Class Relationships
Hours 4 & 5; and Chapter 5

Goals and Objectives
Hours 4 & 5; and Chapter 5
- Relationships
- Inheritance
- Implementing
- Casting
- Polymorphism
- Using
- Understanding
- Programming
- Implementing
- Association
- Multiplicity
- Roles & qualifiers
- Special cases
- Association classes
- Aggregation
- Composition
- Dependency
- Interfaces
- Contexts

Generalization-Specialization
Inheritance: is-a, is-a-kind-of; is-like-a, simile

- Subclasses inherit all features of the super class
- Implemented by keyword
- Relationship between a class and a more refined version
- Abstraction for sharing similarities among classes while preserving their differences
- Common features are placed in the super class
- Unique features are placed in the subclass
- Mechanism for code reuse
- Simplification by reducing the number of unique features
- An instance of a subclass is simultaneously an instance of all its ancestor classes (i.e., it contains sub-objects)

Programming Inheritance
Denoted by language syntax
- C++
  - class Employee :: public Person
  - class Employee :: private Person
  - class Employee :: protected Person
- Java
  - public class Employee extends Person
  - The only relationship implemented by special syntax (i.e., by keyword)
  - Suggests its importance in the object model
  - Second feature defining the object-oriented model

Single vs Multiple Inheritance
Avoiding the "deadly diamond"
- Multiple inheritance permits a subclass (D) to inherit features from multiple super classes (B & C)
- Object-orientation requires inheritance
  - Only single inheritance is required, not multiple inheritance
  - Java supports single; C++ supports multiple
- Multiple inheritance becomes problematic if the super classes (B & C) have a common ancestor (A)
  - This results in the "deadly diamond" inheritance hierarchy
  - C++ introduced "virtual inheritance" to deal with this situation
### Implementing Inheritance

Implemented through sub-objects

- Instantiating a `Square` implies simultaneously instantiating a `Rectangle`, a `Polygon`, and a `Shape`
- Constructors executed from top to bottom
- Form sub-objects, which are embedded within subclass objects

```java
new Square();
```

### Calling Super Class Constructors

**Initializing sub-objects**

```cpp
C++
class Person { 
private: 
char* name;
public: 
Person(char* n) { name = n; } 
};
class Employee : public Person { 
public: 
Employee(char* n) : Person(n) { 
super(n); 
}
};
Java
class Person { 
private String name;
public Person(String n) { 
name = n; 
}
};
class Employee extends Person { 
public Employee(String n) { 
super(n); 
}
};
```

### Overriding and Overloading

Methods with the same name

- **Overloading**
  - One class
  - Two or more methods with the same name
  - Methods must have different argument lists
  - Return types may be different

- **Overriding**
  - Two or more classes
  - Classes directly related by inheritance
  - Two or methods with the same name but in different classes
  - Methods must have the same argument list
  - Return types must be the same

### Accessing Overridden Features

**C++**

```cpp
class Foo { 
protected: 
char* name;
public: 
Foo(char* n) { name = n; }
char* getName() { 
return name;
}
};
class Bar : public Foo { 
private: 
char* name;
public: 
Bar(char* f, char* l) : Foo(l) { 
name = f;
}
public String getName() { 
return name + super().getName();
}
};
```

**Java**

```java
class Foo { 
protected String name;
public Foo(String n) { 
name = n;
}
String getName() { 
return name;
}
};
class Bar extends Foo { 
private String name;
public Bar(String f, String l) { 
super(f);
name = f;
}
public String getName() { 
return name + super.getName();
}
};
```

### Casting

Type conversions

- Casting is the conversion of one type to another
- Types must be "close" before casting can occur
  - From one numeric type to another
  - Between objects related through inheritance
  - Safe casts are automatic or transparent
  - Lossy casts must be explicit: (destinationType)expression
  - See box p. 52, Schmuller
- Up-casting (from a subclass to a super class) does not require an explicit cast
- Down-casting (from a super class to a subclass) requires an explicit cast

```java
Shape C = new Square(); // up-cast
Square C0 = (Square) C; // down-cast
```
Polymorphism
The requirements
- Polymorphism is the selection of the correct method, from a list of possible methods with the same signature, at the moment of the method call
  - Java methods are polymorphic by default
  - C++ functions are made polymorphic with the “virtual” keyword
- Polymorphism requires
  - inheritance
  - upcasting
    - object name must be an address variable (pointer or reference in C++)
    - object name (variable) is of super class type (may be abstract)
    - object is of subclass type
  - method or function overriding
  - non-polymorphic methods and functions are unaffected
- Third defining feature of the object-oriented model

Polymorphism Example
Dynamic or run-time binding
- User given a choice
- “new” in switch
- Cannot determine at compile time—selection deferred until runtime.

A Functional Model Solution
+ Memories, of the way we were *

```cpp
enum Shape = {
    CIRCLE, SQUARE, TRIANGLE
};

struct Drawable {
    void* object;
    enum Shape shape;
};

struct Drawable d;

switch(d.shape) {
    case CIRCLE :
        drawCircle(d.object);
        break;
    case SQUARE :
        drawSquare(d.object);
        break;
    case TRIANGLE :
        drawTriangle(d.object);
        break;
}
```

The Power of Polymorphism
Modifying the “program”
- Add a new Shape: Ellipse
- Functional model
  - Create Ellipse functions and compile
  - Update user choice menu and switch statement; recompile
  - The draw function call (which in a more complex example might represent dozens of functions) may be spread throughout the program—find every location where an Ellipse may be drawn and update the switch or if-then statement that determines what to draw
  - Recompile all effected modules and relink the executable (i.e., run make)
- Object-Oriented model
  - Create Ellipse class and compile
  - Update user choice menu and switch statement; recompile
  - (C++) only relink object files to create executable

Determining Polymorphism
Which method or function is called?
- Shape s = new Square();
- void render(Shape s);
- render(new Shape());
- Non-polymorphic call (default in C++)
  - Method/function belongs to the class named on the left hand side of the assignment operator
- Polymorphic call (default in Java™)
  - Method/function belongs to the instantiated class on the right hand side of the assignment operator
- Example
  - Non-polymorphic: s.draw() // Shape draw method
  - Polymorphic: s.draw() // Square draw method

Abstract Classes and Polymorphism
A review and an update
- Abstract classes are typically used with polymorphism
  - Super class may be abstract
  - Super class may have concrete data and concrete operations
  - Subclasses MUST override abstract super class operations
  - Super class becomes a generic type specifier used with upcasting
- Example
  ```cpp
  class Shape {
      abstract Shape {
      private:
          void virtual draw() = 0;
      };
  };
  ```
  - Abstract functions/methods do not have a body

©2002, Delroy A. Brinkerhoff
Implementing Polymorphism: C++
Implemented through pointers and redirection

- Objects containing a virtual function embed a "vptr"
- vptr points to the class virtual table: VTABLE
- VTABLE points to individual virtual functions
- Constructors run from the super class to the subclass
- Vptr is set by each constructor as it runs

Objects

<table>
<thead>
<tr>
<th>VTABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
</tr>
<tr>
<td>Circle</td>
</tr>
<tr>
<td>Rectangle</td>
</tr>
<tr>
<td>Triangle</td>
</tr>
</tbody>
</table>

Implementing Polymorphism: Java
Implemented through pointers

- sqr points to a "handle"
- The handle points to
  - Circle object (data)
  - Circle method area

- Shape constructor sets handle to point to Shape draw method
- Circle constructor updates handle to point to Circle draw method

```
Shape s = new Shape();
Circle c = new Circle();
```

Implementing Association
A peer-to-peer connection

- Associations are connections between peer classes
- Allows objects to send each other messages
- Allows objects to call each other's functions/methods
- A link is an instance of an association and connects objects
- Implemented as address variables (references in Java, pointers in C++)

- Associated objects
  - May have different life times
  - May change relationships dynamically or persist once created
  - May participate in multiple associations

- Associations are usually bidirectional
  - The association may be named or labeled
  - Optional arrow head indicates the direction that the name is read
  - Can be unidirectional (line becomes an arrow)

Association Adornments
Identifying class participation

- A role is the part played by or the characteristic and expected behavior of an object
- Qualifiers are used for lookup (i.e., it distinguishes a specific object in a many relationship)

```
Person lifeNumber : int
```

Multiplicity / Cardinality
one-to-one, one-to-many, and many-to-many

- Multiplicity
  - 1 exactly 1
  - 0..1 0 or 1
  - 1..* 1 or more
  - 0..* 0 or more
  - 2..4 2 or 4 (other numbers are permitted)
  - 2..4 2, 3, or 4 (other numbers are permitted)
  - read .. as "to" (i.e., a range) and "," as or; see Schmuller p. 49

- 1 Contractor works on 1 to many Projects

<table>
<thead>
<tr>
<th>Contractor</th>
<th>WorksOn</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1..*</td>
<td></td>
</tr>
</tbody>
</table>
Multiple Associations
Optional connection

- It’s possible to have multiple associations between the same classes
  - Use names (label) or roles to clarify

```
Instructor * 1..*
teachesFor * 1..* Department
- chair *
```

Reflective Associations
Also known as reflexive

- It is possible to have both ends of an association connect to the same class
  - This means that you can associate two instances of the class
  - Use roles to clarify

```
Person
- supervisor * 1
- employee +
```

“OR” Associations
Constraining an association

- Specifies that, over a set of associations, exactly one is manifest for each associated object

```
Person
\( \{\text{or}\} \) ridesOn
AirPlane
\( \{\text{or}\} \) ridesOn
Train
```

N-ary Associations
Associations between three or more classes

- Most associations are binary (> 99% ?)
  - Some associations may be tertiary (< 1% ?)
  - Some associations may involve 4 or more classes (?)

```
Contractor
Project
ProgrammingLanguage
```

Association Classes
Also known as “link attributes”

- Attributes belong to the association not the classes
  - Attributes may be put in one of the classes in one-to-one and one-to-many associations
  - Association classes are necessary in many-to-many associations

```
Contractor 1..* 1..* Project
WorksOn
- time - rate
```

Aggregation
Whole-Part: assembly, has a, a part of, contains a

- Conceptual form of association
  - Weak ownership or binding
  - Parts exist independent of whole (i.e., parts and whole have independent lifetimes)
  - Parts may be shared by multiple wholes
  - Parts may change during execution (i.e., whole/part binding is dynamic)
  - Asymmetric (i.e., one way)
    - If A is part of B, B cannot be part of A
    - Motor & Transmission do not reference Automobile
  - Navigation may be bidirectional

```
Student

Address

Schedule

Part classes

See the example Java & C++ code on the class web page
Implementing Aggregation

C++ and Java

class Address {}
class Schedule {}
class Student {
    private:
        Address* address;
        Schedule* schedule;
    public:
        void setAddress(Address* a) {
            address = a;
        }
        void setSchedule(Schedule* s) {
            schedule = s;
        }
}

public class Address {}
public class Schedule {}
public class Student {
    private Address address;
    private Schedule schedule;
    public void setAddress(Address a) {
        address = a;
    }
    public void setSchedule(Schedule s) {
        schedule = s;
    }
}

“OR” Aggregation

Special case of “OR” Association

- Constraint limiting containment

Composition

Whole-Part: containment or “contains a”

- Form of association; variation of simple aggregation
  - strong ownership or binding
  - whole & parts have coincident lifetimes (i.e., whole is responsible for creation and destruction of parts)
  - parts belong to one whole
  - whole & parts bound for entire execution
  - antisymmetric (i.e., one way)
    - if A is part of B, B cannot be part of A
    - Motor & Transmission do not reference Automobile
  - navigation is from whole to part only

Implementing Composition In C++

“Tight” form of aggregation

class Transmission {}
class Motor {}
class Automobile {
    private:
        Transmission trans; // case 1
        Motor motor; // case 2a
    public:
        Automobile(int d) : trans(d) // case 2b
        {
        }
}

case 1: Part (Transmission) has a default ctor or if Part ctor arguments are known at compile time

case 2: Part (Motor) ctor arguments are passed in through the Whole (Automobile) ctor

Implementing Composition In Java

“Tight” form of aggregation

public class Transmission {}
public class Motor {}
public class Automobile {
    private Transmission trans = new Transmission(); // case 1
    private Motor motor; // case 2a
    public Automobile(int displacement) // case 2b
    {
        motor = new Motor(displacement);
    }
}

case 1: Part (Transmission) has a default ctor or if Part ctor arguments are known at compile time

case 2: Part (Motor) ctor arguments are passed in through the Whole (Automobile) ctor

Dependency

Delegation or using

- One object needs a service/operation of another object
- Maintains object integrity (encapsulation) & responsibility
- Is implemented through local-scope objects
- passing an object as a parameter to a function/method
- instantiating a local object
- Ephermal form of association
- Also known as delegation or using

Dependent object

Independent objects

©2002, Delroy A. Brinkerhoff Class Relationships
Implementing Dependency In C++

Two techniques

class Foo
{ public:
  void doFoo( )
  {
    ...
  }
};

class Bar
{ public:
  void doBar( )
  {
    ...
  }
};

Implementing Dependency In Java

Two techniques

public class Foo
{ public:
  void doFoo( )
  {
    ...
  }
};

public class Bar
{ public void doBar( )
  {
    ...
  }
};