Object-Oriented Programming: Inheritance

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

Inheritance

- Why inheritance is a good idea?
- How inheritance works?
- Public, protected, and private inheritances
- Inheritance and constructors
- Inheritance and destructors
- Multiply-derived classes
- Overriding functions

- Design rules of inheritance

The Problem

- You have a student class defined below for a part of a database.

```
class StudentT {
  public:
    StudentT();
    void setData(char *inName, int inAge);
    int getAge() const;
    char *getName() const;
  private:
    char *name;
    int age;
};
```

You decided to add an additional field for graduate students.

```
int getStipend();    // public member function
int stipend;         // private data member
```

- The problem:
  - You have to modify the existing class (maintenance)
  - The stipend is inappropriate for undergraduates.

Solution to the Previous Problem

- You can create a brand new class called *GraduateT* but most code in StudentT will be duplicated.
- A better solution would be to declare a class GraduateT that is *derived* from StudentT. StudentT is called the *base* class.

```
class GraduateT: public StudentT {//ignore public for now
  public:
    GraduateT(char *inName, int inAge, int inStipend);
    int getStipend() const; // new member function
  private:
    int stipend; // new field
};
GraduateT::GraduateT(char *inName, int inAge, int inStipend)
    :stipend(inStipend) {
    setData(inName,inAge); // function in StudentT
}
int GraduateT::getStipend() const {
    return stipend;
}
```

```
void main() {
    GraduateT student("Tim", 20, 2000);
    cout << student.getName() << ...;
}
```

Access Control Through Inheritance (I)

- GraduateT inherits all public members of StudentT as public functions.
- GraduateT also has internal copied of all private data members from StudentT but *no* access even in member functions.

```
int GraduateT::getStipend() const {
    if (age>30) // illegal
        return 0;
    return stipend;
} // getAge() is legal
```

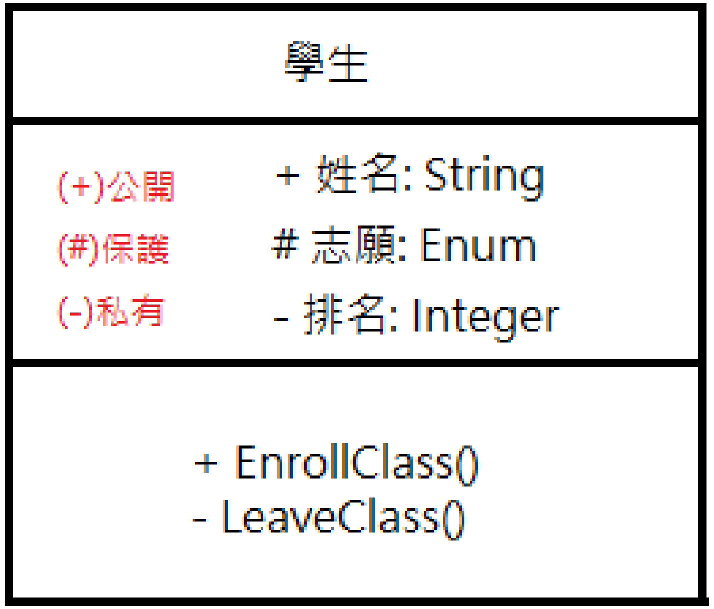
Access Control Through Inheritance (II)

- The base class can choose to give the inherited class access to "private" data by declaring the data protected in StudentT.

```
class StudentT {
  public:
   StudentT();
   void setData(char *inAame, int inAge);
   int getAge() const;
   char *getName() const;
  protected:
   char *name;
   int age;
};
```

Protected members are like private members except that they are accessible in the derived class.

UML (Unified Modeling Language)



學生

(+)公開 + 姓名: String

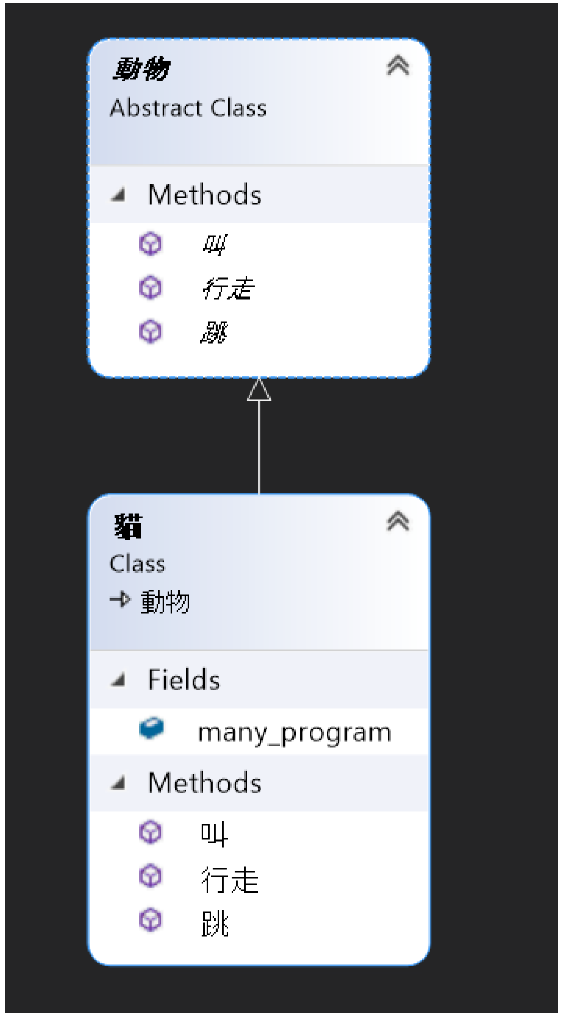
(#)保護 # 志願: Enum

(-)私有 - 排名: Integer

+ EnrollClass()
- LeaveClass()

類別屬性區域 (類別變數)

類別方法



動物
Abstract Class

Methods
- *叫*
- *行走*
- *跳*

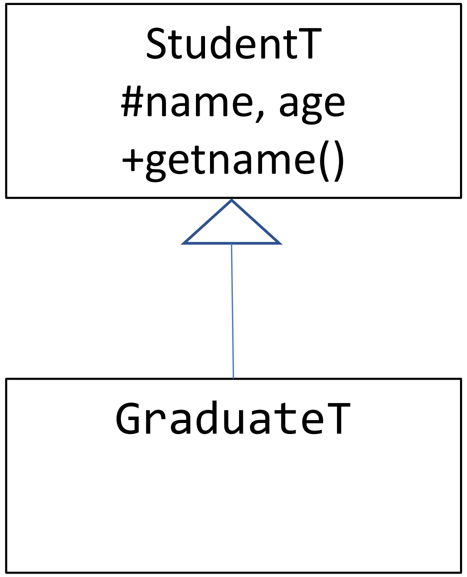
貓
Class
→ 動物

Fields
- many_program

Methods
- 叫
- 行走
- 跳

Inheritance

Accessibility in public Inheritance



StudentT
#name, age
+getname()

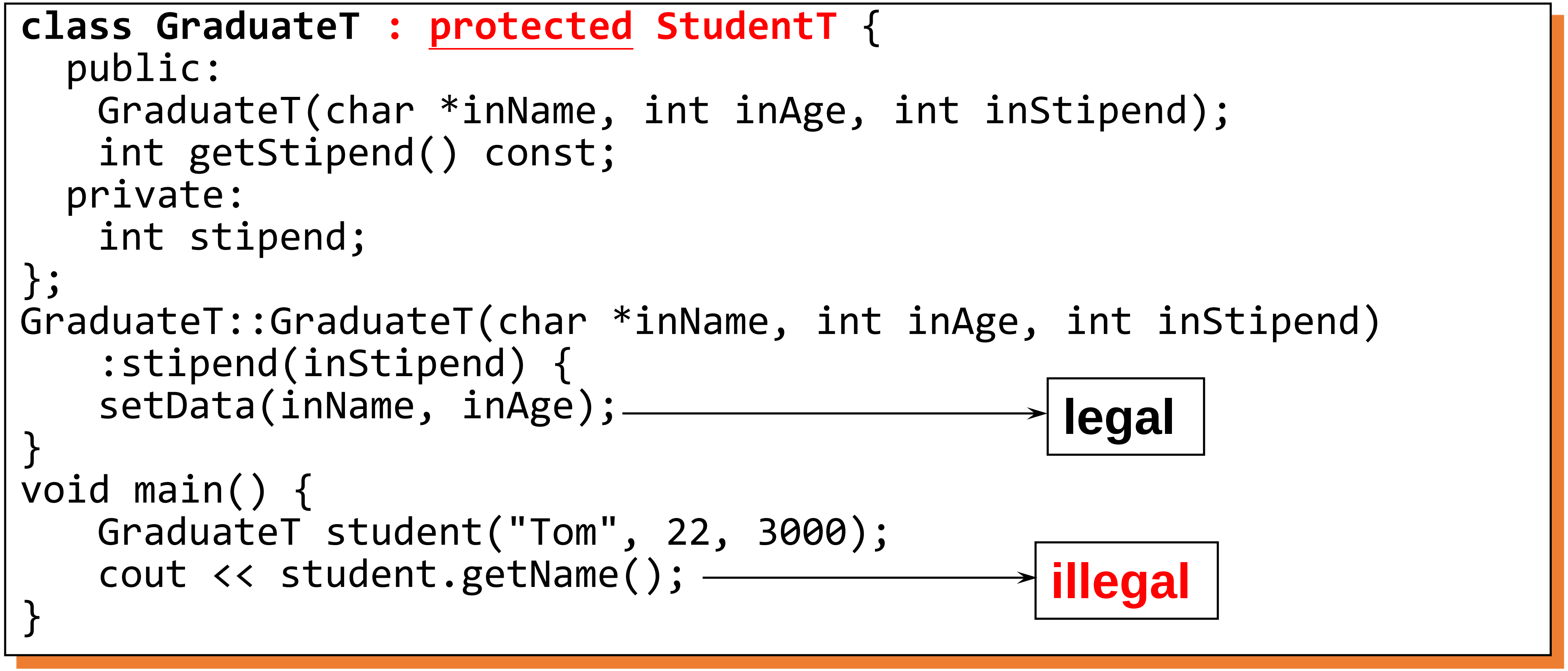
GraduateT

Accessibility in public Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Protected Access Specifier

- Protected inheritance: all public and protected members of StudentT are protected in GraduateT.



```
class GraduateT : protected StudentT {
  public:
    GraduateT(char *inName, int inAge, int inStipend);
    int getStipend() const;
  private:
    int stipend;
};
GraduateT::GraduateT(char *inName, int inAge, int inStipend)
    :stipend(inStipend) {
    setData(inName, inAge);                          legal
}
void main() {
    GraduateT student("Tom", 22, 3000);
    cout << student.getName();                      illegal
}
```

Private Access Specifier (I)

- Private inheritance: all public and protected members of StudentT become *private* in GraduateT.

```
class GraduateT : private StudentT {
  public:
    GraduateT(char *inName, int inAge, int inStipend);
    int getStipend() const;
  private:
    int stipend;
};
```

Private Access Specifier (II)

- Private inheritance would function like protected inheritance in this case. But further derivation from GraduateT would not be able to access any elements from StudentT.

- How often are protected and private inheritance used?

  Protected: unusual private: less common

- Default setting: *private*. So, be careful.

Inheritance and Constructors

- Redefine the class student so that there is no StudentT() and setData().

```
class StudentT {
  public:
    StudentT(char *inName, int inAge);
    int getAge() const;
    char *getName() const;
  private:
    char *name;
    int age;
};
```

```
GraduateT::GraduateT(char *inName, int inAge, int inStipend)
    :stipend(inStipend) {
}
```

Error: cannot construct base class 'StudentT'

- A base class *must* be constructed before the inherited class.

```
GraduateT::GraduateT(char *inName, int inAge, int inStipend)
    :StudentT(inName,inAge), stipend(inStipend) {
}
```

Only through base constructor. Calling `age(inAge)` is illegal.

Inheritance and Destructors

- The compiler automatically calls each destructor in the opposite order of the constructors.
- Assume that we place some printing statements in the constructors and destructors of these two classes.

```
void main() {
    GraduateT student("Tom", 22, 4000);
    cout << student.getName() << "is" << student.getAge() <<
    " years old and has a stipend of " <<
    student.getStipend() << "dollars\n";
}
```

Output:

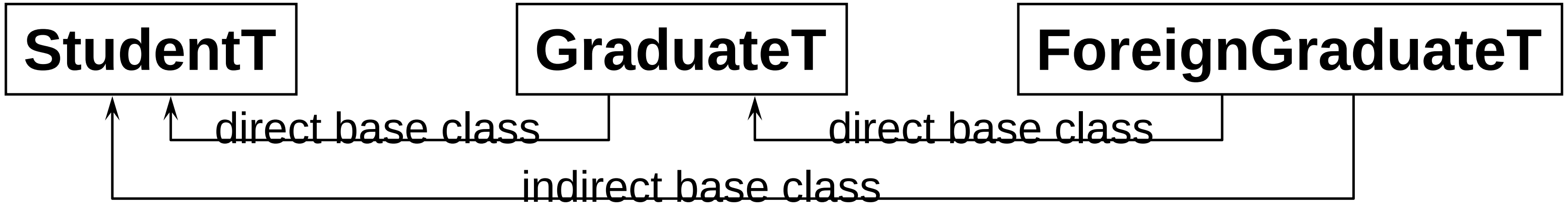
```
In student constructor
In graduate constructor
Tom is 22 years old and has a stipend of 4000 dollars.
In graduate destructor
In student destructor
```

Multi-level Derived Classes

- Derive another class, called ForeignGraduteT from GraduateT

```
class ForeignGraduateT : public GraduateT {
  public:
    ForeignGraduateT(char *inName, int inAge, int inNationality);
    ~ForeignGraduateT();
    char *getNationality();
  private:
    char *nationality;
};
ForeignGraduateT::ForeignGraduateT(char *inName, int inAge, char *inNationality)
: GraduateT(name, age, 0){
    nationality = new char[strlen(inNationality)+1];
    strcpy(nationality, inNationality);
}
```

Note: only the direct base class needs to be called



Overriding in Inheritance

- What happens if a derived class has a member function with the same name as one in the base class?

The function in the derived class *overrides* the one in the base class. This allows the derived class to *redefine* the inherited behavior.

```
void StudentT::display() const {
    cout<<getName()<<" is "<<getAge()<<"years old.\n";
}
void GraduateT::display() const {
    cout<<getName()<<" is "<<getAge()<<"years old.\n";
    cout<<“He/She has a stipend of "<<getStipend()<<"dollars\n";
}
void main() {
    StudentT student1("Mary", 20);
    GraduateT student2("Joy", 24, 4000);
    student1.display();
    student2.display();
}
```

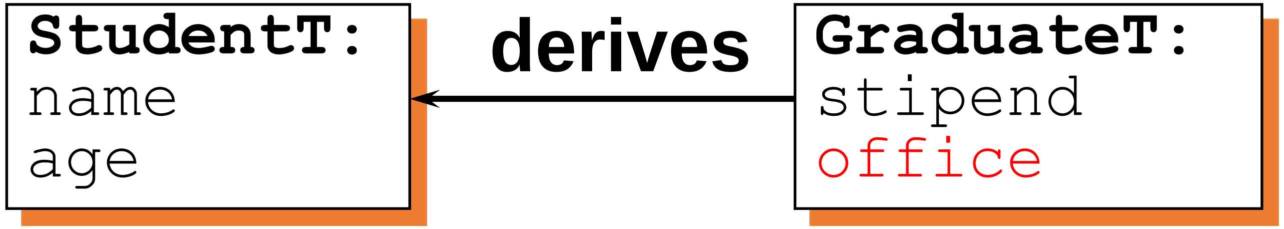
can be replaced by Student::display();

Output:

```
Mary is 20 years old.
Joy is 24 years old.
He/She has a stipend of 4000 dollars.
```

Inheritance Design Considerations (I)

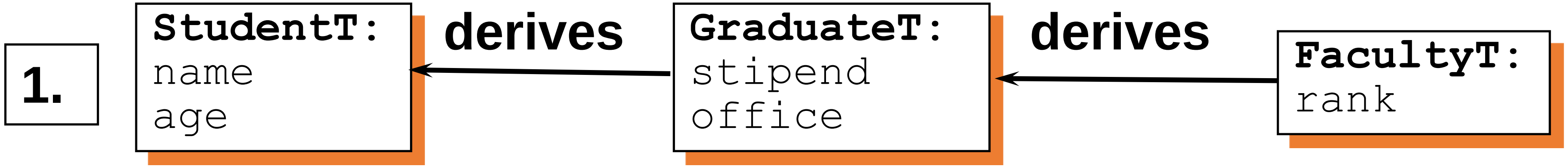
- In the previous example, assume we have following data:

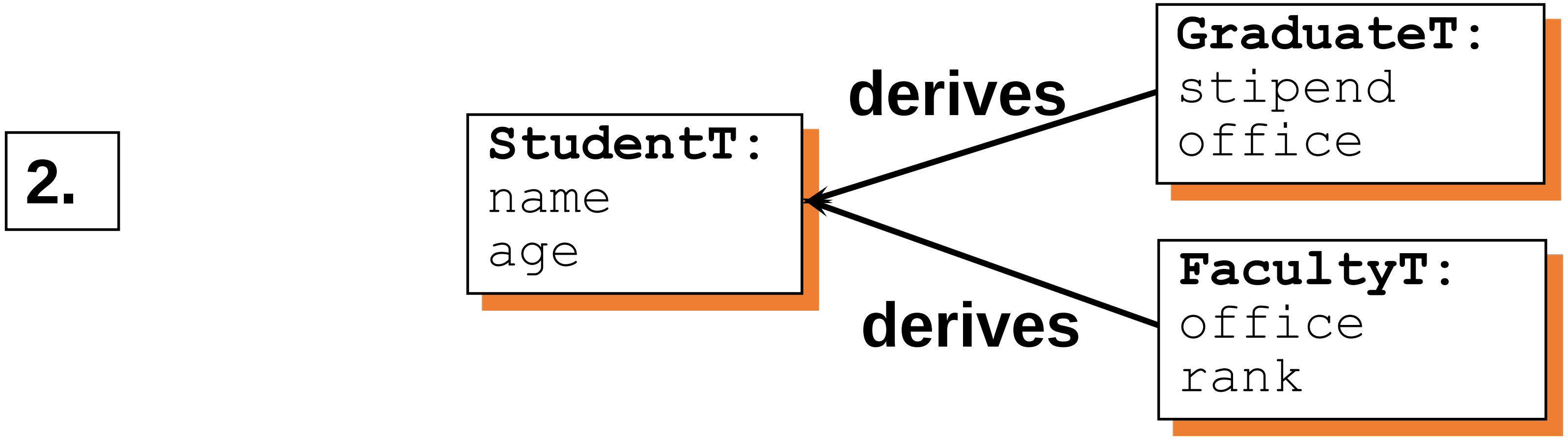


- We would like to add a class FacultyT that have *name*, *age*, and *office* but they have *no stipend*. In addition, FacultyT has a *rank*. Should we derive from StudentT or GraduateT? We can derive FacultyT either from GraduateT or from StudentT.

Inheritance Design Considerations (II)

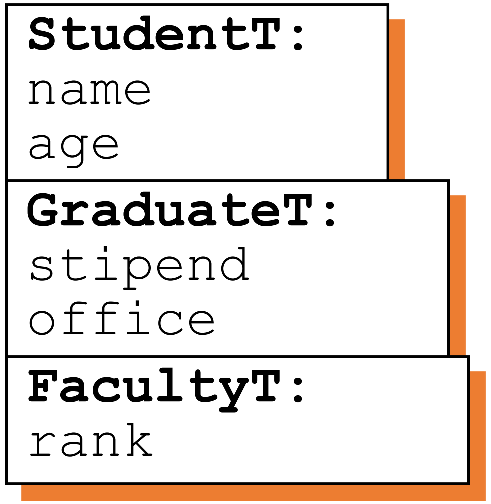
- Two design alternatives:





Inheritance Design: Exploring Solution #1

```
class GraduateT : public StudentT {
  public:
    GraduateT(char *inName, int inAge, int inStipend,
              char *inOffice);
    ~GraduateT();
    void display();
  protected:
    char *getOffice() const;
  private:
    int getStipend();
    int stipend;
    char *office;
};
class FacultyT : public GraduateT {
  public:
    FacultyT(char *inName, int inAge, char *inOffice,
             char *inRank);
    ~FacultyT();
  private:
    char *rank;
};
```



StudentT:
name
age

GraduateT:
stipend
office

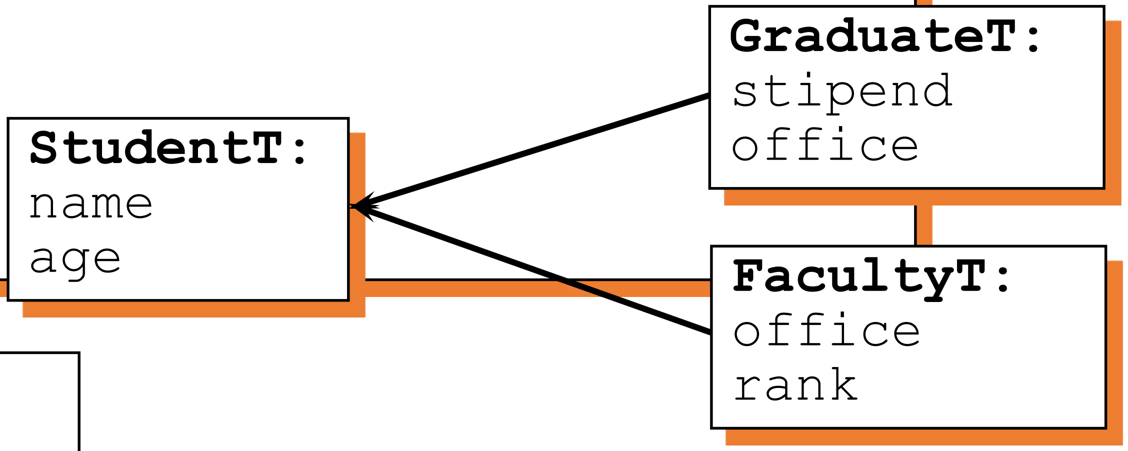
FacultyT:
rank

Problem:

Why should FacultyT inherit a stipend?

Inheritance Design: Exploring Solution #2

```
class FacultyT : public StudentT {
  public:
    FacultyT(char *inName, int inAge, char *inOffice,
                 char *inRank);
    ~FacultyT();
  private:
    char *getOffice() const;
    char *office;
    char *rank;
};
```



GraduateT:
stipend
office

StudentT:
name
age

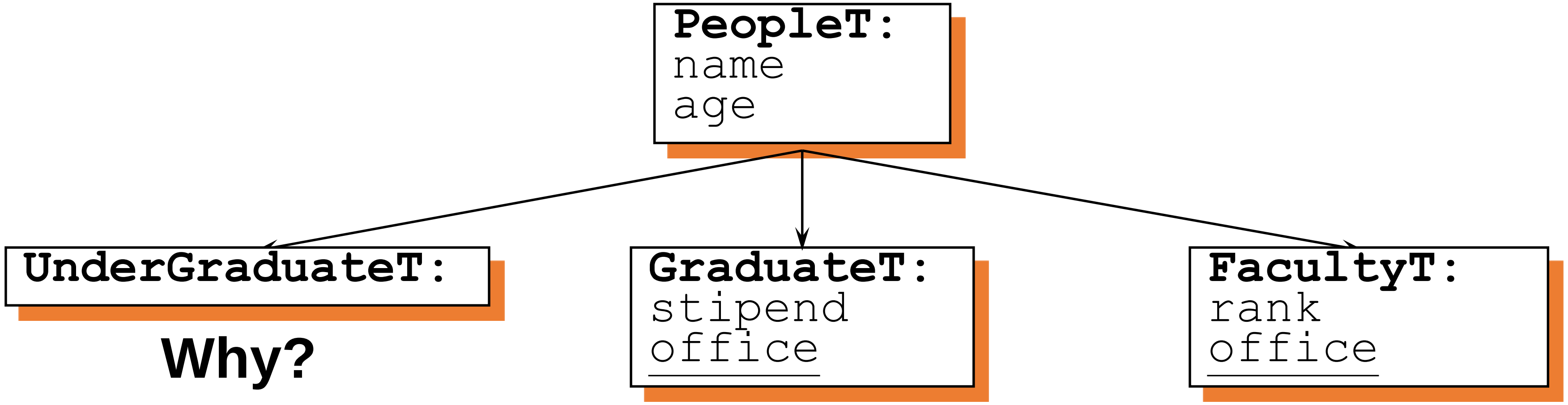
FacultyT:
office
rank

Problem:
Duplicating code in GraduateT

- The situation will get worse when you add to StudentT more data members that are irrelevant to FacultyT. For example, advisor, club, etc.

Inheritance Design: A Better Design (1)

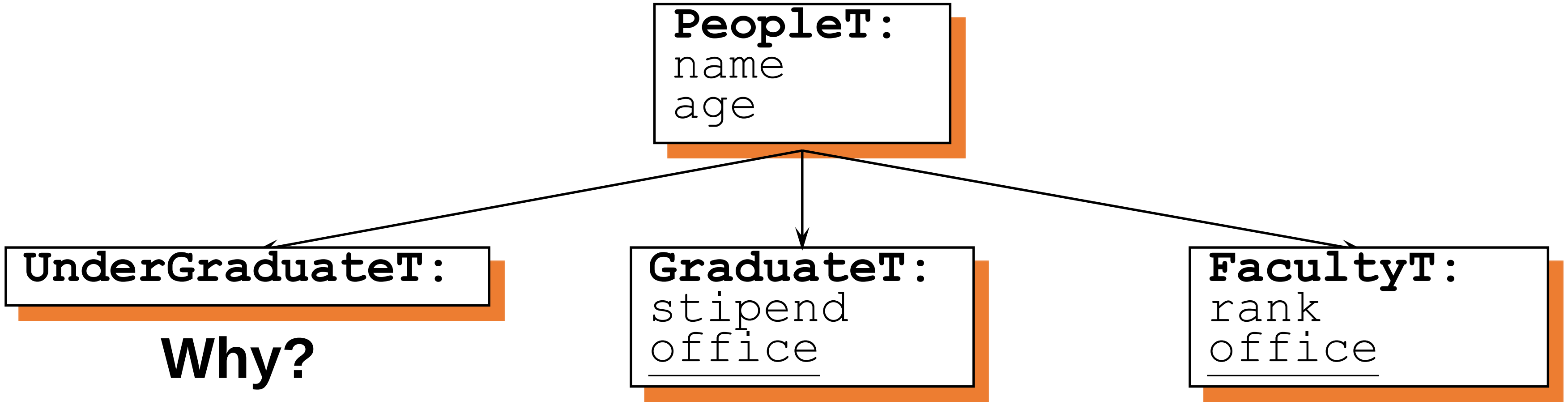
- Create a *people* class and put everything common there.



Why?

Inheritance Design: A Better Design (2)

- Create a *people* class and put everything common there.



PeopleT:
name
age

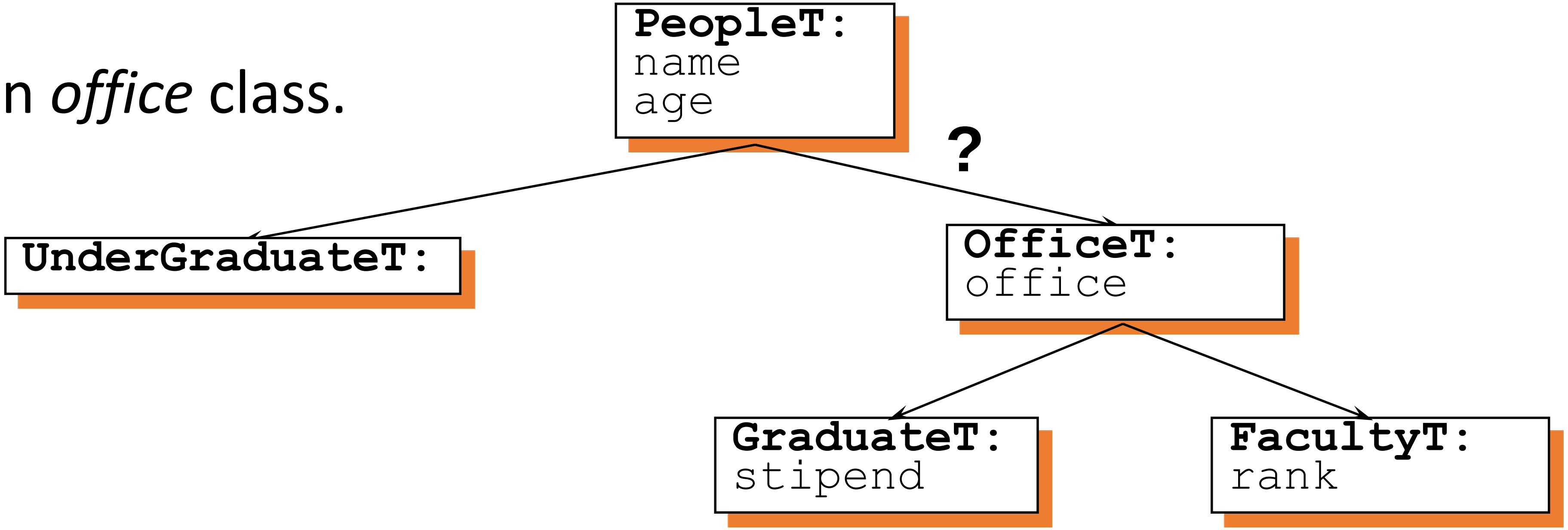
UnderGraduateT:

Why?

GraduateT:
stipend
office

FacultyT:
rank
office

- Add an *office* class.



PeopleT:
name
age

?

UnderGraduateT:

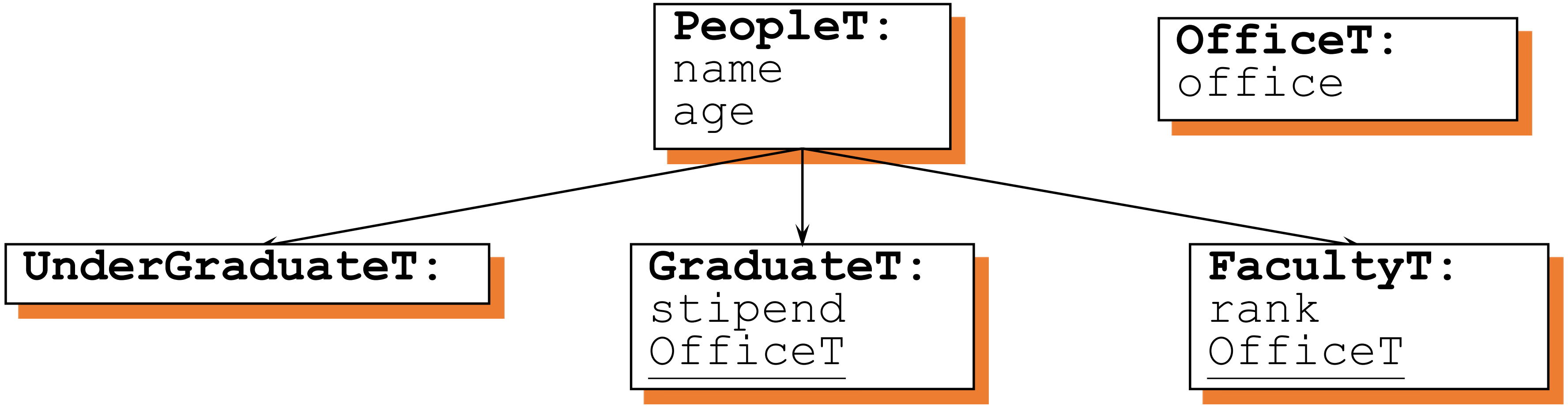
OfficeT:
office

GraduateT:
stipend

FacultyT:
rank

Inheritance Design: Final Solution

- Separate the OfficeT class and make GraduateT and FacultyT contain the class.

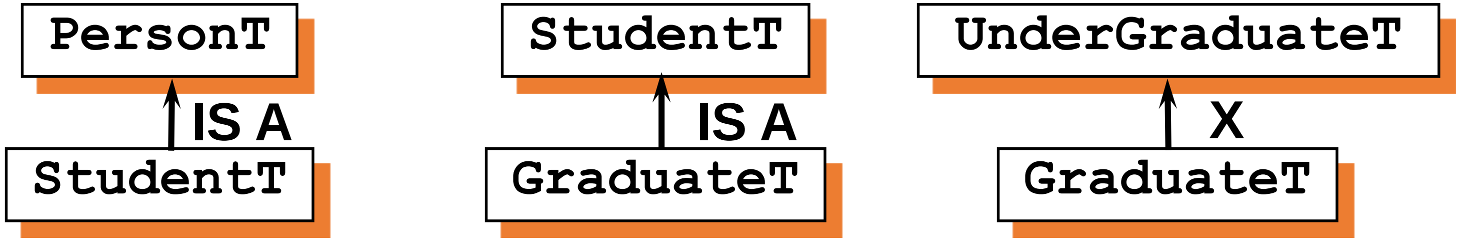


- With this design we have no duplication of code and no class inherits or accesses inappropriate members.
- We have duplicate the OfficeT fields in the GraduateT and FacultyT classes. *Why is this a better design?*

Design Rules for Inheritance

- The prime directive of inheritance: *is-a* relationship.

  Class A should only be derive from class B if class A *is a* type of class B. For example, a student *is a* person.



PersonT

IS A

StudentT

StudentT

IS A

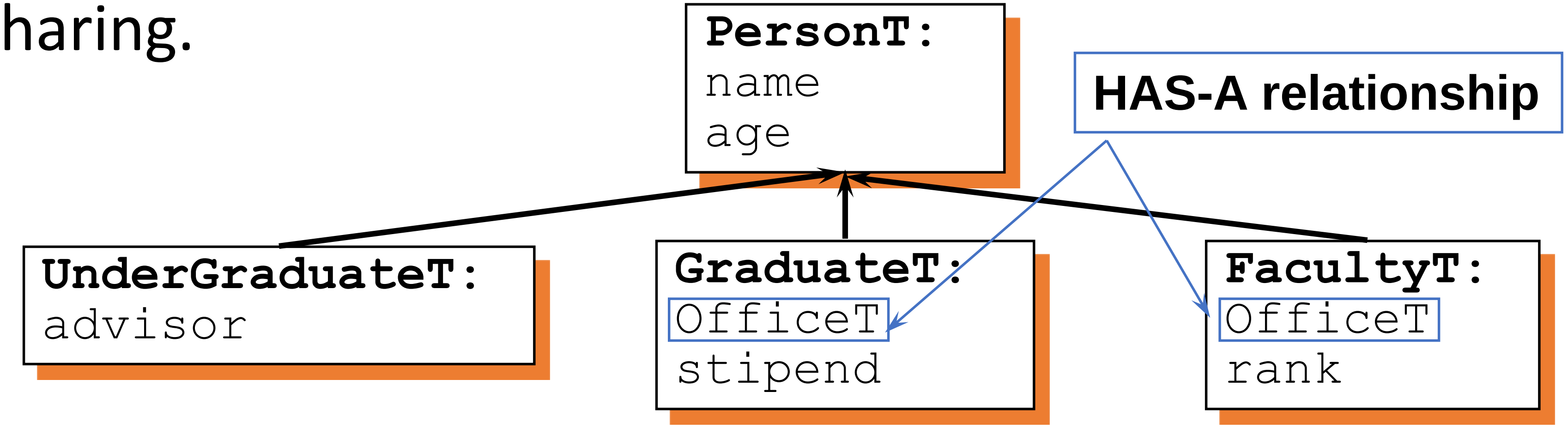
GraduateT

UnderGraduateT

X

GraduateT

- Common code between classes can be *shared* by abstracting it away into base classes. But never violate the prime directive for the sake of code sharing.



PersonT:
name
age

HAS-A relationship

UnderGraduateT:
advisor

GraduateT:
OfficeT
stipend

FacultyT:
OfficeT
rank

Dubious Examples (I)

- The derived class always extends the base class, not the other way around.

  Example: A stack *is a* form of linked list. So, what's wrong?

  Explanation: you can do things with a linked list that you should not do with a stack, e.g., Inserting an element anywhere in the list.

  Correct solution: The stack class should contain a linked list.

Dubious Examples (II)

- Inheritance must not allow operations which could violate a *class invariant*.

  Example: derive a file pathname class from a string class.

  ↑

  Explanation: A file pathname cannot exceed a certain length on most OS. String operations inherited from the base class would allow the client to make an error.

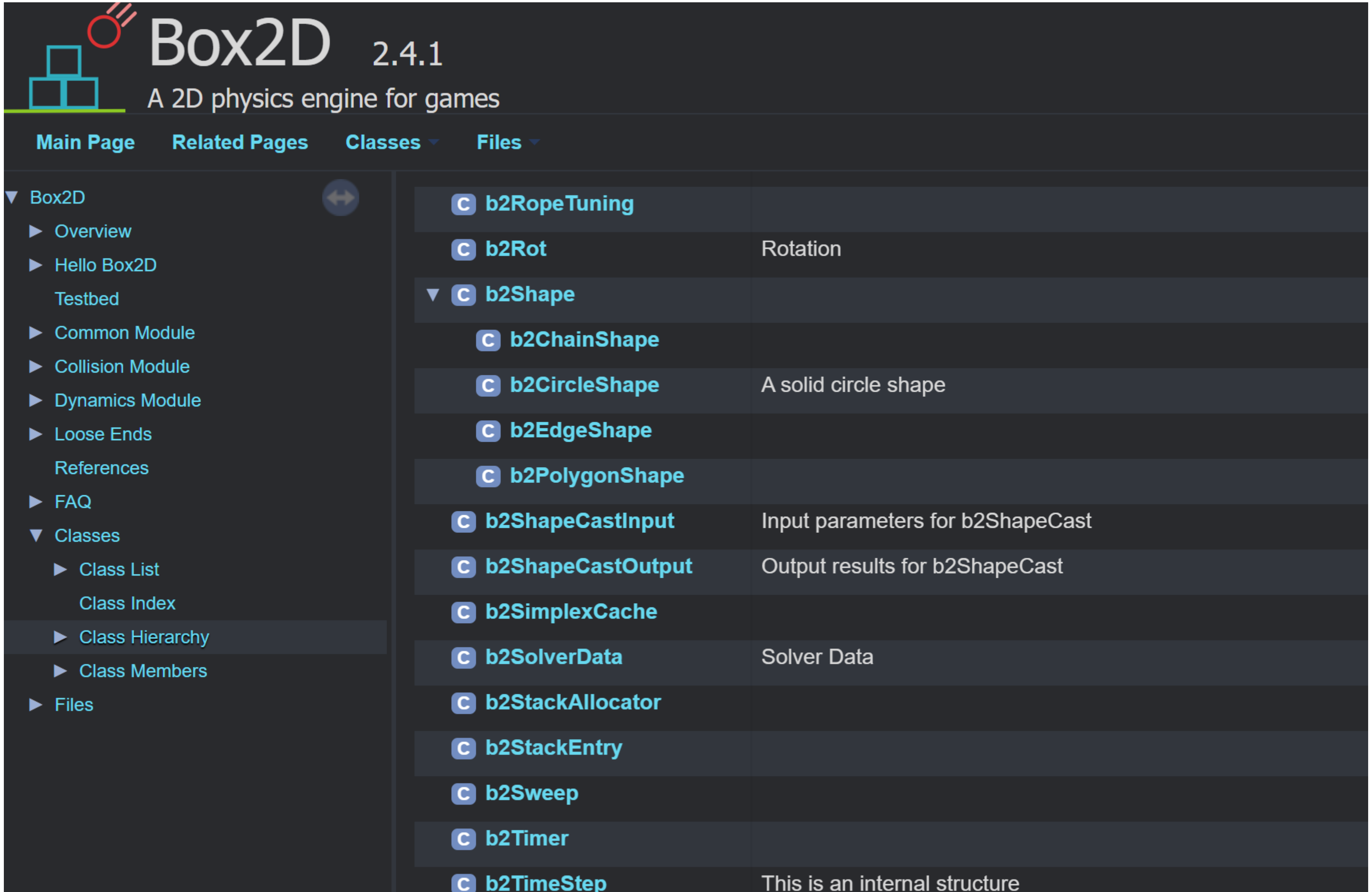
  Correct solution: The file pathname class should contain a string.

resource

Doxygen

- Doxygen is the de facto standard tool for generating documentation from annotated C++ sources

BOX2D class hierarchy (github)

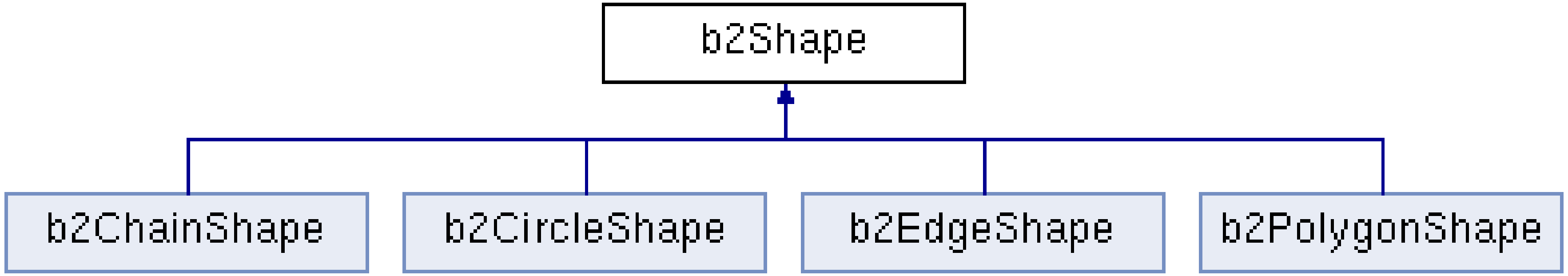


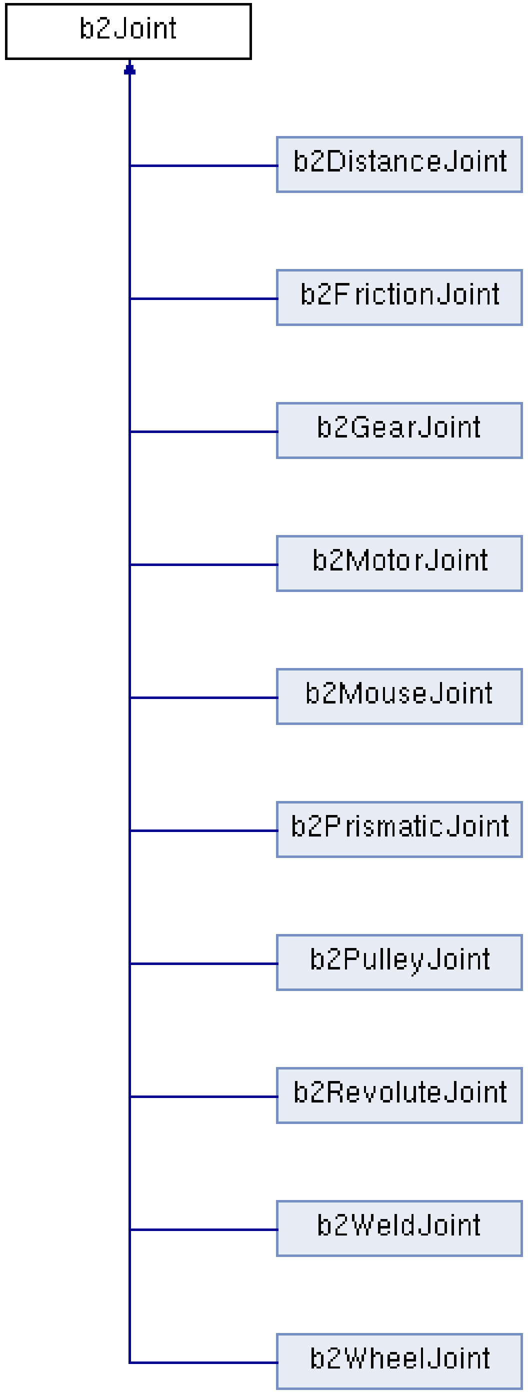
Box2D 2.4.1

A 2D physics engine for games

b2RopeTuning	
b2Rot	Rotation
▼ b2Shape	
b2ChainShape	
b2CircleShape	A solid circle shape
b2EdgeShape	
b2PolygonShape	
b2ShapeCastInput	Input parameters for b2ShapeCast
b2ShapeCastOutput	Output results for b2ShapeCast
b2SimplexCache	
b2SolverData	Solver Data
b2StackAllocator	
b2StackEntry	
b2Sweep	
b2Timer	
b2TimeStep	This is an internal structure

Box2D


b2Shape
b2ChainShape
b2CircleShape
b2EdgeShape
b2PolygonShape


b2Joint
b2DistanceJoint
b2FrictionJoint
b2GearJoint
b2MotorJoint
b2MouseJoint
b2PrismaticJoint
b2PulleyJoint
b2RevoluteJoint
b2WeldJoint
b2WheelJoint