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

UML Class diagram 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
### Notation

#### Visibility:
- `–` is private
- `#` is protected
- `+` is public

#### Specification
- member type follows definition
- parameter type follow name
- optionally, can specify parameter direction (in/out)

---

#### 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
Two classes can be related by:

- Inheritance
- Association
- Aggregation
- Composition

**Inheritance**

- DerivedClass is derived from BaseClass
- BaseClass class has an abstract method (in italic)
  - operation() is a pure virtual method
- DerivedClass implements the virtual method
Object notation

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

We can also remove the member values, and even the object name
A Link is represented as a line connecting two or more object boxes.

It can be shown on an object diagram or class diagram.

A link is an instance of an association.

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
- 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?
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

- **hou:Airport**
  - code = HOU
  - name = Hobby
  - timeZone = Central

- **iah:Airport**
  - code = IAH
  - name = Intercontinental
  - timeZone = Central

- **houst:City**
  - name = Houston
  - population = 3000000

  ![Diagram showing relationships between houst:City, hou:Airport, and iah:Airport](image-url)
Association

- Association represents the ability of one instance to send a message to another instance.
- This is typically implemented with a pointer or reference instance variable, although it might also be implemented as a method argument, or the creation of a local variable.

```cpp
class Researcher {
    vector<Paper *> paper_list;
    ...
};
class Paper {
    vector<Researcher *> author_list;
    ...
};
```
Directional association

- Association can be directed if the link only goes in one direction

![Diagram of an association from Abc to Cde](image)

```cpp
class Abc {
    private:
        Cde *c;
    public:
        ...
};
```

Bidirectionality

- 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;
    \texttt{unsigned} population;
    vector\langle Airport* \rangle serves;
};

class Airport {
    ...
protected:
    string airportName;
    CODE airportCode;
    ZONE timeZone;
    vector\langle City* \rangle serves;
};

Alternative implementation of “serves” association

class City {
    ...
protected:
    string cityName;
    \texttt{unsigned} population;
};

class Airport {
    ...
protected:
    string airportName;
    CODE airportCode;
    ZONE timeZone;
};
multimap\langle City*, Airport* \rangle cityServes;
multimap\langle Airport*, City* \rangle airportServes;
Aggregation

- Association is a general relationship
- When we have a “whole/part” relationship, we can use a special kind of association, called “aggregation”

Instances cannot have cyclic aggregation relationships (i.e. a part cannot contain its whole)

Composition

- Composition is the same as aggregation, but in addition the “whole” controls the “part”

```cpp
class Car {
public:
    virtual ~Car() {delete itsCarb;}
private:
    Carburetor* itsCarb
};
```
Equivalent to:

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