Object Oriented Programming

Inheritance: Behavior Change

2023/24

Lecture Outline

- Behavior extension
- Behavior change
- Example

Extension of Behavior

When we extend behavior…

- We can safely use what we already have.
- There is no problem in understanding how the object behaves.
- The object acts for itself…
- …or one of his ancestors.

Specialization-extension paradox

- The inheritance is a g*eneralization-specialization* relationship.
- The descendant is, therefore, a *special case* of each its ancestor.
- The paradox is that when extended, the *descendant can do more than any of its ancestors*.

…and so

- **The richer the behavior we consider, the fewer classes provide it.**
- In the inheritance hierarchy, the smallest common behavior is defined in a common ancestor.
- The terminal classes of this hierarchy (so-called leaves) have the richest behaviors (each slightly different).

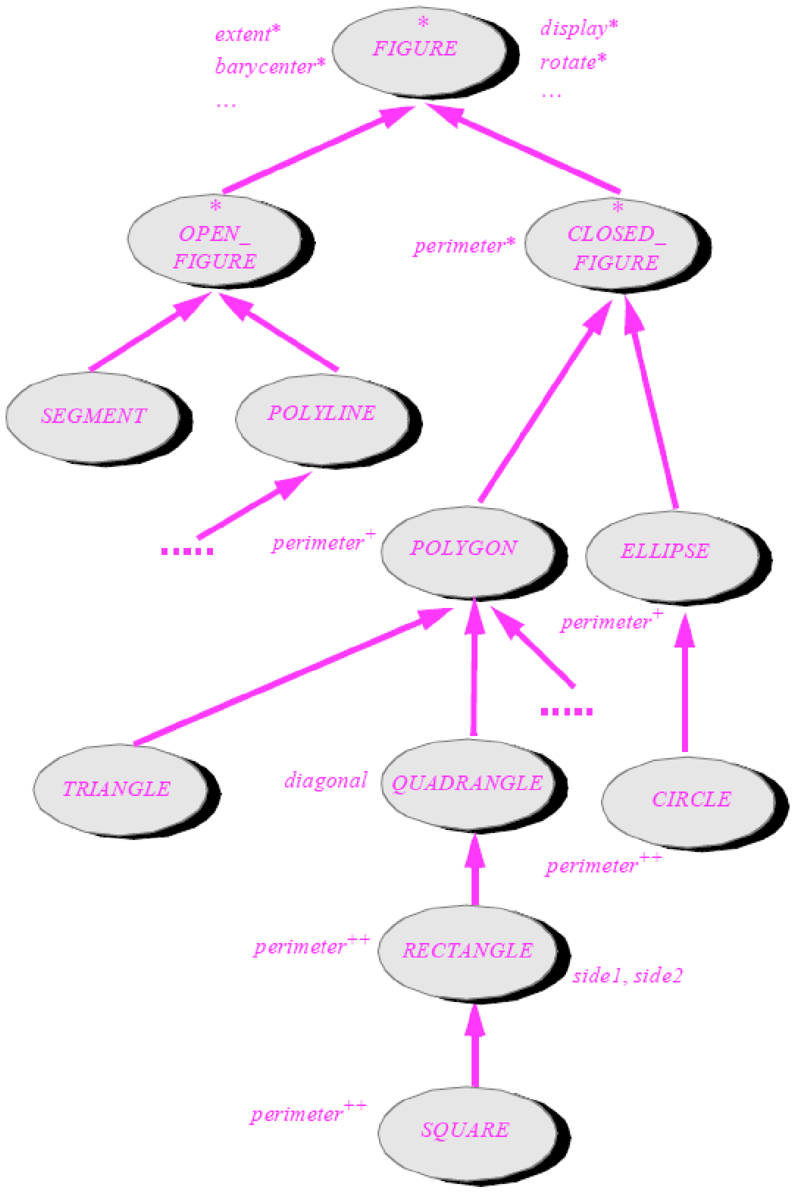


Figure type hierarchy

Wrong example

- The need for extension alone is not sufficient to use inheritance.
- E.g., in the relationship between a point and a circle, we might need to extend the point to work with the radius (new behavior).
- Is this sufficient to decide to use inheritance?

No!!!

- The **specialization** condition is not satisfied (a circle is not a special case of a point).

Changing of Behavior

Behavior Change

- If the behavior is declared in the ancestor, we can declare it again in the descendant.
  - Then, there are multiple methods of the same name.
- We then have to implement the declared behavior in the descendant (to make it executable).
- *The declared behavior does not have to be implemented in the ancestor.*

Overloading as Extension of Behavior

Overloading

- By overloading, we mean a situation where a given method has the same name but has:
  - different number of parameters,
  - different types of parameters,
  - different type of return value.
- However, overloading *is* **not a change of behavior**, even though the method has the same name.

Types of Overloading (summary)

- **The method name remains the same.**
- A different number of parameters.
- Different data types of parameters.
- Different return value (not in C ++).
- These can be combined.

Overriding/Shadowing

- By overriding/shadowing, we mean a situation where the descendant and ancestor methods have the same declaration (the same signature).

- The descendant also inherits the ancestor's method. Thus, it has two methods with the same declaration.

When to use overriding/shadowing?

- Constructors are a typical example of the use of *overloading*.

- A typical example of use *overriding/shadowing* is an actual change in child behavior.

- An example is the *withdraw* method in different types of bank accounts.

Example

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();
};
```

Shadowing

- We declare the *CreditAccount* class.

- We will shadow the *CanWithdraw* method.

- It has the same signature but a different definition.

- Consequence: In the *CreditAccount* class, the instance method *CanWithdraw* will be twice!!!

Child Declaration

```cpp
class CreditAccount : public Account
{
private:
    double credit;

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

    bool CanWithdraw(double a);
};
```

Definition

```
bool Account::CanWithdraw(double a)
{
    return (this->balance >= a);
}
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

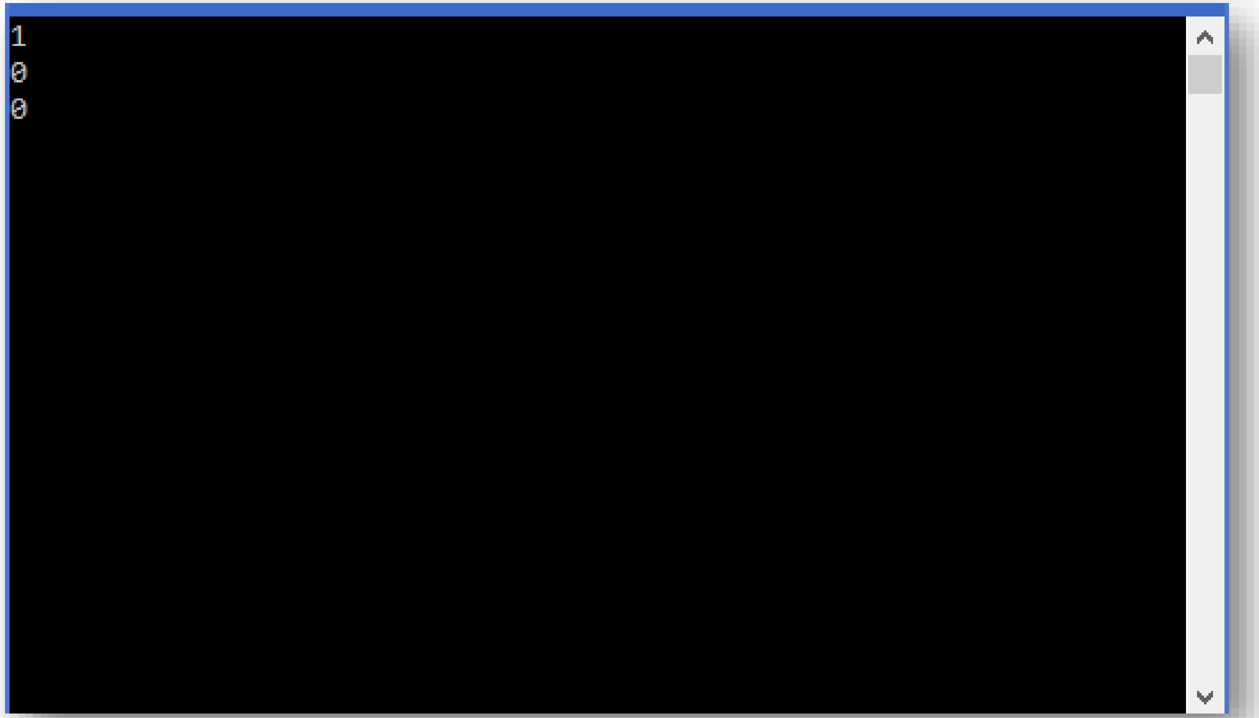
Definition (remainder)

```
bool Account::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

Usage

```cpp
int main()
{
    Client *o = new Client(0, "Smith");

    CreditAccount *ca = new CreditAccount(1, o, 1000);
    cout << ca->CanWithdraw(1000) << endl;

    Account *a = ca;
    cout << a->CanWithdraw(1000) << endl;

    cout << ca->Withdraw(1000) << endl;

    a = nullptr;
    delete ca;

    getchar();
    return 0;
}
```

Result?



Are we finished?

- No!!!

- How do we withdraw from the account (if we can) if we do not have access to the balance variable?

- What are our possibilities?

New extra method?

```
class CreditAccount : public Account
{
private:
    double credit;

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

    bool CanWithdraw(double a);
    bool Withdraw(double a);
};
```

We have a problem…

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

So, what are our possibilities?

- *Public* access to a data member?

- Encapsulation violation?

- Or else?

New Version of Parent Class

```
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();
};
```

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

    Client *owner;

protected:
    double balance;

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();
};
```

It works, but…

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

- We have the same code twice.
- We break the encapsulation.
- Different methods are used to substitute the ancestor with the descendant.

Result

Encapsulation Violation

- When behavior changes, you may need to work with the private part of the ancestor.
- This is, of course, a violation of the encapsulation and we must be aware of that…
- However, every reasonable rule has some exceptions.

Is it possible to call the ancestor method?

- It is the same as calling a static method ☹
- We call the original method from the child object.
- *Account::CanWithdraw(a);*

Seminar Assignments

- Implement examples from the presentation. Focus on the shadowing, use "protected" section.

- Design and implement a simple inheritance hierarchy of geometric figures that will share the "*Area*" and "*Perimeter*" methods. Take advantage of the shadowing and analyze the behavior when using the substitution principle.

Seminar Questions

- What do we mean by the paradox of specialization-extension?
- Give right and wrong examples of the "generalization-specialization" relationship.
- What do we mean by changing behavior in inheritance?
- What do we mean by overloading? Is it an extension or a change of behavior?
- What are different types of overloading?
- What do we mean by overriding/shadowing? Is it an extension or a change of behavior?
- What principle do we violate if we use "protected" and why?
- What is the main problem of the change of behavior in inheritance?
- Describe how the different levels of access to class members work in practice.
- How does the use of "protected" affect the ancestor-descendant relationship?

Sources

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