Abstract Data Types
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

Hiding details: abstractions
Different types of abstractions

Concept maps are hierarchical: why?

The computer systems stack

How data flows through system

Boolean logic gates and functions

Combining devices to do useful work

Transistors and wires

Silicon process technology

Application
Algorithm
Programming Language
Operating System
Instruction Set Architecture
Microarchitecture
Register-Transfer Level
Gate Level
Circuits
Devices
Technology

Each level has few items

Image Credit: Christopher Batten, Cornell University

Image Credit: Christopher Batten, Cornell University
Our challenge

- We need to design large systems.
- We need to reason about complex algorithms.
- Our working memory can only manipulate 4 things at once.
- We need to interact with computers using programming languages.
- Solution: abstraction
  - Abstract reasoning.
  - Programming languages that support abstraction.
- We already use a certain level of abstraction: functions. But it is not sufficient. We need much more.

Data types

- Programming languages have a set of primitive data types (e.g., int, bool, double, char, ...).
- Each data type has a set of associated operations:
  - We can add two integers.
  - We can concatenate two strings.
  - We can divide two doubles.
  - But we cannot divide two strings!
- Programmers can add new operations to the primitive data types:
  - gcd(a,b), match(string1, string2), ...
- The programming languages provide primitives to group data items and create structured collections of data:
  - C++: array, struct.
  - python: list, tuple, dictionary.
Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them

Operations:
- Number of vertices
- Number of edges
- Shortest path
- Connected components

Data type: Graph

Data type: Polynomial

Operations:
- $P + Q$
- $P \times Q$
- $P/Q$
- $\text{gcd}(P, Q)$
- $P(x)$
- degree($P$)

$P(x) = x^3 - 4x^2 + 5$

Abstract Data Types (ADTs)

- Separate the notions of specification and implementation:
  - Specification: “what does an operation do?”
  - Implementation: “how is it done?”

- Benefits:
  - Simplicity: code is easier to understand
  - Encapsulation: details are hidden
  - Modularity: an ADT can be changed without modifying the programs that use it
  - Reuse: it can be used by other programs

Abstract Data Types (ADTs)

- An ADT has two parts:
  - Public or external: abstract view of the data and operations (methods) that the user can use.
  - Private or internal: the actual implementation of the data structures and operations.

- Operations:
  - Creation/Destruction
  - Access
  - Modification
Abstract Data Types (ADTs)

Example: a Point

A point can be represented by two coordinates \((x,y)\).

Several operations can be envisioned:
- Get the \(x\) and \(y\) coordinates.
- Calculate distance between two points.
- Calculate polar coordinates.
- Move the point by \((\Delta x, \Delta y)\).

### Example: a Point

```java
// Things that we can do with points
Point p1(5.0, -3.2); // Create a point (a variable)
Point p2(2.8, 0); // Create another point

// We now calculate the distance between p1 and p2
double dist12 = p1.distance(p2);

// Distance to the origin
double r = p1.radius();

// Create another point by adding coordinates
Point p3 = p1 + p2;

// We get the coordinates of the new point
double x = p3.getX(); // x = 7.8
double y = p3.getY(); // y = -3.2
```

ADTs and Object-Oriented Programming

- OOP is a programming paradigm: a program is a set of objects that interact with each other.
- An object has:
  - fields (or attributes) that contain data
  - functions (or methods) that contain code
- Objects (variables) are instances of classes (types).
  A class is a template for all objects of a certain type.
- In OOP, a class is the natural way of implementing an ADT.
Let us design the new type for Point

// The declaration of the class Point
class Point {
public:
    // Constructor
    Point(double x_coord, double y_coord);
    // Gets the x coordinate
    double getX() const;
    // Gets the y coordinate
    double getY() const;
    // Returns the distance to point p
    double distance(const Point& p) const;
    // Returns the radius (distance to the origin)
    double radius() const;
    // Returns the angle of the polar coordinate
    double angle() const;
    // Creates a new point by adding the coordinates of two points
    Point operator + (const Point& p) const;
private:
    double x, y;  // Coordinates of the point
};

Implementation of the class Point

// The constructor: different implementations
Point::Point(double x_coord, double y_coord) {
    x = x_coord; y = y_coord;
}

// or also
Point::Point(double x_coord, double y_coord) :
    x(x_coord), y(y_coord) {}

// or also
Point::Point(double x, double y) : x(x), y(y) {}

All of them are equivalent, but only one of them should be chosen. We can have different constructors with different signatures.

Note: compilers are smart. Small functions are expanded inline.
Implementation of the class Point

double Point::angle() const {
    if (getX() == 0 and getY() == 0) return 0;
    return atan(getY()/getX());
}

Point Point::operator + (const Point& p) const {
    return Point(getX() + p.getX(), getY() + p.getY());
}

Public or private?

• What should be public?
  – Only the methods that need to interact with the external world. Hide as much as possible. Make a method public only if necessary.

• What should be private?
  – All the attributes.
  – The internal methods of the class.

• Can we have public attributes?
  – Theoretically yes (C++ and python allow it).

Class Point: a new implementation

• Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).

• We can modify the private section of the class without modifying the specification of the public methods.

• The private and public methods may need to be rewritten, but not the programs using the public interface.

// The declaration of the class Point
class Point {
public:
    // Constructor
    Point(double x, double y);
    // Gets the x coordinate
    double getX() const;
    // Gets the y coordinate
    double getY() const;
    // Returns the distance to point p
    double distance(const Point& p) const;
    // Returns the radius (distance to the origin)
    double radius() const;
    // Returns the angle of the polar coordinate
    double angle() const;
    // Creates a new point by adding the coordinates of two points
    Point operator + (const Point& p) const;
private:
    double _radius, _angle;  // Polar coordinates
};
Class Point: a new implementation

A new class: Rectangle

- We will only consider orthogonal rectangles (axis-aligned).
- An orthogonal rectangle can be represented in different ways:

  - Two points (extremes of diagonal)
  - One point, width and height

Rectangle: abstract view

- Create
- Scale
- Rotate
- Move
- Flip (horizontally/vertically)
- Intersection
- Point inside?
Rectangle: ADT

class Rectangle {
public:
  // Constructor (LL at the origin)
  Rectangle(double width, double height);
  // Returns the area of the rectangle
  double area() const;
  // Scales the rectangle with a factor s > 0
  void scale(double s);
  // Returns the intersection with another rectangle
  Rectangle operator *(const Rectangle& R) const;

  ...
};
Rectangle: other public methods

```cpp
Point Rectangle::getLL() const {
    return ll;
}
Point Rectangle::getUR() const {
    return ll + Point(w, h);
}
double Rectangle::getWidth() const {
    return w;
}
double Rectangle::getHeight() const {
    return h;
}
```

Rectangle: overloaded operators

```cpp
double Rectangle::area() const {
    return w*h;
}
// Notice: not a const method
void Rectangle::scale(double s) {
    w *= s;
    h *= s;
}
bool Rectangle::empty() const {
    return w <= 0 or h <= 0;
}
Rectangle* Rectangle::operator *= (const Rectangle& R) {
    // Calculate the ll and ur coordinates
    Point Rll = R.getLL();
    ll = Point(max(ll.getX(), Rll.getX()),
                max(ll.getY(), Rll.getY()));
    Point ur = getUR();
    Point Rur = R.getUR();
    double urx = min(ur.getX(), Rur.getX());
    double ury = min(ur.getY(), Rur.getY());
    // Calculate width and height (might be negative \(\rightarrow\) empty)
    w = urx - ll.getX();
    h = ury - ll.getY();
    return *this;
}
Rectangle Rectangle::operator * (const Rectangle& R) const {
    Rectangle result = *this;  // Make a copy of itself
    result *= R;
    return result;
}
```

What is *this?

- **this** is a pointer (memory reference) to the object (pointers will be explained later)

- ***this** is the object itself

Exercises: implement

```cpp
// Rotate the rectangle 90 degrees clockwise or counterclockwise, depending on the value of the parameter
void rotate(bool clockwise);
// Flip horizontally or vertically, depending on the value of the parameter.
void flip(bool horizontally);
// Check whether point p is inside the rectangle
bool isPointInside(const Point& p) const;
```

Re-implement the class Rectangle using an internal representation with two Points: lower-left (LL) and upper-right (UR) corners.
Let us work with rectangles

Rectangle R1(Point(2,3), Point(6,8));
double areaR1 = R1.area(); // areaR1 = 20

Rectangle R2(Point(3,5), 2, 4); // LL=(3,5) UR=(5,9)

// Check whether the point (4,7) is inside the
// intersection of R1 and R2.
bool in = (R1*R2).isPointInside(Point(4,7));
// The object R1*R2 is “destroyed” after the assignment.

R2.rotate(false); // R2 is rotated counterclockwise
R2 *= R1; // Intersection with R1

Exercise: draw a picture of R1 and R2 after the execution of the previous code.

Yet another class: Rational

Rational R1(3);   // R1 = 3
Rational R2(5, 4); // R2 = 5/4
Rational R3(8, -10); // R3 = -4/5
R3 += R1 + Rational(-1, 5); // R3 = 2
if (R3 >= Rational(2)) {
    R3 = -R1*R2; // R3 = -15/4
}
cout << R3.to_str() << endl;

What we would like to do:

Rational R1(3);     // R1 = 3
Rational R2(5, 4);  // R2 = 5/4
Rational R3(8, -10); // R3 = -4/5
R3 += R1 + Rational(-1, 5); // R3 = 2
if (R3 >= Rational(2)) {
    R3 = -R1*R2;  // R3 = -15/4
}
cout << R3.to_str() << endl;

The class Rational

```cpp
class Rational {
private:
    int num, den; // Invariant: den > 0 and gcd(num,den)=1
    // Simplifies the fraction (used after each operation)
    void simplify();

public:
    // Constructor (if some parameter is missing, the default value is taken)
    Rational(int num = 0, int den = 1);
    // Returns the numerator of the fraction
    intgetNum() const {
        return num;
    }
    // Returns the denominator of the fraction
    int getDen() const {
        return den;
    }
    // Returns true if the number is integer and false otherwise.
    bool isInteger() const {
        return den == 1;
    }
    // Equality comparison
    bool operator == (const Rational& r) const {
        return Rational(-getNum(), getDen()) == r;
    }
    // Returns a string representing the rational
    string to_str() const;
};
```

The class Rational

```cpp
class Rational {
public:

    ... // Arithmetic operators
    Rational& operator += (const Rational& r);
    Rational operator + (const Rational& r) const;
    // Similarly for -, *, and /

    // Unary operator
    Rational operator - () const {
        return Rational(-getNum(), getDen());
    }

    // Equality comparison
    bool operator == (const Rational& r);

    // Returns a string representing the rational
    string to_str() const;
};
```
Rational::Rational(int num, int den) : num(num), den(den) {
    simplify();
}

void Rational::simplify() {
    assert(den != 0); // We will discuss assertions later
    if (den < 0) {
        num = -num;
        den = -den;
    }

    // Divide by the gcd of num and den
    int d = gcd(abs(num), den);
    num /= d;
    den /= d;
}

// Rational: arithmetic and relational operators
Rational& Rational::operator += (const Rational& r) {
    num = getNum() * r.getDen() + getDen() * r.getNum();
    den = getDen() * r.getDen();
    simplify();
    return *this;
}

Rational Rational::operator + (const Rational& r) {
    Rational result = *this; // A copy of this object
    result += r;
    return result;
}

bool Rational::operator == (const Rational& r) {
    return getNum() == r.getNum() and getDen() == r.getDen();
}

bool Rational::operator != (const Rational& r) {
    return not operator == (r);
}

string Rational::to_str() const {
    string s(to_string(getNum()));
    if (not isInteger()) s += "/" + to_string(getDen());
    return s;
}