Pattern-Oriented Software Design

Introduction to UML Class Diagrams

CSIE Department, NTUT

Woei-Kae Chen

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
UML class diagrams

- Three perspectives
  - Conceptual
    - represents 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

UML: a class

```
class_name

variable1
variable2
function1()
function2()
```
Let's think of `main()` as a class

Example: OBSort1.cpp

```
IntArray
-a
-size
+getInput()
+printOutput()
+Sort()
+cleanUp()
+getSize()
```

Example: OBSort1.cpp

```
Context(main)

Let's think of main() as a class

IntArray
-a
-size
+getInput()
+printOutput()
+Sort()
+cleanUp()
+getSize()
```
**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

Delete Polygon → delete Point
Delete Polygon → delete Style

Point

Polygon

Circle

Style

Larman: use association instead of aggregation

a Style may be shared by many Polygons and Circles

Relationship Examples

• Car ⇔ Engine ?
• Person ⇔ Cell Phone ?
• Human ⇔ Brain ?
• Fighter ⇔ Bomb ?
• Fighter ⇔ F16
• Bomb ⇔ Explosive
• Bomb ⇔ Nuclear Bomb
• MyComplex ⇔ Math ?
• Tree Node ⇔ Child Node ?
• Tree Node ⇔ Parent Node ?
• Hero ⇔ Life ?
• Hero ⇔ Score ?
• Hero ⇔ Map ?

→ Composition
→ Association/Composition/Dependency
→ Composition
→ Association
→ Inheritance
→ Composition/Association/Inheritance
→ Inheritance
→ Dependency
→ Composition
→ Association (if needed)
→ Composition/Attribute
→ Association/Dependency
→ 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

```cpp
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?

<table>
<thead>
<tr>
<th>Example #1</th>
<th>Example #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>Y an_y();</td>
<td>Y an_y();</td>
</tr>
<tr>
<td>X an_x(&amp;an_y);</td>
<td>X an_x();</td>
</tr>
<tr>
<td>an_x.f();</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>an_x.SetY(&amp;y);</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td>an_x.f();</td>
</tr>
</tbody>
</table>

**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();}
};
Example: OBSort3.cpp

```cpp
// Example: OBSort3.cpp

Example:
- `getSize()`
- `operator[]`

UML Example (C++): Composition 1

```cpp
class X {
    ...
    Y a; // 1; Composition
    Y b[10]; // 0..10; Composition
};
```
UML Example (C++): Composition 2

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

UML Example (C++): Composition 3

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

Implementation detail

Hiding implementation detail

Composite of vector<Y>

NOT Composition of Y
UML Example: OBSort3.cpp

- Context (main)
- Sorter
- + sort()
- IntArray
  - a
  - size
  - + getInput()
  - + printOutput()
  - + cleanUp()
  - + getSize()
  - + operator[]()

UML Example (C++): Aggregation

- No example here
- Use Association instead of Aggregation
UML Example (C++): Inheritance

```c++
class Y {
    ...
};

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

“is a” relationship

Example: OOSort2.cpp
UML Example (C#): Implementation

```csharp
public interface Y {
    int Foo();
}

class X : Y {
    public int Foo()
    {
        ...
    }
}
```

- No fields
- X implements Y
- "Can do" relationship

UML Example (C++): Implementation

```c++
class Y {
    ...
};

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

- No variables
- Only pure virtual functions
- C++ allows multiple inheritance
**UML Example (C++): Template Class**

```
template <class T>
class X {
    ...
    ...
    ...
};
X<Y> a;
...
C++ static member

Order

getNumber
getNextNewNumber