Object Oriented Programming

Inheritance: Introduction

2023/24

Lecture Outline

- Why do we need inheritance?
- Example
- Inheritance - the basic principle

Why do we need inheritance?

What is being addressed?

- Reusability
  - We do not want to rewrite (copy) source code that we have already written and tested.
- Extendibility
  - We want to extend (change) the source code that we have already…

Class Roles?

- Reusability and extendibility in the context of class usage can be understood as:
  - Combining with other classes, composition
  - Extension with new behavior
  - Modification of existing behavior

Composition x Inheritance

- By composition, we achieve that an object of one class is a composition of objects of other classes.
  - This is a "HAS" relationship.
- By inheritance, we achieve that the new class is an extension or a special case of an existing class (or multiple classes).
  - **This is an "IS" relationship.**

Example

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;
    Client *partner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);
    Account(int n, Client *o, Client *p);
    Account(int n, Client *o, Client *p, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    Client *GetPartner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

What is wrong with the *Account* class?

- An account with a partner is an extension of an account without a partner.
- An *account with a partner* is at the same time an *account*.
- We can use inheritance.
- How?

Parent Declaration

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

Child Declaration

```
class PartnerAccount : public Account
{
private:
    Client *partner;

public:
    PartnerAccount(int n, Client *o, Client *p);
    PartnerAccount(int n, Client *o, Client *p, double ir);

    Client *GetPartner();
};
```

Constructor Definitions

```
Account::Account(int n, Client *o)
{
    this->number = n;
    this->owner = o;
    this->balance = 0;
    this->interestRate = 0;
}

Account::Account(int n, Client *o, double ir)
{
    this->number = n;
    this->owner = o;
    this->balance = 0;
    this->interestRate = ir;
}
```

The *PartnerAccount* child uses the *Account* parent constructor to initialize the data members, to which it provides the necessary initialization values.

```
PartnerAccount::PartnerAccount(int n, Client *o, Client *p) : Account(n, o)
{
    this->partner = p;
}

PartnerAccount::PartnerAccount(int n, Client *o, Client *p, double ir) : Account(n, o, ir)
{
    this->partner = p;
}
```

Using (substitution)

```
int main()
{
    Account *a;
    PartnerAccount *pa;
    pa = new PartnerAccount(0, new Client(0, "Smith"), new Client(1, "Jones"));
    a = pa;

    cout << a->GetOwner()->GetName() << endl;
    //cout << a->GetPartner()->GetName() << endl;

    cout << pa->GetPartner()->GetName();

    getchar();
    return 0;
}
```

Bank with Two Types of Accounts

```
class Bank
{
private:
    Client * * clients;
    int clientsCount;

    Account** accounts;
    int accountsCount;

public:
    Bank(int c, int a);
    ~Bank();

    Client* GetClient(int c);
    Account* GetAccount(int n);

    Client* CreateClient(int c, string n);
    Account* CreateAccount(int n, Client *o);
    Account* CreateAccount(int n, Client *o, double ir);
    PartnerAccount* CreateAccount(int n, Client *o, Client *p);
    PartnerAccount* CreateAccount(int n, Client *o, Client *p, double ir);

    void AddInterest();
};
```

Should it work? And why?

```
int main()
{
    Account *a;
    PartnerAccount *pa;

    Bank *b = new Bank(100, 1000);
    Client *o = b->CreateClient(0, "Smith");
    Client *p = b->CreateClient(1, "Jones");
    a = b->CreateAccount(0, o);
    pa = b->CreateAccount(1, o, p);

    cout << a->GetOwner()->GetName() << endl;
    cout << pa->GetPartner()->GetName() << endl;

    cout << b->GetClient(1)->GetName() << endl;
    //cout << b->GetClient(1)->GetPartner() << endl;

    getchar();
    return 0;
}
```

Inheritance - the basic principle

Terminology

- Ancestor - descendant, direct ancestor-descendant
- Parent-child (daughter, son)
- Super (base) class - subclass

Inheritance - relationships

A

B

C

- A is a base class of class B; A is a parent of B; A is an ancestor of C
- B is a base class of class C; class B inherits from class A; B is a parent of C
- C inherits from B and A; class C is a child of class B; C is a descendant of A and a direct descendant of B.

Examples

- Vehicle - bicycle, motorcycle, car
- Person - user, administrator
- Collection - list, set

Is it Wrong?

- Car – Skoda
  - Skoda is BRAND of car.
- Tree – Pine
  - Pine is a SPECIES of a coniferous tree.

Generalization - specialization

- Do not confuse the relationship "*is an instance*" and "*inherits from*."
  - "Is an instance" is a relationship between a class and its object.
  - "Inherist from" is the relationship between classes.
- The inheritance defines the *GENERAL - SPECIAL* relationship.
- The inheritance should therefore represent a special case of the ancestor…
- … and the ancestor should represent the generalization of its descendants.

In other words…

- The ancestor defines the common behavior of all its descendants.
- Descendants can extend or modify (change) this behavior.
- Descendants cannot avoid this behavior.
- And that is why:
  - Everything is inherited with no exception!!!
  - The degree of information hiding is also inherited…

Composition vs. Inheritance

- Composition „HAS“ x inheritance „IS“.
- However:
  - Inheritance can be understood as a consequence of composition.
  - An instance of a descendant class contains everything that an instance of an ancestor class has.

Hierarchy

- When inheritance is used, class hierarchies are created.
- In our case, we work with single inheritance.
  - Each child has exactly one parent.
- A parent can have multiple children.
- In the case of single inheritance, this hierarchy is a tree.
- Do not confuse the hierarchy of objects (composition) and the hierarchy of classes (inheritance).

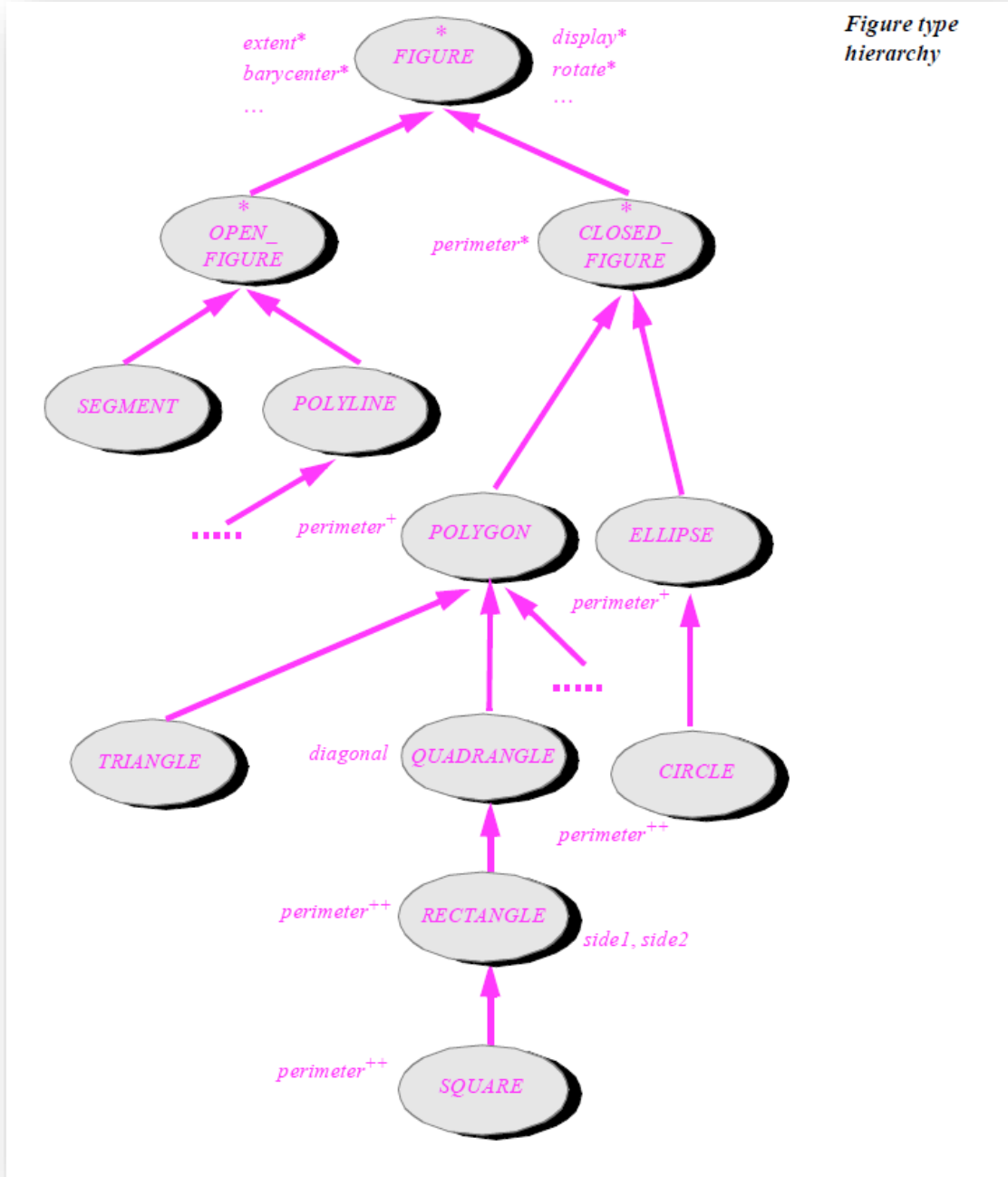
Figure type hierarchy
extent*
barycenter*
…
FIGURE
display*
rotate*
…
OPEN_ FIGURE
perimeter*
CLOSED_ FIGURE
SEGMENT
POLYLINE
perimeter+
POLYGON
ELLIPSE
perimeter+
TRIANGLE
diagonal
QUADRANGLE
CIRCLE
perimeter++
perimeter++
RECTANGLE
side1, side2
perimeter++
SQUARE

Liskov substitution principle

- Barbara Liskov 1987. *Data abstraction and hierarchy*.
- Bertrand Meyer. *Behavior invariants*.
- The descendant can always substitute its ancestor…
  - …because of their common behavior.
- The reverse is not true…

Initialization of a descendant

1. Object constructor is called.
2. Parent constructor is called.
3. Parent constructor is executed.
4. Object constructor is executed.

Seminar Assignments

- Implement the example from the presentation and create a bank with many clients and accounts. Focus on understanding the substitution principle and how constructors work in inheritance.

- Design and implement other single inheritance examples with extended common state and behavior, such as *Vehicle, Car, Truck*.

Seminar Questions

- Which roles do classes play in inheritance? Use the correct terminology.
- Explain the general relationship between the class from which it is inherited and the class which inherits.
- What is inherited, what is not and why?
- What do we mean by single inheritance?
- What Liskov substitution principle is, and how does it is applied in inheritance?
- How are constructors called and executed in inheritance?

Sources

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [459-467]