UML class diagrams

Giuseppe Lipari
http://retis.sssup.it

Scuola Superiore Sant'Anna – Pisa

June 8, 2009
Goal: Be able to “reason about” a design
   i.e., understand designer’s intent
   Critique/improve the design

Claim: Source code not best medium for communication and comprehension
   Lots of redundancy and detail irrelevant for some program-understanding tasks
   Especially poor at depicting relationships among classes in OO programs
   To understand an OO design, one must be able to visualize these relationships

Solution: Use abstract, visual representations - UML
UML diagrams

- Collection of notations representing software designs from three points of view:
  - *Class model* describes the static structure of objects and relationships in a system
  - *State model* describes the dynamics aspects of objects and the nature of control in a system
  - *Interaction model* describes how objects in a system cooperate to achieve broader results

- Generally, we need all three models to describe a system
- No single model says everything
- Here we focus on class model
1. UML Class diagram notation
2. Objects
3. Relationships
4. Template notation
Boxes denote classes
Each box comprises:
- Class name
- List of data attributes
- List of operations
More compact than code and more amenable to depicting relationship among classes
Abstraction in class diagrams

- Class diagrams often elide details
  - Method associated with an operation
  - Attribute and operations may be hidden in diagrams to improve readability
  - even if they exist in C++ code

<table>
<thead>
<tr>
<th>ClassName</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr1: type1 = def1</td>
<td>firstName: string</td>
</tr>
<tr>
<td>attr2: type2 = def2</td>
<td>lastName: string</td>
</tr>
<tr>
<td>opName1(arg1: argtype1): restype1</td>
<td>hireDate: Date</td>
</tr>
<tr>
<td>opName2(arg2: argtype2): restype2</td>
<td>department: short</td>
</tr>
</tbody>
</table>
Inheritance

- **DerivedClass** is derived from **BaseClass**.
- **BaseClass** class has a virtual method (in italic).
- **DerivedClass** reimplemented the virtual method.
Outline

1. UML Class diagram notation
2. Objects
3. Relationships
4. Template notation
Object notation

- **Object: Classname**
  - attr1=value1
  - attr2=value2

**Notes:**
- The UML symbol for an object is a box with an object name followed by a colon and the class name. The object name and class name are both underlined.
- Attribute values and the object name are optional.
- Only list attributes that have intrinsic meaning. Attributes of computer artifacts (such as pointers) should not be listed.
Example

```java
Employee doe("John", "Doe", ...);
Employee* doe =
    new Employee("John", "Doe", ...);

eList.addEmpl(
    new Employee("Mary", "Smith", ...)
);
```
A More formal distinction

- **Value**: Primitive “piece of data”
  - E.g., the number 17, the string “Canada”
  - Unlike objects, values lack identity

- **Object**: Meaningful concept or “thing” in an application domain
  - Often appears as a proper noun or specific reference in discussions with users.
  - May be attributed with values
  - Has identity

- Two objects containing the “same values” are not the same object!
  - They are distinct objects
  - They may be considered “equivalent” under a certain definition of “equality”
What’s the big deal about identity?

- Useful in reasoning about “goodness” of a design
  - Many poor designs result from an “encoding” of one object within another, using attribute values
  - By reasoning about identity, one may identify such a design flaw early
  - Best illustrated by example

- Also allows us to model relationships among objects and classes more explicitly
Exercise: Travel-planning system

- A city has a name, a certain population, and a specific time zone
- A city has one or more airports
- An airport has a name and a unique code
Exercise: Travel-planning system

- A city has a name, a certain population, and a specific time zone
- A city has one or more airports
- An airport has a name and a unique code
- How many classes should you design?
Is this design correct?

- These attributes are “hiding” an object (the airport) that is meaningful by itself in this domain
- Why it might be bad to encode one object as a collection of attribute values within another?
Design tip

Answer:
- Potential for redundancy/inconsistency due to duplication
  - some airports serve multiple cities
  - some cities served by no airports
  - some cities served by multiple airports
- Operations over Airport objects may not need to know details associated with cities, such as population

When designing a class:
- Apply the identity test to each attribute (including attributes in combination)
- Never use an attribute to model an “object identifier”

UML notation helps enforce this discipline

So then how do we model connections between objects, such as Cities and Airports?
Outline

1. UML Class diagram notation
2. Objects
3. Relationships
4. Template notation
Relationships among objects

- **Link**: Physical or conceptual connection between objects
  - Much more abstract than pointers/references
  - Most (not all) links relate exactly two objects

- **Association**: Description of a group of links with common structure and semantics

A link is an instance of an association:
- Links connect objects of same classes
- Have similar properties (link attributes)
- Association describes set of potential links just like a class describes a set of potential objects
Examples of links

**Houston: City**
- cityName = “Houston, TX"
- population = 3,000,000

**HOU: Airport**
- airportCode = HOU
- airportName = “Hobby"
- timeZone = Central

**IAH: Airport**
- airportCode = IAH
- airportName = “Intercontinental"
- timeZone = Central
From links to association

City
- cityName: string
- population: unsigned

Airport
- airportCode: code
- airportName: string
- timeZone: zone

1..* serves *
Bidirectionality

- Links may be navigated in either direction!
- Benefits:
  - During early design, it is often difficult to predict the navigation directions that will be needed
    - Especially true for many-to-many associations
    - Better to model connections as bidirectional associations and later refine these associations into more implementation-level structures (e.g., pointers, vectors of pointers maps etc)
  - Often several ways to implement an association and the details are not salient to the “essence” of the design
Implementation of “serves” association

class City {
    ...
protected:
    string cityName;
    unsigned population;
    vector<Airport*> serves;
};

class Airport {
    ...
protected:
    string airportName;
    CODE airportCode;
    ZONE timeZone;
    vector<City*> serves;
};
Implementation of “serves” association

```cpp
class City {
    ...
    protected:
        string cityName;
        unsigned population;
        vector<Airport*> serves;
};

class Airport {
    ...
    protected:
        string airportName;
        CODE airportCode;
        ZONE timeZone;
        vector<City*> serves;
};
```

```cpp
class City {
    ...
    protected:
        string cityName;
        unsigned population;
};

class Airport {
    ...
    protected:
        string airportName;
        CODE airportCode;
        ZONE timeZone;
};
```

```
multimap<City*, Airport*> cityServes;
multimap<Airport*, City*> airportServes;
```
You should get comfortable with the various methods for refining a UML association

- be able to easily switch back and forth between what is said in the diagram and what is allowable in the code
- start to “think” using links/associations rather than pointers and references

This is good training in abstraction
Outline

1. UML Class diagram notation
2. Objects
3. Relationships
4. Template notation
**Template notation**

Equivalent to:

```cpp
template<class T>
class MyClass {
    T var;
    int number;
public:
    ...
    T operator[](int index);
};
```