Objektorientētā programmēšana

Programmēšanas paradigmas

Igors Ščukins
Daugavpils RTU

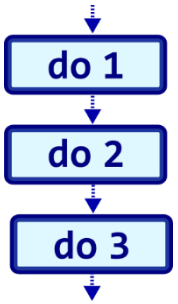
Programmēšanas paradigmas

Programmēšanas paradigma ir programmēšanas stils jeb rekomendāciju kopums, kas nosaka programmu rakstīšanas veidu. Piemēram, objektorientētajā programmēšanā programmētājam jādomā par programmu, kas sastāv no savstarpēji saistītiem objektiem, kamēr funkcionālajā programmēšanā programma sastāv no secīgi izsaucamām funkcijām.

Programmēšanā ir divas pamatparadigmas: **imperatīvā** un **deklaratīva**.

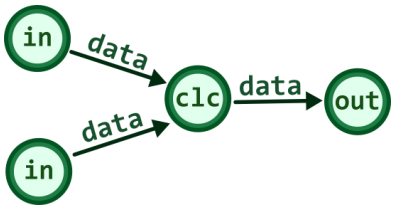
Imperatīvā paradigma

Imperatīvā programma **ir komandu secība** (*imperative – pavēles izteiksme*), kas secīgi jāizpilda, lai risinātu uzdevumu. Imperatīvā pieeja plaši izmanto mainīgos, kas glabā programmas tekošo stāvokli, un piešķiršanas komandas, kas secīgi maina šo stāvokli.



Deklaratīvā paradigma

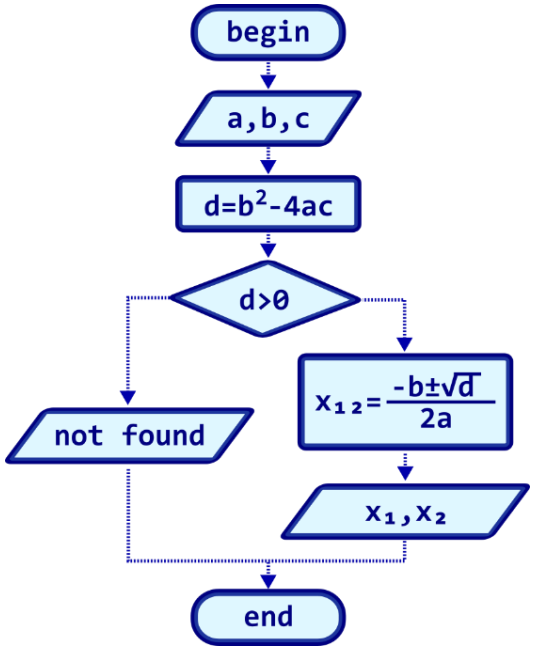
Deklaratīvā pieejā nosaka **uzdevuma aprakstu**. Piemēram, kā un kur paņem datus, kā tā apstrādāt un ko darīt ar rezultātu. Deklaratīvās programmas neizmanto mainīgā jēdzienu un piešķiršanas operatoru.



Imperatīvā programmēšana

Datoram lai atrisinātu uzdevumu ir jāizpilda noteikto darbību secīgums jeb algoritms. Programma, kā to definē fon Neimans un Tjūrings, ir paredzēta, lai aprakstītu šādus algoritmus. No šī viedokļa programma sastāv no komandām, kas datoram ir jāizpilda secīgi. Lai realizētu nosacītas operācijas un ciklus, programmā var būt pārejas instrukcijas (**goto**), kas maina izpildīšanas secību.

Imperatīvās programmas aprakstām var pielietot blokshēmas:



Tīram imperatīvam programmām ir raksturīga operatora **goto** pielietošana:

```
int main()
{
    double x1, x2;
    double a = 1, b = -5, c = 6;
    double d = b * b - 4 * a * c;
    if (d < 0) goto NOTFOUND;
    x1 = (-b + sqrt(d)) / (2 * a);
    x2 = (-b - sqrt(d)) / (2 * a);
    printf("x1=%.2f\nx2=%.2f\n", x1, x2);
    goto EXIT;
NOTFOUND:
    printf("Roots not found");
EXIT:
}
```

Strukturālā programmēšana

Strukturālā programmēšana ir imperatīvās paradigmas paveids, kuras pamatā ir programmas attēlojums kā ligzdotu bloku struktūra, neizmantojot **goto** operatoru (Edgar Dijkstra "*GOTO Considered Harmful*"). Saskaņā ar šo paradigmu jebkura programma sastāv no trim vadības struktūrām: **secība**, **zarošana**, **cikls**.

```
int main()
{
    double a = 1, b = -5, c = 6;
    double d = b * b - 4 * a * c;
    if (d < 0) {
        printf("Roots not found");
    }
    else {
        double x1 = (-b + sqrt(d)) / (2 * a);
        double x2 = (-b - sqrt(d)) / (2 * a);
        printf("x1=%.2f\nx2=%.2f\n", x1, x2);
    }
}
```

Procedurālā programmēšana

Procedurālā programmēšana ir imperatīvās paradigmas paveids, kurā secīgi izpildāmie operatori tiek savākti apakšprogrammās, kas ir **funkcionāli pabeigtās** koda vienības.

```
double getD(double a, double b, double c) {
    return b * b - 4 * a * c;
}

bool solve(double a, double b, double c, double& x1, double& x2) {
    double d = getD(a, b, c);
    if (d < 0) return false;
    x1 = (-b + sqrt(d)) / (2 * a);
    x2 = (-b - sqrt(d)) / (2 * a);
    return true;
}

int main() {
    double x1, x2;
    if (solve(1, -5, 6, x1, x2)) printf("x1=%.2f\nx2=%.2f\n", x1, x2);
    else printf("Roots not found");
}
```

Funkcionālā programmēšana

Funkcionālā programmēšana ir deklaratīvās paradigmas paveids, kurā uzdevums tiek aprakstīts kā funkciju izsaukšanas rezultāts no sākotnējiem datiem un citu funkciju rezultātiem, un tas ne uzņemas skaidru programmas stāvokļa saglabāšanu mainīgajos.

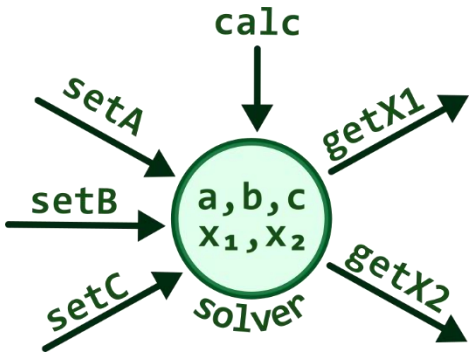
```
double A() { return 1;  }
double B() { return -5; }
double C() { return 6;  }

double  D(double a, double b, double c) { return b * b - 4 * a * c; }
double X1(double a, double b, double c) { return (-b + sqrt( D(a, b, c) )) / (2 * a); }
double X2(double a, double b, double c) { return (-b - sqrt( D(a, b, c) )) / (2 * a); }

int main()
{
    D( A(), B(), C() ) > 0 ?
        printf( "x1=%.2f\nx2=%.2f\n", X1(A(), B(), C()), X2(A(), B(), C()) )
        :
        printf("Roots not found");
}
```

Objektorientētā programmēšana

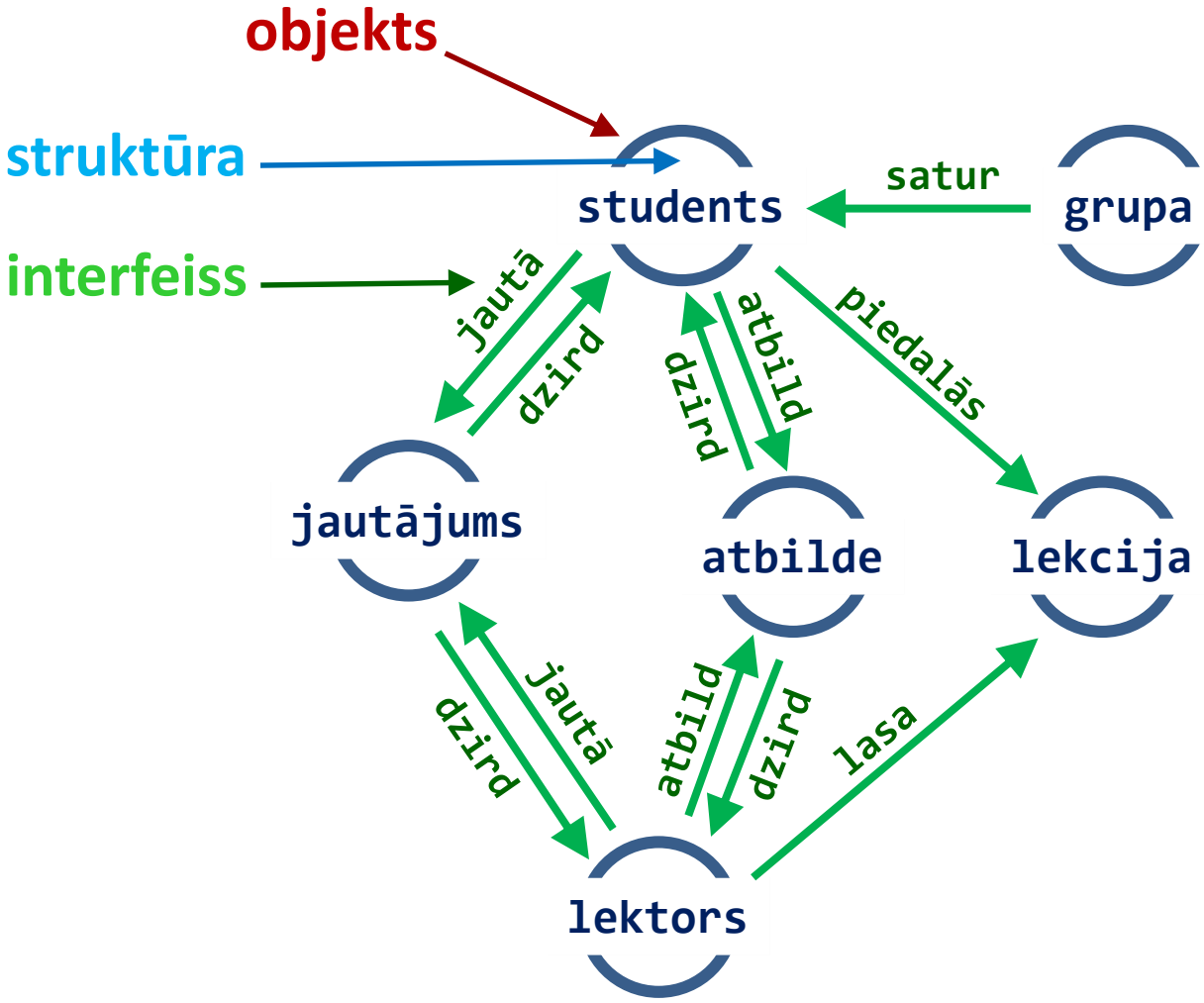
Objektorientētajā programmēšanā programma tiek uztverta kā **savstarpēji saistītu un mijiedarbojošu objektu kopums**. Tāpat kā reāliem objektiem dabā, OOP objektiem arī ir savas individuālas īpašības, tie var veikt kādas darbības, kā arī nodot un saņemt informāciju no citiem objektiem.



Objektorientētai programmēšanai ir īpašības, kas ir raksturīgas gan imperatīvajai gan deklaratīvajai paradigmām.

OOP programmas piemērs

```
class Solver {
private:
    double _x1 = 0, _x2 = 0;
    double _a = 1, _b = -5, _c = 6;
public:
    bool compute() {
        double d = _b * _b - 4 * _a * _c;
        if (d < 0) return false;
        _x1 = (-_b + sqrt(d)) / (2 * _a);
        _x2 = (-_b - sqrt(d)) / (2 * _a);
        return true;
    }
    double getX1() { return _x1; }
    double getX2() { return _x2; }
};

int main() {
    Solver solver;
    if (solver.compute()) printf("x1=%.2f\nx2=%.2f\n", solver.getX1(), solver.getX2());
    else printf("Roots not found");
}
```

OOP struktūras piemērs