

Programming Language Concepts

Object Oriented Prog: Relations

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

1 Class Relations

2 Aggragate

3 Composition

- Integrity of Contained Objects

4 Generalization/Inheritance

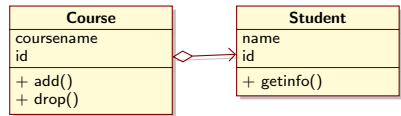
- Integrity of Superclass
- Member Hiding

5 Multiple Inheritance

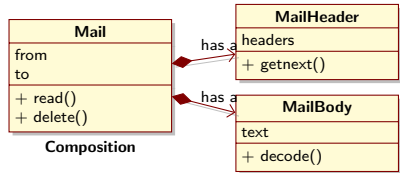
- Virtual base class

Class Relations

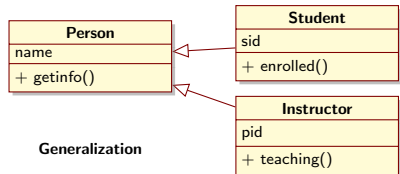
- In Object Oriented paradigm objects interact in order to solve a problem.
- Basic class relations:
 - Aggregate (“has a”)
 - Composition (“has a”)
 - Generalization (inheritance, “is a”)
- Other associations/relations exist.
- When two classes have such a relation, one **depends** on the other.



Aggregation



Composition



Generalization

Aggragate

- Class A can have 0 or more instance of class B
- Lifetime of class B objects are independent of class A
- Catalog relationship. In terms of [references](#).
- Members of class B are regular objects in scope of A
they are not in scope of A. So private members ... ?

```
class Course {  
    char name[40];  
    int no;  
    List students;  
public:  
    void register(Student &a) {  
        student.insert(&a);  
    };  
} ceng242 ;
```

```
void Student {  
    char name[30];  
    int no;  
public:  
    void add(Course &c) {  
        c.register(*this);  
    }  
};
```

Composition

- Class A can have 0 or more instance of class B
- Lifetimes of class B objects depend on the class A object
- Class B objects are destroyed when A is destroyed.
- Members of class B are regular objects in scope of A
they are not in scope of A as in aggregate.

```
class FrameBox {
    Shape frame;
    String text;
    double coordx, coordy;
public:
    Framebox(Frame &f,
              String &t) {
        ...}
    void draw() {
        frame.draw(); text.draw();
    }
} ceng242 ;
```

```
class Shape {
    enum Type {Circle, Rect} type;
    double sizex, sizey;
public:
    void draw();
};

class String {
    ...
};
```

Framebox

Shape frame	Type type	sizeof(int)
	double coordx	sizeof(double)
	double coordy	sizeof(double)
String text
double coordx	sizeof(double)	
double coordy	sizeof(double)	

- Container class vs. contained classes
- Composition nests storage of contained classes into container class.
- frame and text are regular object variables in member functions of Framebox
- Integrity of contained objects?

```

class Student {
    char name[40];
    int id;
public:
    Student() { name[0]=0; id=0;}
    void setnameid(const char *s,int i);
    ...
};

class StudentArr {
    Student *content;
public:
    StudentArr(int size) {
        content=new Student[size];
    }
    ~StudentArr() { delete [] content;}
    Student &operator[](int i) {
        return content[i];
    }
}
...
StudentArr a(10);
a[5].setnameid("onur",55717);

```

Integrity of Contained Objects

```
class A {
    int x;
public:
    A(int a) { x=a; }
};
class B {
    int y;
public:
    B(int a) { y=a; }
};
```

```
class C {
    int c;
    A a;
    B b;
public:
    C(int x,y,z):a(x),b(y) {
        c=z; /*can refer a,b */
    }
    ~C() { /*can refer a,b */ }
};
```

- When constructors called? Tip: Container class constructor may refer to the contained objects.
- When destructors called? Tip: Container class destructor may refer to the contained objects.

- Constructors of contained objects called just before the body of container constructor executed.
- Destructors of contained objects called just after the container destructor called.
- Container constructor can pass arguments to member object constructors.

```
ACons(int x):a(x),b(x),c(x) {...}
```

- `friend` declaration can be used if the objects need to access others private member.

Generalization/Inheritance

- Class **Circle** is a **Shape** but has extra features.
- It has all members of **Shape** plus specific ones.
- **Circle** extends **Shape**
- **Shape** is super class of **Circle**
- **Shape** is more general, **Circle** has more information

```
class Shape {  
    double x,y;  
public:  
    Shape(double a, double b);  
    void draw();  
};  
  
class Circle: public Shape {  
    double radius;
```

```
public:  
    void draw();  
};  
  
class Square: public Shape {  
    double width;  
public:  
    void draw();  
};
```

Circle

Shape double x	sizeof(double)
double y	sizeof(double)
double radius	sizeof(double)

- There is an inherent **Shape** object in each **Circle** object.
- $\text{Env}(\text{Circle}) = \text{Env}(\text{Shape}) \cup \text{Members specific to Circle}$
- All members are inherited. They **are** in the scope of the subclass.
- How about their accessibility, protection?
- Two new thing: **protected** label, **derivation label**
- A subclass can access **protected** members of the upper classes.
- **derivation label** is a filter defining how members of superclass interpreted when used through subclass (object of subclass or further derivations from subclass)

```

class A {
private:    int a;
protected: int b;
public:    int c;
    void Amember() { ① }
} Aobj;
class B: DLABEL A { // DLABEL=public/protected/private
    void Bmember() { ② };
} Bobj;
... Aobj.③;
... Bobj.④;
class C: public B {
    void Cmember() { ⑤ } };

```

	①	②	③			④		⑤			
a				DLABEL		a	b	c	a	b	c
b				private							
c				protected							
				public							

- DLABEL is only significant outside of the derived class
- protection is minimum of original label and DLABEL

```

class A {
private:    int a;
protected: int b;
public:    int c;
    void Amember() { ① }
} Aobj;
class B: DLABEL A { // DLABEL=public/protected/private
    void Bmember() { ② };
} Bobj;
... Aobj.③;
... Bobj.④;
class C: public B {
    void Cmember() { ⑤ } };

```

	①	②	③		④			⑤		
				DLABEL	a	b	c	a	b	c
a	✓	×	×	private	×	×	×	×	×	×
b	✓	✓	×	protected	×	×	×	×	✓	✓
c	✓	✓	✓	public	×	×	✓	×	✓	✓

- DLABEL is only significant outside of the derived class
- protection is minimum of original label and DLABEL

- The inherent superclass object should have a valid value.
- Constructors/Destructors should be called

```
class A {  
    int x;  
public:  
    A(int a) { x=a;}  
    ~A() { ... }  
};  
class B : public A {  
    int y;  
public:  
    B(int a):A(a) { y=a;}  
    ~B() { ... }  
};
```

- Similar to contained objects:
Superclass constructor called just before class constructor
Superclass destructor called just after class destructor

Member Hiding

- members of the subclass hides member of the superclass with same name
- but superclass member still exists
- Scope operator can be used to access the member

```
class A {  
protected:  
    int x;  
public:  
    int get() {return x};  
} Aobj;  
class B : public A {  
    int x;  
public:  
    int get() {return x+A::x}  
} Bobj;  
...  
cout << Bobj.get() << Bobj.A::get() ;
```

Multiple Inheritance

- Can a class be derived from two superclasses?

Multiple Inheritance

- Can a class be derived from two superclasses?
- Land vehicle+Water vehicle → Hovercraft

Multiple Inheritance

- Can a class be derived from two superclasses?
- Land vehicle+Water vehicle → Hovercraft
- Student+Instructor → A lecturer still having PhD

Multiple Inheritance

- Can a class be derived from two superclasses?
- Land vehicle+Water vehicle → Hovercraft
- Student+Instructor → A lecturer still having PhD
- A class hierarchy for vehicle types, a class hierarchy for engines:
A boat with diesel engine, a car with electrical engine or hybrid engine

Multiple Inheritance

- Can a class be derived from two superclasses?
- Land vehicle+Water vehicle → Hovercraft
- Student+Instructor → A lecturer still having PhD
- A class hierarchy for vehicle types, a class hierarchy for engines:
A boat with diesel engine, a car with electrical engine or hybrid engine
- Multiple inheritance is necessary in some rare cases. C++ provides it, Java avoids it and uses [Interfaces](#) for essential functionality similar to multiple inheritance.

```

class Shape {
    int x,y;
public:
    Shape(int a, int b) { x=a; y=b;}
    ~Shape() { ... }
};

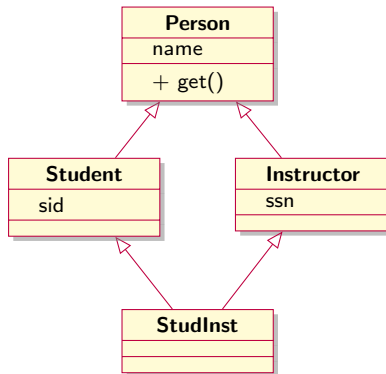
enum LineStyle {None, Solid, Dashed, Dotted, Double}
enum FillStyle {None, Full, Half, Pattern}
class ShapeAttr {
    LineStyle ls; double lw; FillStyle fill;
public:
    ShapeAttr(LineStyle a, double b, FillStyle c) {
        ls=a;lw=b;fill=c;}
    ~ShapeAttr() { ... }
};

class Circle: public Shape, public ShapeAttr {
    int radius;
public:
    Circle(int a, int b, int c, LineStyle d,
           double e, FillStyle f):Shape(a,b),ShapeAttr(d,e,f) {
        radius=c;
    }
}

```

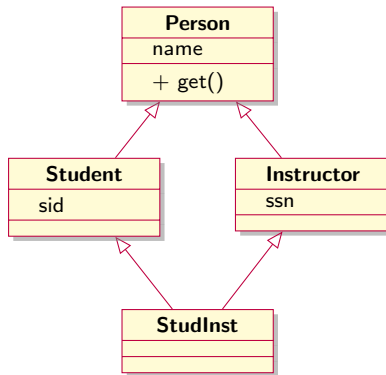
Diamond Problem

- Multiple inheritance may cause same super class duplicated in the resulting class



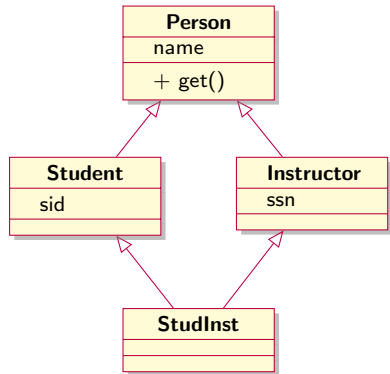
Diamond Problem

- Multiple inheritance may cause same super class duplicated in the resulting class
- Causes ambiguity.
StudInst contains two Person's
get() call refers to which one?
What's the name ?



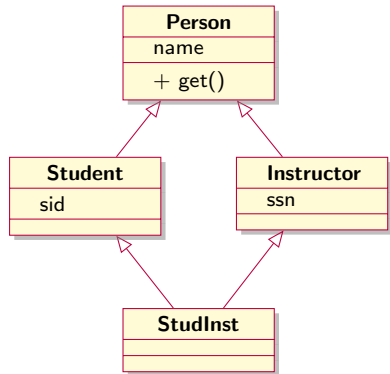
Diamond Problem

- Multiple inheritance may cause same super class duplicated in the resulting class
- Causes ambiguity.
`StudInst` contains two `Person`'s `get()` call refers to which one?
What's the `name` ?
- Ambiguity can be solved by scope operator:
`Student::name` VS `Instructor::name`



Diamond Problem

- Multiple inheritance may cause same super class duplicated in the resulting class
- Causes ambiguity.
`StudInst` contains two `Person`'s `get()` call refers to which one?
What's the `name` ?
- Ambiguity can be solved by scope operator:
`Student::name` VS `Instructor::name`
- But a person with two names?
Do we need that redundancy?
NO!



Virtual base class

■ `virtual` keyword used

in inheritance gets only a single copy of base class in subclasses.

```
class Person {
    char name[40];
public: Person(char *s) {...}
};
class Student: virtual Person {
    int id;
public: Student(char *s, int i):Person(s) {...}
};
class Instructor: virtual Person {
    int ssn;
public: Instructor(char *s, int i):Person(s) {...}
};
class StudInst:public Student, public Instructor {
public: StudInst(char *s, int a, int b)
        :Person(s),Student(s,a),Instructor(s,b) {...}
};
```

- `virtual` keyword is for subclasses
- It is an overloaded keyword. We also have virtual member functions which is completely different.
- Multiple inheritance is not essential feature in OOP.
- There are ways to live without it. Assume two hierarchies with M and N classes. First is under `Vehicle`, second is `Engine`
- **Bridge pattern** Put a `Engine*` member in `Vehicle`
- **Nested classes** Create all $M \times N$ possibilities derived from `Vehicle`
- Such cases are rare and primary advantage of inheritance is **Polymorphism**