

Pattern-Oriented Software Design



Introduction to UML Class Diagrams

CSIE Department, NTUT

Woei-Kae Chen

8.14.1

UML: Unified Modeling Language



- Successor to OOA&D methods
 - late 1980s and early 1990s
- Unifies
 - Jacobson & OMT (Booch & Rumbaugh)
- Graphical notation used to express designs
 - Use cases
 - **Class diagrams**
 - Interaction diagrams
 - **Sequence diagrams**
 - Collaboration diagrams
 - Package diagrams
 - State diagrams
 - Activity diagrams
 - Deployment diagrams

GoF Book

8.14.1

UML class diagrams



- Three perspectives
 - **Conceptual**
 - represents of the domain under study
 - relate to the class that implement them, but often no direct mapping
 - **Specification**
 - looking at types rather than classes
 - a type represents an interface that may have different implementations
 - **Implementation**
 - looking at classes

for our POSD class

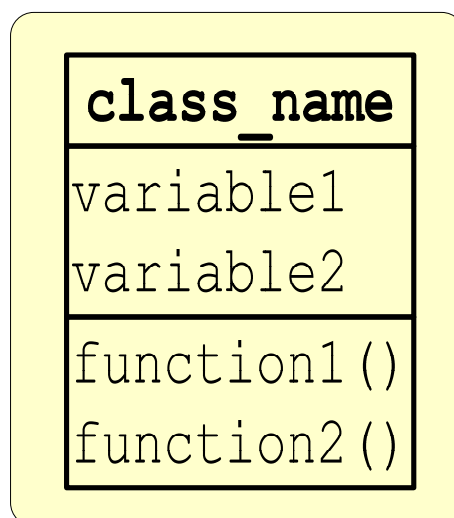
UML: a class



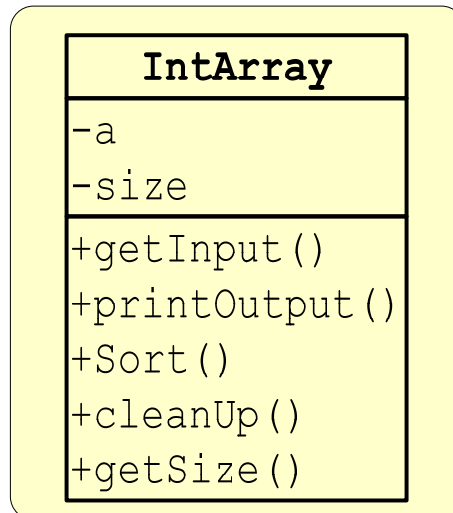
+ public
protected
- private

Abstract
Concrete

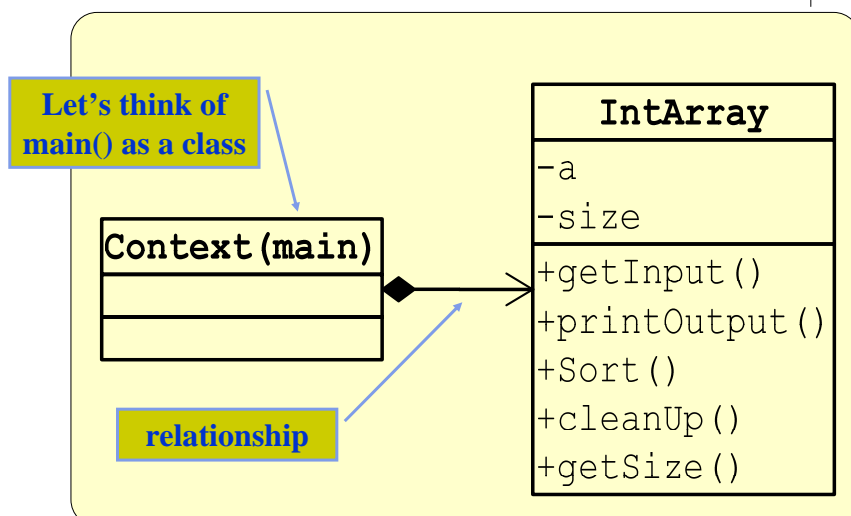
•data type
•parameter



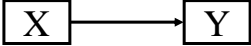
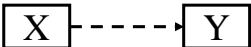
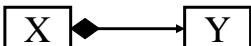
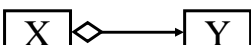
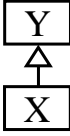
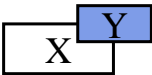
Example: OSort1.cpp



Example: OSort1.cpp

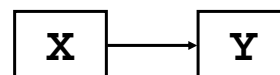


UML: class relationship

- Association  (knows a)
- Dependency  (uses a)
- Composition  (has a)
- Aggregation  (has a)
- Inheritance  (is a)
- Class template  (parameterized class)

“Uses a” ⇔ “Knows a” relationship

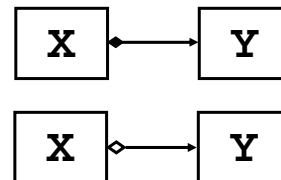
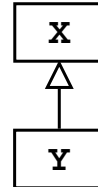
- “Uses a”
 - Dependency
 - One object issues a function call to a member function of another object
- “Knows a”
 - Association
 - One object is aware of another; it contains a pointer or reference to another object



“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



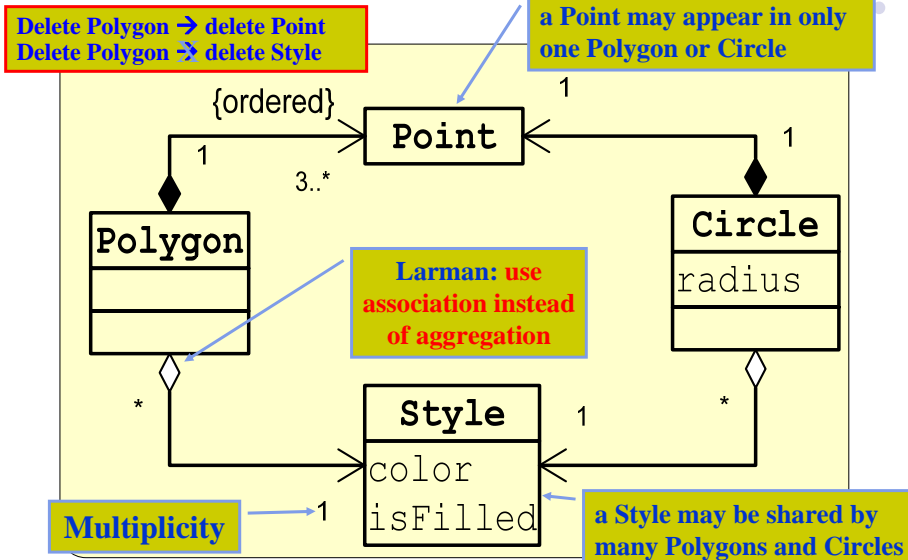
Aggregation ⇔ Composition



- 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**

Following Larman OOAD:
use of aggregation is NOT
recommended

Example: “has a” relationship



Relationship Examples

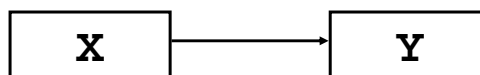
- | | |
|-----------------------------|---------------------------------------|
| • Car ⇔ Engine ? | → Composition |
| • Person ⇔ Cell Phone ? | → Association/Composition/Dependency |
| • Human ⇔ Brain ? | → Composition |
| • Fighter ⇔ Bomb ? | → Association |
| • Fighter ⇔ F16 | → Inheritance |
| • Bomb ⇔ Explosive | → Composition/Association/Inheritance |
| • Bomb ⇔ Nuclear Bomb | → Inheritance |
| • MyComplex ⇔ Math ? | → Dependency |
| • Tree Node ⇔ Child Node ? | → Composition |
| • Tree Node ⇔ Parent Node ? | → Association (if needed) |
| • Hero ⇔ Life ? | → Composition/Attribute |
| • Hero ⇔ Score ? | → Association/Dependency |
| • Hero ⇔ Map ? | → Association/Dependency |

Relationship Examples



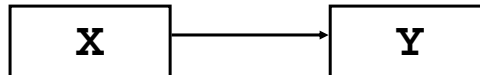
- Flight 123 ⇔ Airplane ? → Association/Dependency
- Flight 123 ⇔ Airport ? → Association
- Flight 123 ⇔ Passenger ? → Association/Dependency
- Flight 123 ⇔ Flight Captain ? → Association/Dependency
- Flight 123 ⇔ Flight Attendant ? → Association/Dependency
- Airplane ⇔ Boeing 747 ? → Inheritance
- Airplane ⇔ Seat ? → Composition
- Airplane ⇔ Fuel ? → Composition/Attribute
- Passenger ⇔ Flight ? → Association/Dependency
- Passenger ⇔ Ticket ? → Association/Dependency
- Passenger ⇔ Travel Agent ? → Association/Dependency
- Ticket ⇔ Price ? → Composition/Attribute

UML Example (C++): Association



```
class X {
    X(Y *y) : y_ptr(y) {}
    void SetY(Y *y) {y_ptr = y;}
    void f() {y_ptr->Foo();}
    ...
    Y *y_ptr; // pointer
};
```

UML Example (C++): Association



How is an association created?

Example #1

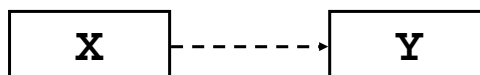
```
...  
Y an_y();  
X an_x(&an_y);  
an_x.f();  
...  
...
```

Example #2

```
...  
Y an_y();  
X an_x();  
...  
an_x.SetY(&y);  
...  
an_x.f();
```

E.14

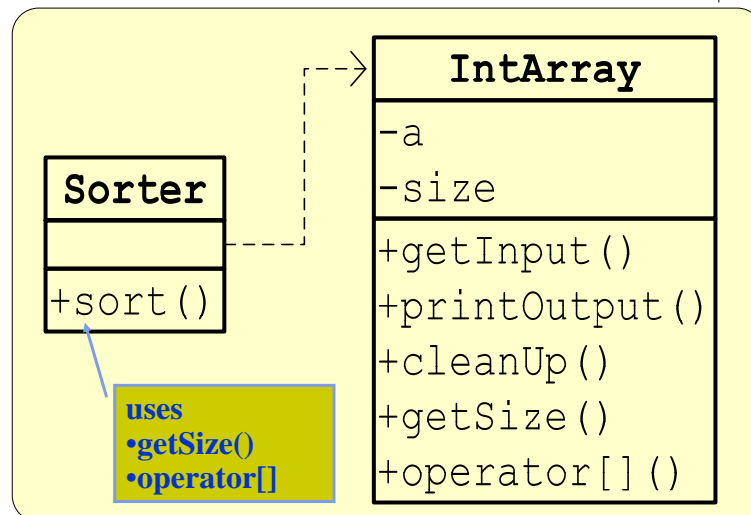
UML Example (C++): Dependency



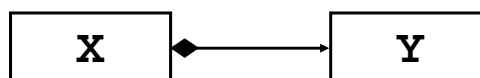
```
class X {  
...  
void f1(Y y) {...; y.Foo();}  
void f2(Y *y) {...; y->Foo();}  
void f3(Y &y) {...; y.Foo();}  
void f4() {Y y; y.Foo();...}  
void f5() {...; Y::StaticFoo();}  
};
```

E.15

Example: OSort3.cpp



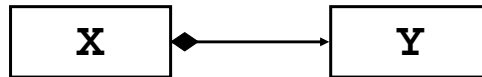
UML Example (C++): Composition 1



```
class X {
    ...
    Y a;           // 1; Composition
    Y b[10];       // 0..10; Composition
};
```

Java?

UML Example (C++): Composition 2



```

class X {
    X() { a = new Y[10]; }
    ~X(){ delete [] a; }
    ...
    Y *a;           // 0..10; Composition
};
    
```

NOT Association

UML Example (C++): Composition 3



Implementation detail



```

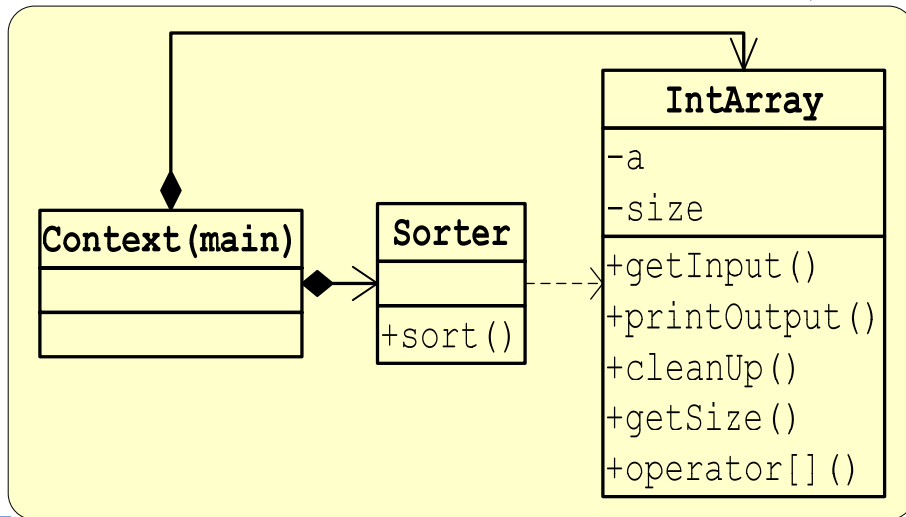
class X {
    ...
    vector<Y> a;
};
    
```

Hiding implementation detail

Composition of vector<Y>

NOT Composition of Y

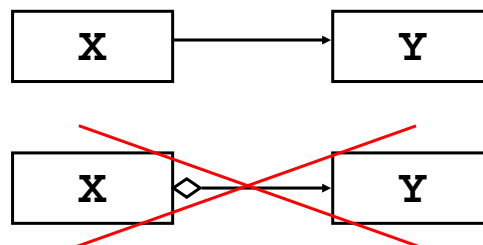
UML Example: OSort3.cpp



UML Example (C++): Aggregation



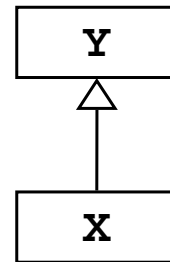
- No example here
 - Use Association instead of Aggregation



UML Example (C++): Inheritance

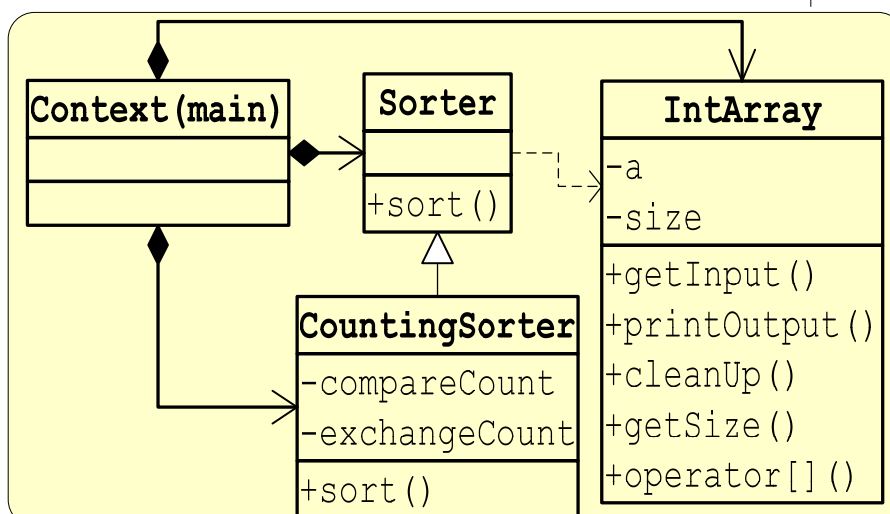


```
class Y {  
    ...  
};  
  
class X : public Y {  
    ...  
};
```

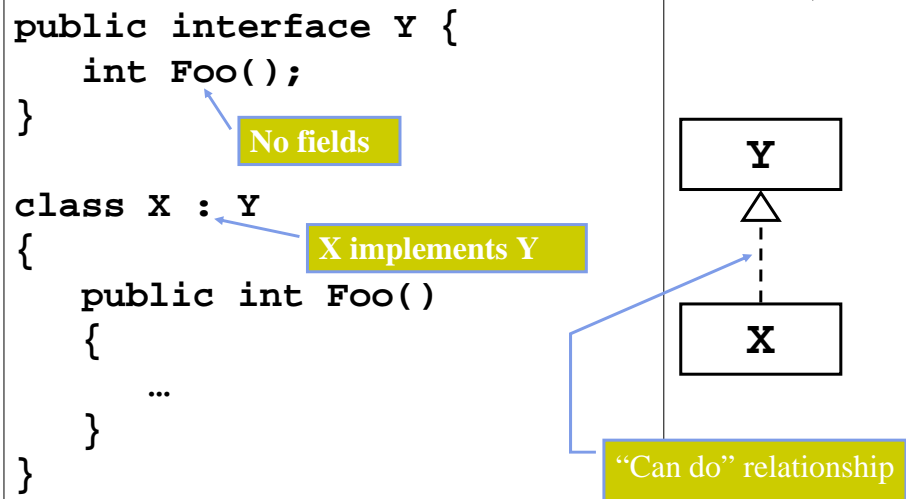


“is a” relationship

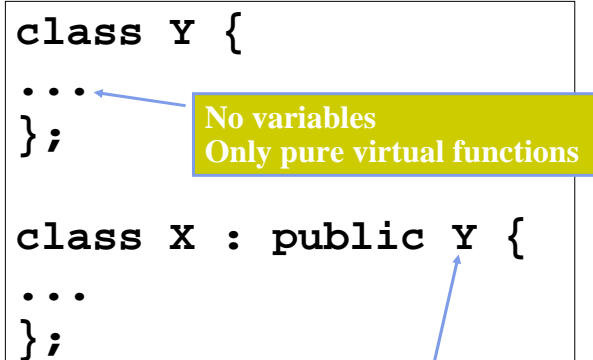
Example: OOSort2.cpp



UML Example (C#): Implementation



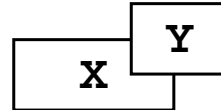
UML Example (C++): Implementation



UML Example (C++): Template Class



```
template <class T>
class X {
    ...
    ...
    ...
};
```



```
...
X<Y> a;
...
```

