

Disseny Descendent

(nocions bàsiques)

...

PRO1. Curs 2026-2027

Top-Down Design (a.k.a. Stepwise Refinement): Definition

Top-down design is a programming and problem-solving strategy where you **start from the big picture** of the problem and **break it down** into smaller, more manageable subproblems, repeating the process **until each subproblem is simple enough** to implement directly in code

(Think of it like zooming in from a map of a country to a street-level view of a neighborhood. You start general, then refine into details)

Top-Down Design: Key Steps

1. Start with the main task

- Identify the overall goal of your program (e.g., “calculate statistics for student grades”)

2. Divide into subtasks

- Break the main task into major steps (e.g., “read grades”, “compute average”, “display results”)

3. Refine each subtask

- Continue breaking down each step until it becomes simple enough to write as a function or a small block of code

Top-Down Design: Key Steps

4. Implement modular code. Each subtask becomes a function or module. This makes the program:
 - *Readable, testable, maintainable and reusable*
5. Combine the pieces
 - Finally, assemble the modules in the main program according to the high-level plan

Top-Down Design: Advantages

- **Clarity:** The main function describes what the program does at a high level
- **Modularity:** Functions handle specific subtasks
- **Maintainability:** Easy to modify or expand the program
- **Debugging:** Easier to test individual modules

Top-Down Design: Example Analogy

- **Problem:** Build a bicycle
- **Level 1: Identify the major components**
 - Build the frame
 - Assemble the wheels
 - Assemble the drivetrain
 - Assemble the braking system
 - Assemble the steering system
 - Put everything together
- **Level 2: Decompose complex components**
 - Assemble drivetrain
 - Install pedals
 - Install crankset
 - Install chain
 - Install gears
 - etc.

Top-Down Design: Example Analogy

```
Build bicycle
|
├── Build frame
|
├── Assemble wheels
|   ├── Assemble front wheel
|   |   ├── Assemble hub
|   |   ├── Attach spokes
|   |   └── Install tyre
|   └── Assemble rear wheel
|
├── Assemble drivetrain
|   ├── Install pedals
|   ├── Install crankset
|   ├── Install chain
|   └── Install gears
|
├── Assemble brakes
└── Assemble steering
```

- Build_bicycle() is the main program
- Each subprocess is a function
- At each level, think about what components you need, not yet about how each component is built
 - we don't initially care how assemble_wheels() works. We assume that operation exists and concentrate on the higher-level problem
- Move down one level and repeat until every component is simple enough to build directly

Top-Down Design: Coding Example

Problem

Write a program to sum a sequence of pairs of fractions

Input: each line contains two fractions a/b and c/d in the form of four int numbers separated by spaces "a b c d"

Output

Summation of the pairs of fractions $a/b + b/c$ (one per line) and the final sum of all of them at the end. The fractions need to be simplified. If a fraction is $0/n$ you should write only 0. If the fraction is $n/1$ you should write only n.

Top-Down Design: Coding Example

Example

Input

1 3 2 3
-1 5 2 10
34 45 12 12

Output

$1/3 + 2/3 = 1$
 $-1/5 + 2/10 = 0$
 $34/45 + 12/12 = 79/45$
Total = $124/45$

Let's work on the code using top-down design!

Coding Example: Summing Pairs of Fractions

Preamble: we want to represent rational numbers with a struct

```
struct Rational {  
    int num; // numerator  
    int den; // denominator  
};
```

Coding Example: Summing Pairs of Fractions (2)

A very simple main program (comments omitted)

```
int main() {  
    Rational rat1, rat2, total={0,1};  
    while (read_pair_rationals(rat1, rat2)) {  
        Rational sum = sum_and_simp(rat1, rat2);  
        write_sum(rat1, rat2, sum);  
        total = sum_and_simp(total, sum);  
    }  
    write_result(total);  
}
```

Coding Example: Summing Pairs of Fractions (3)

First-level functions and actions

`read_pair_rationals(rat1, rat2)`

`sum_and_simp(rat1, rat2)`

`write_sum(rat1, rat2, sum)`

`write_result(total)`



This is direct

This needs more decomposition

These two need an extra level but they are very simple

Coding Example: Summing Pairs of Fractions (4)

First-level read_pair_rationals() function:

```
// PRE: standard input contains a possibly empty sequence of pairs of rational numbers
// POST: returns true if both rational numbers have been read correctly, false otherwise
bool read_pair_rationals(Rational &r1, Rational &r2) {
    return(cin >> r1.num >> r1.den >> r2.num >> r2.den);
}
```

Coding Example: Summing Pairs of Fractions (5)

First-level `sum_and_simp()` function:

```
// PRE: x and y are rational numbers
// POST: returns the simplified sum of x and y
Rational sum_and_simp(Rational x, Rational y) {
    Rational sum;
    sum.num = x.num * y.den + y.num * x.den;
    sum.den = x.den * y.den;
    return simplify(sum);
}
```

Coding Example: Summing Pairs of Fractions (6)

Second-level simplify() function

```
// PRE: r is a rational number
// POST: returns r in simplified form
Rational simplify(Rational r) {
    // the denominator must be positive to compute the GCD
    if (r.den < 0) {
        r.den = -r.den;
        r.num = -r.num;
    }
    int g = gcd_euclides(abs(r.num), r.den);
    r.num = r.num/g;
    r.den = r.den/g;
    return r;
}
```

Coding Example: Summing Pairs of Fractions (7)

Third-level gcd_euclides() function

```
// PRE: a and b are non-negative integers, b > 0
// POST: returns the greatest common divisor of a and b
int gcd_euclides(int a, int b) {
    while (b != 0) {
        int residual = a % b;
        a = b;
        b = residual;
    }
    return a;
}
```

Coding Example: Summing Pairs of Fractions (8)

First-level write_sum() procedure

```
// PRE: r1, r2, and sum are rational numbers
// POST: the expression  $r1 + r2 = \text{sum}$  has been written to standard output
void write_sum(Rational r1, Rational r2, Rational sum) {
    write_rational(r1);
    cout << " + ";
    write_rational(r2);
    cout << " = ";
    write_rational(sum);
    cout << endl;
}
```

Coding Example: Summing Pairs of Fractions (9)

Second-level write_rational() procedure

```
// PRE: r is a rational number
// POST: r has been written in the format specified in the problem statement
void write_rational(Rational r) {
    if (r.num == 0) cout << "0";
    else if (r.den == 1) cout << r.num;
    else cout << r.num << "/" << r.den;
}
```

Coding Example: Summing Pairs of Fractions (10)

First-level write_result() procedure

```
// PRE: total is a rational number
// POST: total has been written in the format specified in the problem statement
void write_result(Rational total) {
    cout << "Total = ";
    write_rational(total);
    cout << endl;
}
```

Coding Example: Summing Pairs of Fractions

Wrapping up

- We solved an apparently complex problem by top-down decomposition into 7 simple functions:
 - 4 at the first level, 2 at the second level, 1 at the third level
- The main program and each function contain only 6–7 lines of code.
- If the decomposition is right, putting all the pieces together is straightforward.
- The complete program has around 100 lines (including comments).
- We definitely do not want those 100 lines inside `main()` !

Coding Example: Summing Pairs of Fractions

Wrapping up

