A Critical Look at Inheritance

"You can choose your associates, but you're stuck with your ancestors."

Inheritance in OOD

- Inheritance is often held to be sacrosanct in OOD.
- Tendency for OO developers to gauge the success of their efforts by the complexity of their inheritance hierarchy.
- It is interesting to note that inheritance hierarchy examples in OO texts seldom deal with software design problems.

Inheritance--The Reality

- Inheritance is a complex issue.
  - Many different types of inheritance relationships.
  - Basic notions differ among OO languages
  - Some controversial issues--e.g. multiple inheritance.
  - Inheritance can break encapsulation.
  - Poorly conceived inheritance relationships can frustrate system reliability, maintainability, and evolvability.
- Inheritance is neither inherently good or bad. It must be used in a disciplined manner.

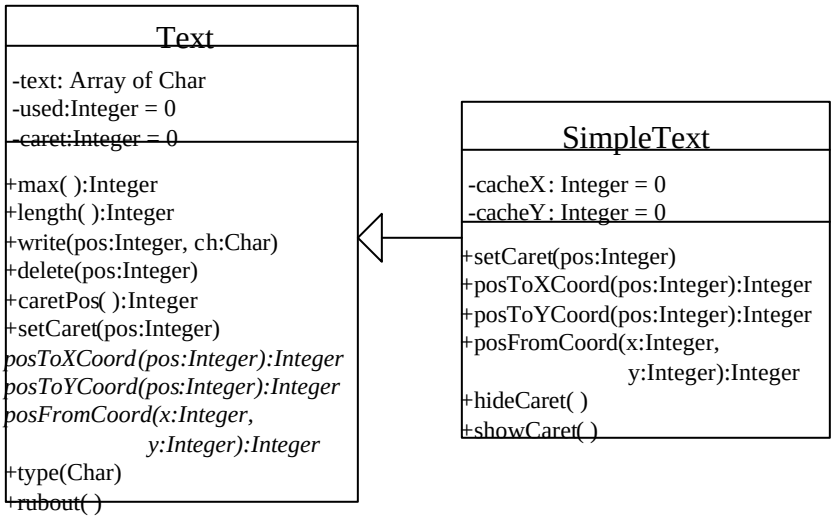
Inheritance--A Simple Classification

- Subclassing
  - inheritance of implementation fragments/code from a superclass.
- Interface Inheritance
  - inheritance of contract fragments/interfaces.

The Complexities of Subclassing

- Methods of a class may freely invoke each other.
- Subclasses may override inherited methods.
- Subclass methods may call methods of superclasses, including overridden superclass methods.
- This is actually a form of “callback” from subclass to superclass.

Inheritance Issues Example



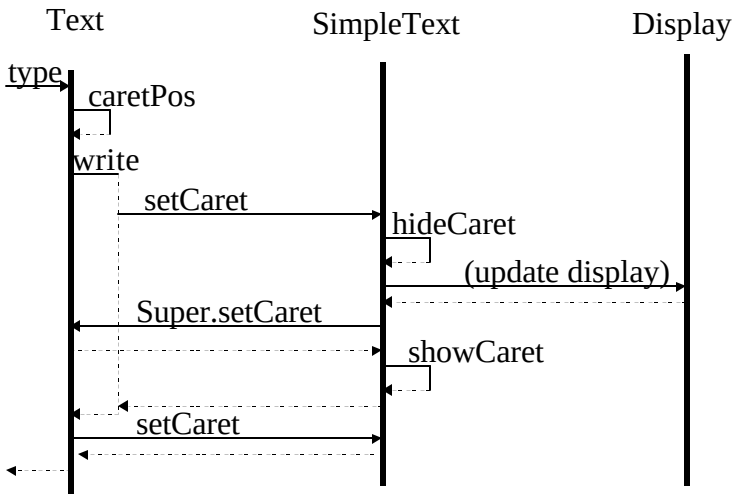
Example--Continued

```
abstract class Text {
        .
        .
        .
  private int caret = 0;
        .
        .
        .
  void setCaret(int pos)  {
    caret = pos;
   }
        .
        .
        .
}
```

```
class SimpleText extends Text {
        .
        .
        .
  void setCaret(int pos) {
    int old = caretPos( );
    if (old != pos) {
      hideCaret( );
      super.setCaret(pos);
      showCaret( );
    }
        .
        .
        .
}
```

Example--Continued

Interaction diagram resulting from call to method **type** of Text class:



Example--Continued

A new version of Text class that "breaks" the subclass SimpleText:

```
abstract class Text  {
        .
        .
        .
  void write (int pos, char ch) {
    int i;
    for ( i = used; i > pos; i--)
      text[i] = text[i-1];
    used = used + 1;
    if (caret >= pos)
      caret = caret +1;
    text[pos] = ch;
  }
        ...
```

Inheritance Issues--The Fragile Base Class Problem

- There is an implicit interface between a class and its ancestor classes (superclasses).
  - Syntactic aspect--Does a class need to be recompiled due to purely syntactic changes among it superclasses?
  - Semantic Aspect--How dependent is a subclass upon changes in the implementation of its superclasses?

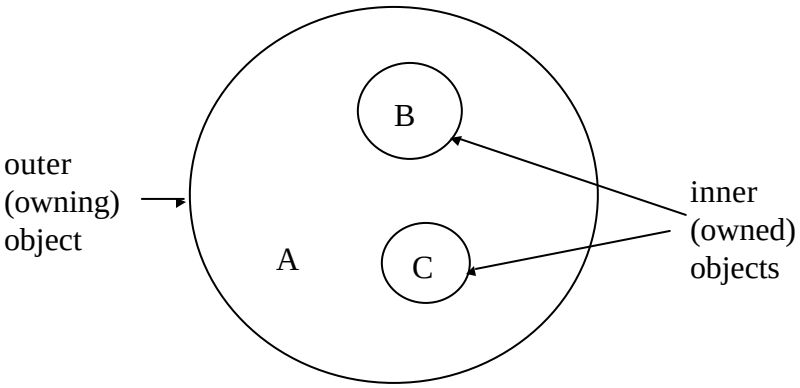
Dealing With Class-Subclass Dependencies

- Specialization Interface
  - Interface between a class and its subclasses
  - For Java and C++, the specialization interface consists of the public and protected interface of the superclass.
- Various methods have been proposed to control behavior across a specialization interface, but these are largely of theoretical interest.

Alternatives to Inheritance--Object Composition

- Object composition--composition of behavior based upon references among objects rather than inheritance relations.
- Based upon “part of” relationship among objects.
  - Suppose object A requests help from object B
  - B is “part of” A is references to B do not leave A.

Object Composition

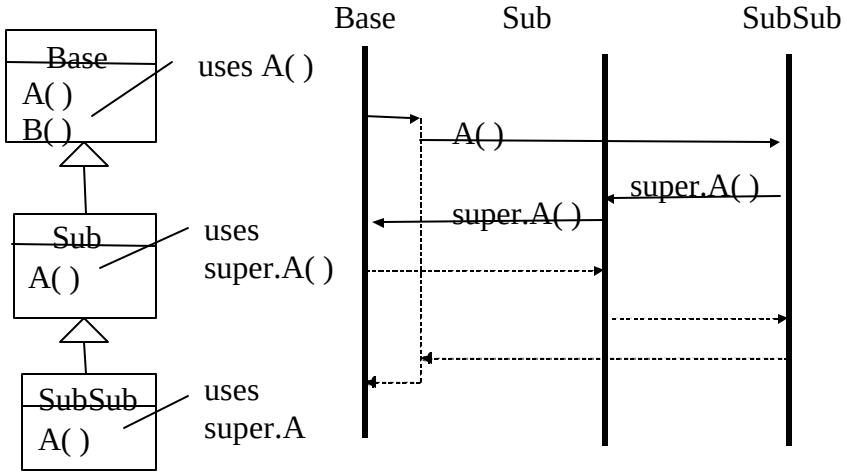


Note: A *reuses* the implementation of objects B and C

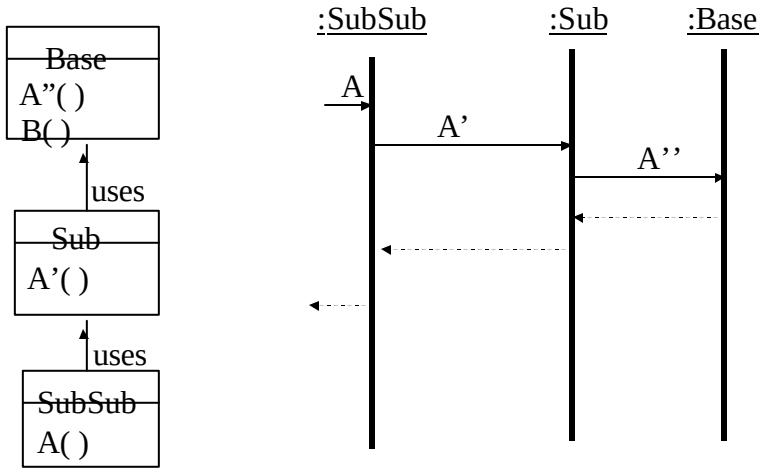
Composition Versus Inheritance

- An instantiated object has one notion of *self* even though it may inherit parts of its implementation from several superclasses.
- "Self-recursive invocations of methods always return to the overriding version in the lowest level subclass
- Composed objects do not have a common self--outer object does not share identify with inner objects.

Example of self-recursive calls



Example of Composition



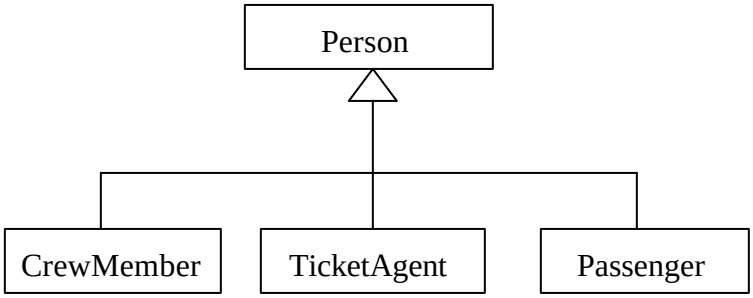
Composition--Additional Observations

- Composition requires that object interactions, including recursive interactions among objects, be explicitly designed-in rather than an implicit by-product of implementation inheritance.
- Composition is a relationship between instantiated objects, not a relationship between classes.
- Composition can be made as general as subclassing by use of *delegation*.

Inheritance Versus Composition--Another Example

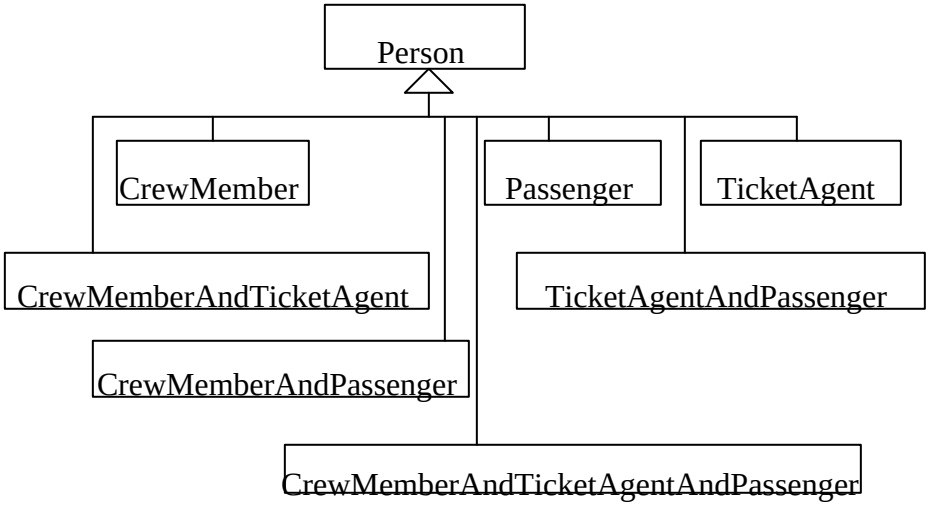
- Inheritance is generally not appropriate for "is a role played by" relationships.
- For instance, consider roles in an airline reservation system:
  - passenger
  - ticket agent
  - flight crew
  - etc.

Roles Example--A Potential Inheritance Hierarchy

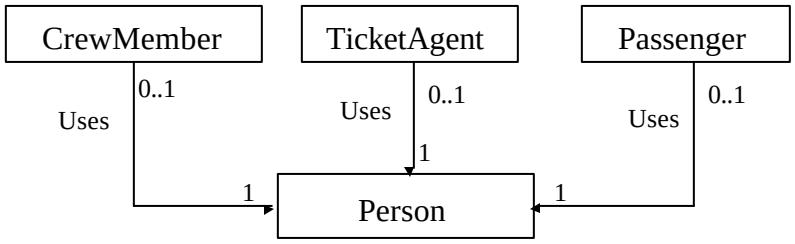


Problem with this approach: a person may play different roles. An instantiated subclass can only represent one role.

Roles Example--An Attempt to Fix the Inheritance Hierarchy

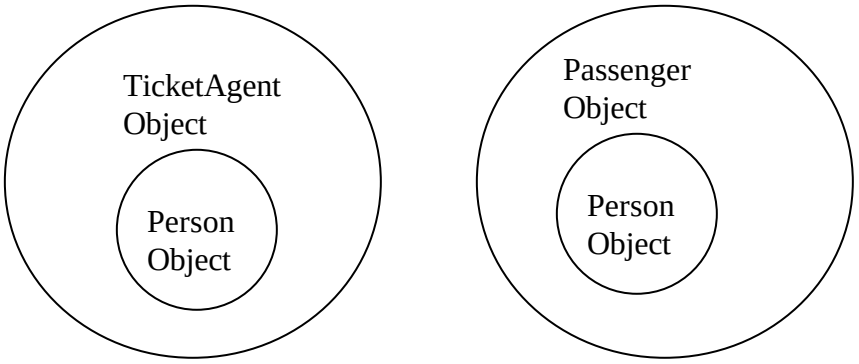


Roles Example--A More Rational Solution using Composition

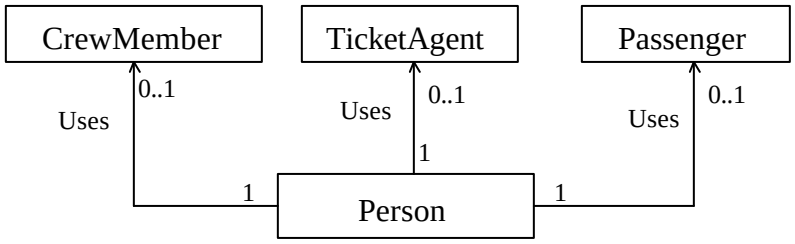


Note: Many authors refer to this as *delegation*.

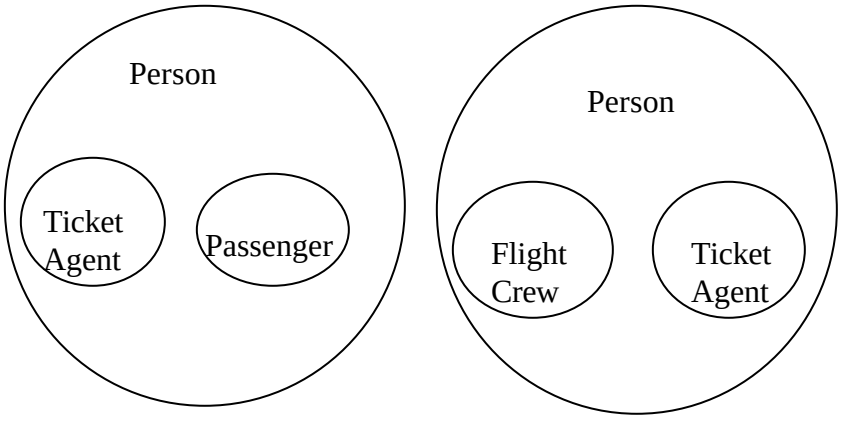
Object Encapsulation via Composition



An Alternative Composition for the Roles Example



Object Encapsulation for Alternative Composition



Inheritance Versus Composition-- Some Guidelines

- It is generally not a good idea to use inheritance for the following purposes:
  - To represent dynamically changing alternative roles of a superclass
  - To hide methods or attributes inherited from a superclass.
  - To implement a domain-specific class as a subclass of a utility class.

Potential Drawbacks of Composition (Delegation)

- There may be some minor performance penalty for invoking an operation across object boundaries as opposed to using an inherited method.
- Delegation can't be used with partially abstract (uninstantiable) classes
- Delegation does not, in and of itself, impose any disciplined structure on the design (but neither does a class hierarchy).