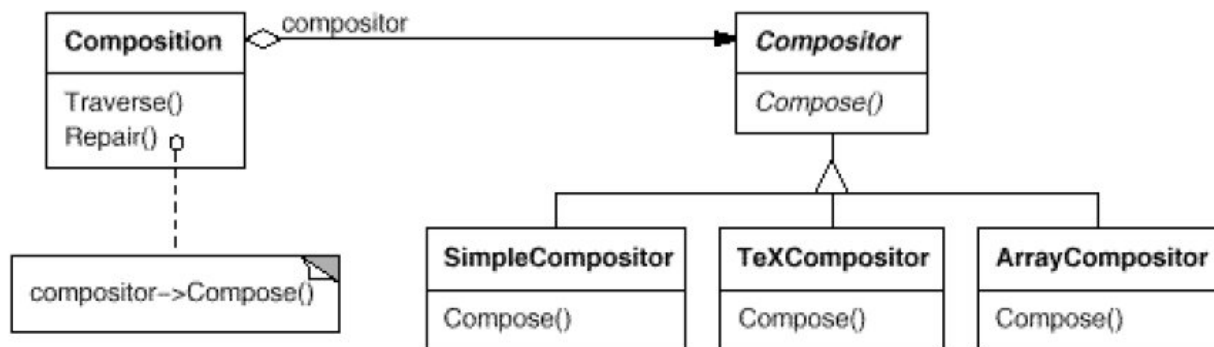
Această abordare conduce la cod complicat, calcule care sunt greu de urmărit. Cod greu de întreținut, extins, înțeles.

Șablonul de proiectare Strategy (policy)

Scop: Definește modul de implementare a unor familii interschimbabile de algoritmi.

Motivare:

Aplicația de editor de documnte, are o clasă **Composition** responsabil cu menținerea și actualizarea aranjării textului (line-breaks). Există diferiți algoritmi pentru formatarea textului pe linii. În funcție de context se folosesc diferiți algoritmi de formatare.



Fiecare strategie de formatare este implementat separat în clase derivate din clasa abstractă **Compositor** (nu **Composition**).

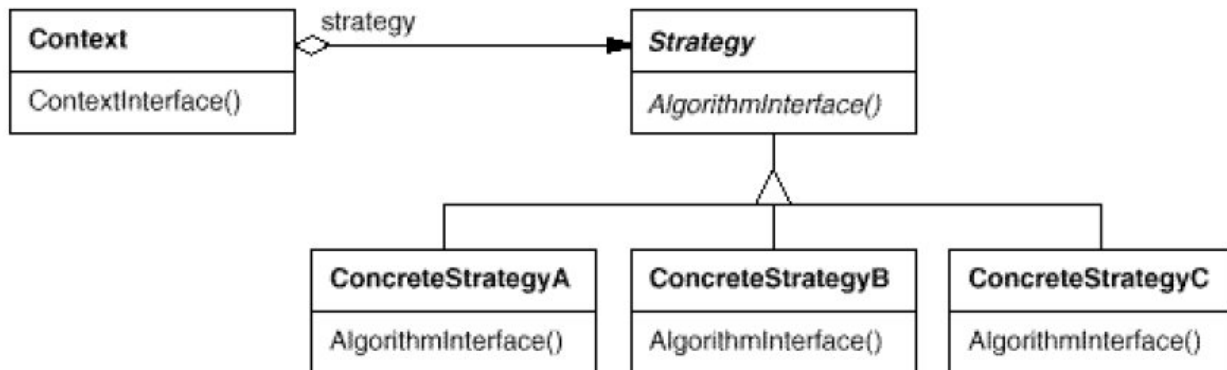
Clasele derivate din **Compositor** implementează strategii:

- **SimpleCompositor** implementează strategie simplă, adaugă linie nouă una câte una.
- **TeXCompositor** implementează algoritmul TeX pentru a identifica poziția unde se adaugă linie nouă (identifică liniile globale, analizând tot paragraful).
- **ArrayCompositor** formatează astfel încât pe fiecare linie există același număr de elemente (cuvinte, icoane, etc).

Strategy (Policy)

Aplicabilitate:

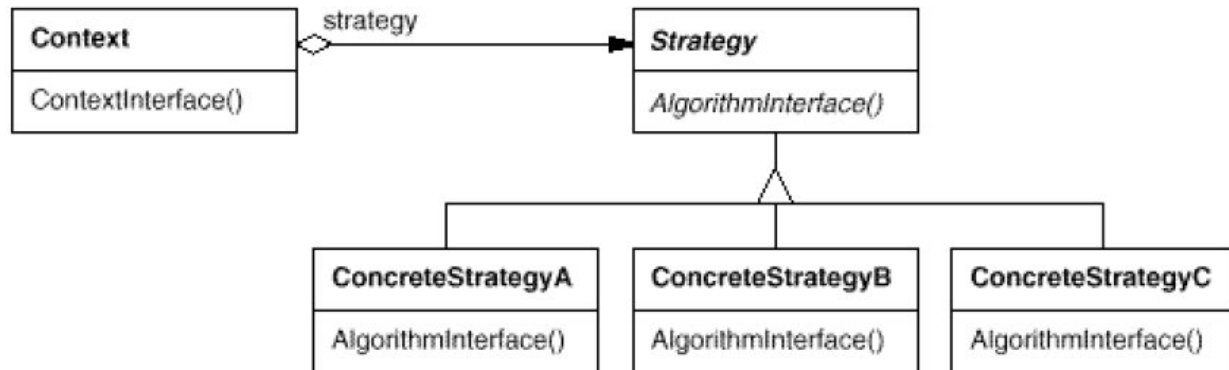
- mai multe clase sunt similare, există diferențe ca și comportament. Șablonul Strategy oferă o metodă de a configura comportamentul.
- Este nevoie de mai multe variante de algoritmi pentru o problemă.
- Un algoritm folosește date despre care clientul nu ar trebui să știe. Se poate folosi șablonul Strategy 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 șablonul Strategy .



Participanți:

- **Strategy** (Compositor): definește interfața comună pentru toți algoritmi. Context folosește această interfața pentru a apela efectiv algoritmul definit de clasa ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor) 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 algoritmi. În general Client crează un obiect ConcreteStrategy și transmite la Context;
- Clientul interacționează doar cu context. În general există multiple versiuni de ConcreteStrategy din care clientul poate alege.

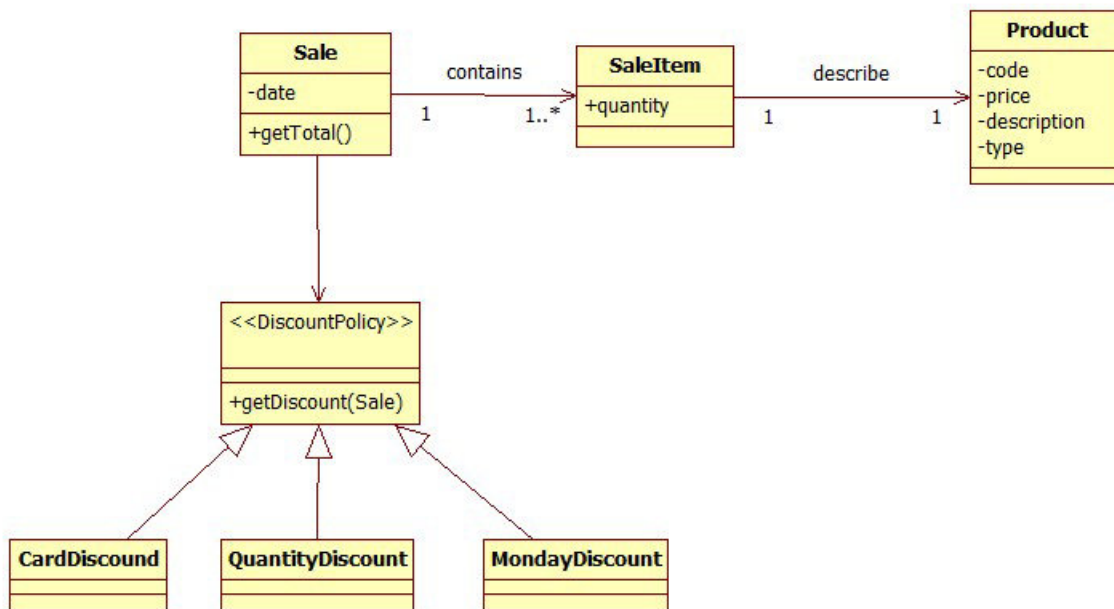
Consecințe:

- Familie de algoritmi se pot defini ca și o hierarhie de clase. Moștenirea poate ajuta să extragem părți comune.
- Se elimină if-else și switch. Șablonul 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 and 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 șablonului 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;
};
/**
 * Apply 2% discount
 */
class CreditCardDiscount: public DiscountPolicy {
public:
    virtual double getDiscount(const Sale* s, SaleItem si) {
        return si.getQuantity() * si.getProduct().getPrice() * 0.02;
    }
};

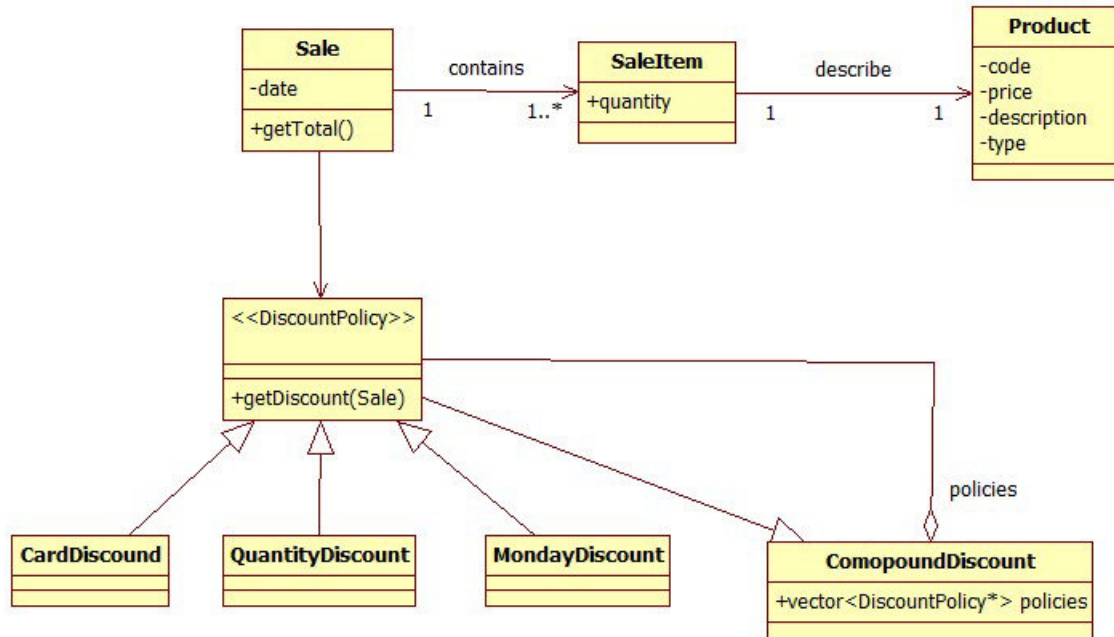
/**
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
    double total = 0;
    for (int i = 0; i < items.size(); i++) {
        SaleItem sIt = items[i];
        double price = sIt.getQuantity() * sIt.getProduct().getPrice();
        //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?

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);

    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)