Class Relationships

Chapter 9

Defining The Object Model

Three "Crown Jewels"

- Encapsulation
  - An object combines data and functions (attributes and behaviors)
  - An object hides its implementation (i.e., its data and its algorithms)
  - Access to the objects interior is controlled (public, private, protected)
  - Users utilize the object through its public interface (public functions)

- Inheritance
  - A relationship between classes
  - The derived class inherits (implicitly defines) all of the features (attributes and behaviors/operations) of the base class
  - The parent is a generalization; the child is a specialization

- Polymorphism
  - The correct function is determined at run time

Chapter 11

Class Relationships

Representing systems as collections of related classes

- Are depicted as diagrams
- Are represented as connected graphs
  - Nodes or vertices are classes (rectangles)
  - Edges, arcs, or paths denote the relationship (connecting lines)
- Allow objects to cooperate in the solution of a problem
  - Objects communicate by sending messages along the connections
  - Objects can call functions in connected objects
- Are supported by specific computer-language syntax

Inheritance

The preeminent relationship

- Relationship between a class and a more refined version
- Abstraction for sharing similarities among classes while preserving their differences
- Mechanism for code reuse
  - Conceptual simplification by reducing the number of unique features
- Each subclass inherits all features of the superclass
  - An instance of a subclass is simultaneously an instance of all its ancestor classes (i.e., it contains subclasses)
  - Overriding—subclass defines operation with the same name and the same signature
  - The second defining feature of the object model
    - Object-Based systems support objects but not inheritance

Inheritance Example

An inheritance hierarchy

- Base Class
  - Person
  - Parent Class
  - Generalization
  - Ancestor
  - Derived Class
  - Employee
  - Child Class
  - Specialization
  - Descendant
  - Contains all members of the base class, which is called a "subobject."

- General class
  - Shape
  - Ellipse
  - Polygon
  - Circle
  - Rectangle
  - Square
  - Rhombus
  - Trapezoid

- Specialized classes
### Inheritance Syntax

**Definitions and usage**

- Declare a new class that is a descendant of an existing class:
  ```cpp
class Child : public Parent {};
```
- Define the access control of the inherited members
  - `public` (most common)
  - `private` (default, but little used)
  - `protected`
- Pass parameters from the derived class constructor to the base class constructor:
  ```cpp
  Child::Child(int a) : Parent(a) {};
  ```
- Use public base members as local members

### Inheritance Example

**Determining which method is called**

```cpp
class shape
{
  private:
    int color;
  public:
    shape(int c) : color(c) {}
    void draw() {}
    void setColor(int C) { color = C; }
};

class circle : public shape
{
  private:
    int radius;
  public:
    circle(int r, int c) : shape(c), radius(r) {}
    void draw() { shape::draw(); }
};
```

```cpp
shape S(blue);
circle C(25, yellow);
S.draw();  // runs Shape draw
C.draw();  // runs Circle draw
C.setColor(red);  // runs Shape setColor
```

### Composition

**Containment or “contains a” also “has a”**

- Form of association
  - strong ownership / binding
    - parts & whole have coincident lifetimes
    - instantiate parts in whole constructor
    - parts belong to one whole
    - parts do not change during execution
  - implemented with object attributes
- Example:
  ```cpp
class Automobile
  {
    private:
      Transmission driveTrain;
      Motor engine;
  };
```

### Composition Example

**Making complex from simple**

```cpp
Automobile
    whole
    part
    Motor
    Transmission
```

### Aggregation

**Whole-part, “has a,” assembly, or “a part of”**

- Form of association
  - weak ownership / binding
    - parts & whole have independent lifetimes
    - instantiate parts when convenient
    - build relationship when convenient
    - parts may belong to multiple wholes
    - parts may change during execution
    - implemented w/ object pointer attributes

```cpp
class Engine
{
  private:
    int size;
  public:
    Engine(int s) : size(s) {};
};

class Automobile
{
  private:
    Engine motor;
    int doors;
  public:
    Automobile(int s, int d) : motor(s), doors(d) {};
};
```
**Association**

A subordinate relationship

- Associations are connections between peer objects, which allows objects to call each others functions
- Implemented as object references (i.e., class-scope variables that reference to other objects)
- Associated objects may not be permanent nor must they be created at the same time (i.e., they have separate existences)
- Associations are bidirectional; the association name indicates a forward direction and an inverse direction
  - are sometimes shown with a single, non-directional line
  - are sometimes shown as one or two directed lines (arrows)

**Association Example**

UML, C++, and Java™

```c++
class Contractor {
private:
    string name;
    Project* worksOn;
};

class Project {
private:
    string name;
    Contractor* personnel;
};
```

**Dependency**

One object uses another

- A relationship where one object requires a feature (attribute or operation) from another object
- Is implemented through local-scope object reference variables
  - instantiating a local object
  - passing an object as a parameter to a function
- Is not implemented by extracting the needed feature
  - “Passing the entire gorilla, just to get at its liver, is messy, cumbersome, and annoys the hell out of the gorilla”
  - Nonetheless, “passing the entire gorilla” preserves the used object’s integrity and maintains the spirit of the object model
- Ephemeral form of association
- Also known as *delegation or using*

**Dependency**

aka using and delegation (2 ways to implement)

```c++
class FooBar {
public:
    int contains(string name) {
        return input.length();
    }
};
```

**Overriding Inherited Functions**

A bar “is like a” foo— a simile

```c++
class foo {
public:
    void func1();
    void func2();
};

class bar : public foo {
public:
    void func1(void) { foo::func1(); }
};

bar B;
B.func1(); // bar::func1 “overrides” foo
B.func2(); // foo::func2
```

**What Is Inherited?**

Most but not all

- Inherited
  - Data members
  - Most member functions
- Functions not inherited
  - Constructors
  - Destructors
  - Friend functions
  - Assignment (=) operator
- All constructors and destructors in the hierarchy are executed
  - Constructors from the least to the most derived
  - Destructors from the most to the least derived