UML Class diagram: classes, and how they are related to other classes

Prof. Y C Cheng
2009/11/20

國立台北科技大學 資訊工程系
National Taipei University of Technology
Dept of Comp Sci and Info Engr
C++ features to support the object-based paradigm

• Class
  – Member functions (overloaded operators)
  – Constructor and destructor
  – Data members
  – Relationship to other classes

• Modeling principle:
  – public interface and private implementation
  – Objects that *do* things
  – Objects use services of other objects by sending *messages* (calling member functions)

• Object life cycle:
  – Creating (+initialization), serving, recycling
UML: a class

- public
- protected
- private

- Abstract
- Concrete

- data type
- parameter

```
class_name

variable1
variable2
function1()
function2()
```
Example: IntArray

<table>
<thead>
<tr>
<th>IntArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
</tr>
<tr>
<td>-size</td>
</tr>
<tr>
<td>+getInput()</td>
</tr>
<tr>
<td>+printOutput()</td>
</tr>
<tr>
<td>+Sort()</td>
</tr>
<tr>
<td>+cleanUp()</td>
</tr>
<tr>
<td>+getSize()</td>
</tr>
</tbody>
</table>
UML: class relationship

- Association  \[X \rightarrow Y\] (knows a)
- Dependency  \[X \rightarrow Y\] (uses a)
- Composition  \[X \diamondrightarrow Y\] (has a)
- Aggregation  \[X \diamondrightarrow Y\] (has a)
- Inheritance  \[\text{Y is a X}\] (is a)
- Class template  \[X \sqcap \text{Y}\] (parameterized class)
“Uses a” $\iff$ “Knows a” relationship

- “Uses a”
  - Dependency
  - One object issues a function call to a member function of another object
- “Knows a”
  - Association
  - One object is aware of another; it contains a pointer or reference to another object
“Is a” ⇔ “Has a” relationship

- “Is a” relationships
  - Inheritance
  - A class is derived from another class

- “Has a” relationships
  - Composition or Aggregation
  - A class contains other classes as members
Aggregation ↔ Composition

- Both are “Has a” or “part-of” relationship
- Composition
  - a stronger variety of aggregation
  - the part object may belong to only one whole
  - expected to live and die with the whole
    - delete whole → delete part
- Aggregation
  - cascading delete is often
  - an aggregated instance can be shared
How are Matrix and Vector related?

#include “MathVector.h”
Class Matrix {
...
public:
    MathVector * getRowVector(int r);
    MathVector * getColumnVector(int c);
...
};
More common relationships

- A student belongs to a department.
- A department has one or more of students.
- A department owns one or more courses.

Unified Modeling Language
Todo

1. I can create a student given a name.
2. I can create a department given its name.
3. I can add a student to a department.
   3.1 A student knows his/her department.
   3.2 A department knows all of its students.
1. I can create a student given a name
2. I can create a department given its name.

• Write constructors given a string
3. I can add a student to a department.

- Forming relationships from Department to Student: A department *knows all* of its students.
  - Department holds a vector of pointers to Students
3. I can add a student to a department.

- Forming relationships from Student to Department: A student *knows* her/his department.
Course and Course Offering

• In a given semester, a department offers a number of courses.
• A course has a name, description, and number of credit.
• The description and number of credits can change over time.
• A course can be offered by zero or more departments (zero = not offered)
• Each offered course has one or more students enrolled
Design One

• A course can change. For example, Computer Programming used to teach Pascal in the 1980s.
Alternative Design

• hw 7
file i/o and factory method

7. I can create the departments from data in a file.
   create a method to create depts from a file.
   design problem: what to return to hold the refs to the depts created?
   note: devc++ auto-inserts a eol when you press save. Use notepad to prepare input file,
   and make sure that you end not at a new line.
8. I can create students from data in a file.
9. I can get student in a dept by name.
Factory Method

• Methods below are called *factory methods*:
  
  vector&lt;Department *&gt; DepartmentsFromFile(const char * fn);
  
  vector&lt;Student *&gt; StudentsFromFile(const char * fn, vector&lt;Department *&gt; depts);
  
• They are responsible for creating objects in your program
Where are the Departments and Students?

Memory model of executing TEST(DepartmentStudent, StudentFromFile) in University fio
Homework 8

• Using your result of homework 7, please do the following:
  – Create the objects of type Student, Department, CourseOffering, Course from the files student-hw8.txt, department-hw8.txt, course-offering-hw8.txt, course-hw8.txt
  – Make sure that the queries in the hw8test.cpp passes
    • Courses offered by Computer Science in the semester 2009F.
    • Number of Computer Science students in the OOP class; number of Music students in the OOP class
    • Number of Out-of-department students in the Data Structures class
    • Number of credits signed up by a particular student
Inheritance

HourlyEmployee gets all data members and member functions of Employee, but can override the member functions.

class HourlyEmployee : public Employee
OBSort1-Source Code

• Note:
  – Use of a class to tie operations and data together.
  – Use of member functions by object.method().
OBSort1 - Problem

• Sort works for IntArray only (IntArray is doing too much).
• Only one sorting algorithm is supported
OBSort2-Source Code

• Note:
  – Sort() no longer belongs to IntArray.
  – Operator overload operator[].
OBSort2- Problem

• Sort() is a global function (not an object).
• Mixing object-based (IntArray) and procedure-based programming (Sort()).
OBSort3-Source Code

• Note:
  – Objectify sort().
  – The sorter object is stateless.
OBSort4-Source Code

• Note:
  – Add Sorter::exchange() and Sorter::compare() to enhance readability, and to allow for future extension.
OBSort5-Source Code

• Note:
  – Add the ability to count the number of comparisons and exchanges.
  – A sorter is no longer stateless.
OBSort5- Problem

• This implementation is inefficient for sorting that does not need to count comparison and exchange.

• How to support two sorting algorithms simultaneously?
OOSort1-Source Code

• Note:
  – It is best to keep Sorter simple for efficiency.
  – Derive a subclass CountingSorter from Sorter.
OOSort1- Problem

• Code of Sorter not reused.
• If s is a Sorter object (not a CountingSorter object), s.getCompareCount() can not be used.
• How to treat a Sorter object and a CountingSorter object in the same way?
OOSort2-Source Code

- `Sorter::sort()` is reused.
- `Sorter:: sort()` is a virtual functions.
- `Foo()` uses reference parameters.
- Function `foo()` can accept either a `Sorter` object or a `CountingSorter` object.
Polymorphism in OOSort2

• Polymorphism: Function foo() can accept either a Sorter object or a CountingSorter object.

• This requires all three of the following:
  – CountingSorter inherits Sorter
  – Foo() calls only member functions in Sorter declared “virtual” (the sort() member function).
  – Foo() takes Sorter by reference or pointer
Back to University...

• The CS department has undergraduate students as well as graduate students.

• Graduation criteria differ for undergraduate and graduate students.
  – Undergraduate student is eligible for graduation if sufficient credits have been earned.
  – Graduate student eligible for graduation if sufficient credits have been earned \textit{and} if a thesis has been approved.
Examples

• Suppose the 132 credits are required for undergraduate and 32 credits for graduates plus thesis.
  – Undergraduate John has earned 128, not eligible for graduation.
  – Undergraduate Mary has earned 135, eligible for graduation.
  – Graduate Amy has earned 32 credits and with her thesis approved, therefore eligible for graduation.
  – Graduate Joe has earned 32 credits but his thesis is not yet approved, therefore not eligible for graduation.

• The problem is to have the department print out a list of students eligible for graduation. How should we design?
Class

- How should we differentiate graduate students from undergraduate students?

![Class Diagram]
void Department::printGraduationList()
{
    for (each student in department)
        if (student->getCategory() == 0) // undergraduate
            if (student->checkUndergraduate())
                cout << *student;
        else // graduate
            if (student->checkGraduate())
                cout << *student;
}
Is the design good or bad?

- `getCategory()` is just a method to query the “type” of the student
- Need two different-named methods for the same objective
- What happens if you have a new type of student?
  - change `Student::getCategory()`
  - Add a new check function to `Student`
  - Change the code in `Department` to test for the new type of student
2. Polymorphism: Derive GraduateStudent from Student

```cpp
void Department::printGraduationList()
{
    for (each student in department)
        if (student->checkGraduation())
            cout << *student;
}
```

### UML Diagram

```
Student
  +checkGraduation()

GraduateStudent
  +checkGraduation()

Department
```

The diagram shows a relationship between the `Student` and `GraduateStudent` classes, which are derived from a common base class, `Student`. The `checkGraduation()` method is marked as a virtual method, indicating that it can be overridden in derived classes to provide specific implementations.
Is the design good or bad?

- `getCategory()` not needed; “type” of the student is part of the class definition

- The same-named method for the same objective
  - Department does not need to distinguish graduate students and undergraduate students

- What happens if you have a new type of student?
  - Derive a new class from `Student`
  - Override `checkGraduation()`
  - No change otherwise!

```
Student
+checkGraduation()

Department
1..n

Graduate
Student
+checkGraduation()

XYZ
Student
+checkGraduation()
```
Introduction to polymorphism

Problem:
A list of unknown shapes?

```
Shape
  draw()
  ...()

Circle
  draw()
  ...()

Rectangle
  draw()
  ...()

Triangle
  draw()
  ...()
```
Declaring a list of unknown shapes

Shape s[100]; // Bad  
Circle s[100];  // Bad  
Shape s[100];  // Bad  
Shape *s[100];  // OK, 但怎麼呼叫draw()?  

1. 混合type的shape  
   怎麼儲存在array?  
2. 怎麼呼叫draw()?  

Solution:  
設計virtual function  

s[0] = new Circle;  
s[1] = new Triangle;  
...  
for (i=0;i<100;i++)  
   s[i]->draw(); // ???  

Does NOT work
Effect of Virtual Functions

Example (non-virtual)

```cpp
class X {
public:
    void f() {
        cout << 'x';
    }
};
class Y : public X {
public:
    void f() {cout<<'y';}
};
...
X a; a.f(); // x
Y b; b.f(); // y
X *p=&a;
p->f();      // x
X *p=&b;
p->f();      // x
```

Example (virtual)

```cpp
class X {
public:
    virtual void f() {
        cout << 'x';
    }
};
class Y : public X {
public:
    void f() {cout<<'y';}
};
...
X a; a.f(); // x
Y b; b.f(); // y
X *p=&a;
p->f();      // x
X *p=&b;
p->f();      // y
```

virtual只對pointer或reference有效(dynamic binding)
Virtual Functions for the shape problem

Shape *s;
s = new Circle;  s->draw(); // 呼叫circle的draw()
s = new Triangle;  s->draw(); // 呼叫Triangle的draw()
...
Shape *s[100];
... // 設定s[0..99]分別指向哪一種物件
for (int i=0;i<100;i++)
    s[i]->draw(); // 依據實際的type呼叫draw()
Virtual Functions & Polymorphism

• Polymorphism
  – Ability for objects of different classes to respond differently to the same function call
  – Implemented through virtual functions
    • C++ 允许使用 base class pointer (或 reference) 呼叫 derived class 的 member function
    • C++ chooses the correct overridden function in object

• Dynamic binding
  – Function determined during execution time
    ShapePtr->Draw();

• Static binding
  – Function determined during compile-time
    ShapeObject.Draw();
Dynamic Binding ↔ Static Binding

- **Dynamic binding (run-time)**
  - 用 base class 的 pointer 或 reference 呼叫 virtual function

- **Static binding (compile time)**
  - 其他 (用 object 呼叫 virtual/non-virtual function, 用 pointer 或 reference 呼叫 non-virtual function)

```cpp
class X {
public:
    void f1() {
        cout << "xf1";
    }
    virtual void f2() {
        cout << "xf2";
    }
};
class Y : public X {
public:
    void f1() {cout << "yf1";}
    void f2() {cout << "yf2";}
};
X a;
a.f1(); // xf1 static
a.f2(); // xf2 static
Y b;
b.f1(); // yf1 static
b.f2(); // yf2 static
X *p=&a;
p->f1(); // xf1 static
p->f2(); // xf2 dynamic
X *pb=&b;
pb->f1(); // xf1 static
pb->f2(); // yf2 dynamic
```
Abstract and Concrete Classes

• Abstract classes
  – To provide a base class for other classes
  – No objects can be instantiated
  – Declare one or more pure virtual functions
  – Example
    
    ```
    virtual void draw() = 0;
    ```

• Concrete classes
  – Objects can be instantiated
  – No unimplemented virtual functions
    • Implement all unimplemented virtual functions
Abstract and Concrete Classes

C++ example

class Shape {
public:
    virtual void draw() = 0;
    void setColor(int c) {
        color = c;
    }
private:
    int color;
};
class Rectangle : public Shape {
public:
    void draw() {...};
    void getArea() {...};
    virtual void f() {...};
};
Abstract and Concrete Classes

UML example

- Abstract class: `Shape`
  - `draw()`
  - `set_color()`

- Concrete classes:
  - `Circle`
    - `draw()`
    - `...()`
  - `Rectangle`
    - `draw()`
    - `...()`
  - `Triangle`
    - `draw()`
    - `...()`
Virtual Destructors

- Problem:
  - How to delete an object with a base class pointer?
- Solution:
  - declare a **virtual** (base-class) destructor

```cpp
class X {
public:
    virtual ~X(); // virtual destructor
    ...
private:
    ...
};
class Y : public X {
    ...
};

X *pa = new Y;
...
// use *pa
...
delete pa;
```
How does compiler handle virtual functions?

- **virtual** functions
  - overheads?
    - memory
    - speed
  - How it works?
    - dynamic binding
    - vtable
虛擬函式表 (vtable)

每個 class 都有一個 vtable，每個 object 都帶著 vtable 的 pointer。例如，當我們呼叫 `Circle` 對象的 `print()` 函式時，會根據 vtable 中的指針來執行正確的函式實作。