

Delegation vs inheritance

Delegation vs inheritance

[Mark Grand98]

▶ Inheritance

- ▶ defines a new class, which use the interface of a parent class while adding extra, more problem-specific methods.

▶ Delegation

- ▶ is a way of reusing and extending the behavior of a class by writing a new class that incorporates the functionality of the original class by using an instance of the original class and calling its methods.

- ▶ **No.1 issue in OO is if a class A should inherit from B or A should use B.**

Motivation

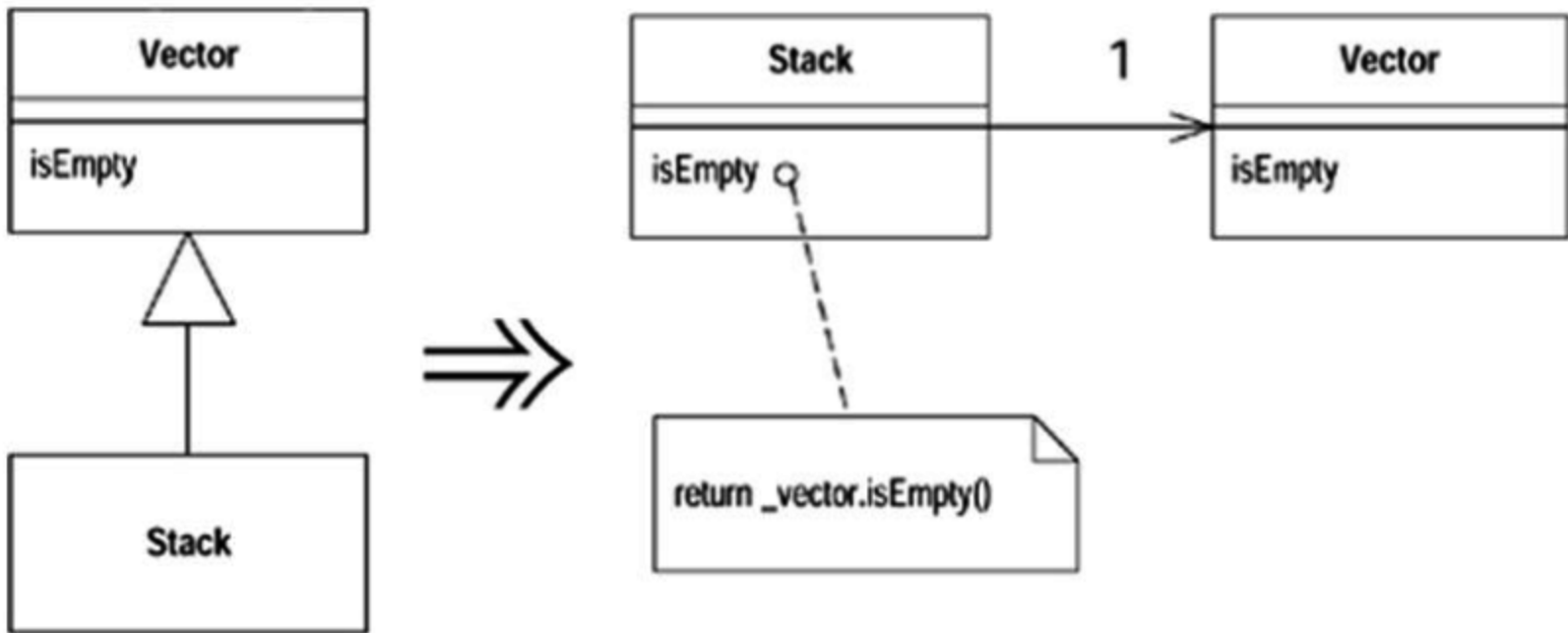
- ▶ Inheritance is a wonderful thing, but sometimes it isn't what you want.
 - ▶ Often you start inheriting from a class but then find that many of the **superclass operations aren't really true of the subclass**. In this case you have an interface that's not a true reflection of what the class does.
 - ▶ Or you may find that you are **inheriting a whole load of data** that is not appropriate for the subclass.
 - ▶ Or you may find that there are **protected superclass methods that don't make much sense** with the subclass.

Motivation (continued)

- ▶ You can live with the situation and use convention to say that although it is a subclass, it's using only part of the superclass function.
 - ▶ But that results in code that says one thing when your intention is something else—a confusion you should remove.
- ▶ **By using delegation** instead, you make it clear that you are making only partial use of the delegated class. You control which aspects of the interface to take and which to ignore.
- ▶ **The cost** is extra delegating methods that are boring to write but are too simple to go wrong.

Replace Inheritance with Delegation

- ▶ Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



Example

- ▶ One of the classic examples of inappropriate inheritance is making a stack a subclass of vector.
- ▶ In this case I use a simplified form of stack:

```
class MyStack extends Vector {  
    public void push(Object element) {insertElementAt(element,0);}  
    public Object pop() { Object result = firstElement();  
                        removeElementAt(0); return result; }  
}
```

- ▶ Looking at the users of the class, I realize that clients do only four things with stack: push, pop, size, and isEmpty. The latter two are inherited from Vector.

Example (continued)

- ▶ I begin the delegation by creating a field for the delegated vector. I link this field to this so that I can mix delegation and inheritance while I carry out the refactoring:

```
private Vector _vector = this;
```

- ▶ Now I start replacing methods to get them to use the delegation. I begin with push:

```
public void push(Object element) {  
    _vector.insertElementAt(element,0);  
}
```

- ▶ I can compile and test here, and everything will still work.

Example (continued)

► Now pop:

```
public Object pop() {  
    Object result = _vector.firstElement();  
    _vector.removeElementAt(0);  
    return result;  
}
```


Example (continued)

- ▶ Once I've completed these subclass methods, I need to break the link to the superclass:

```
class MyStack
    private Vector _vector = new Vector();
```

- ▶ I then add simple delegating methods for superclass methods used by clients:

```
public int size() { return _vector.size(); }
public boolean isEmpty() { return _vector.isEmpty(); }
```

- ▶ Now I can compile and test. If I forgot to add a delegating method, the compilation will tell me.

Mechanics

- ▶ Create a **field in the subclass** that refers to an instance of the superclass. Initialize it to this.
- ▶ Change **each method defined in the subclass** to use the **delegate field**. Compile & test after changing each method.
 - ▶ *You won't be able to replace any methods that invoke a method on super that is defined on the subclass, or they may get into an infinite recurse. These methods can be replaced only after you have broken the inheritance.*
- ▶ **Remove the subclass declaration** and replace the delegate assignment with an assignment to a new object.
- ▶ For each superclass method used by a client, add a simple delegating method.
- ▶ **Compile and test.**

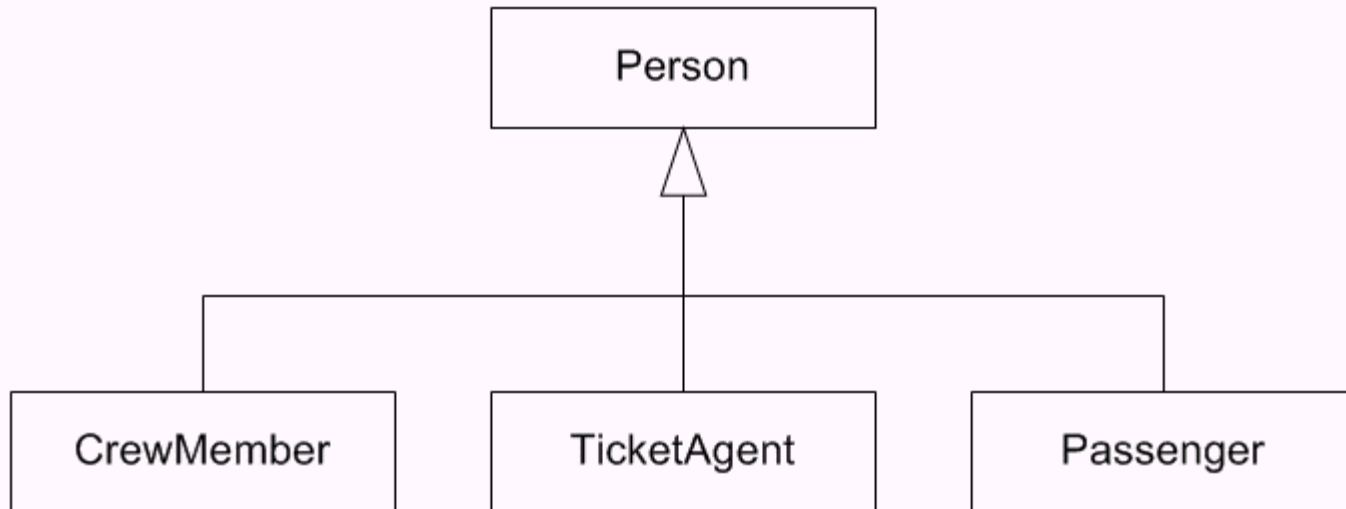
Delegation (When not using inheritance)

[Mark Grand98]

- ▶ Inheritance is a common way of extending and reusing the functionality of a class. However, inheritance is inappropriate for many situations:
 - ▶ Inheritance is useful for capturing is-a-kind-of relationships which are rather static in nature.
 - ▶ is-a-role-played-by relationships are awkward to model by inheritance, where delegation could be a better choice. Using instances of a class to play multiple roles.

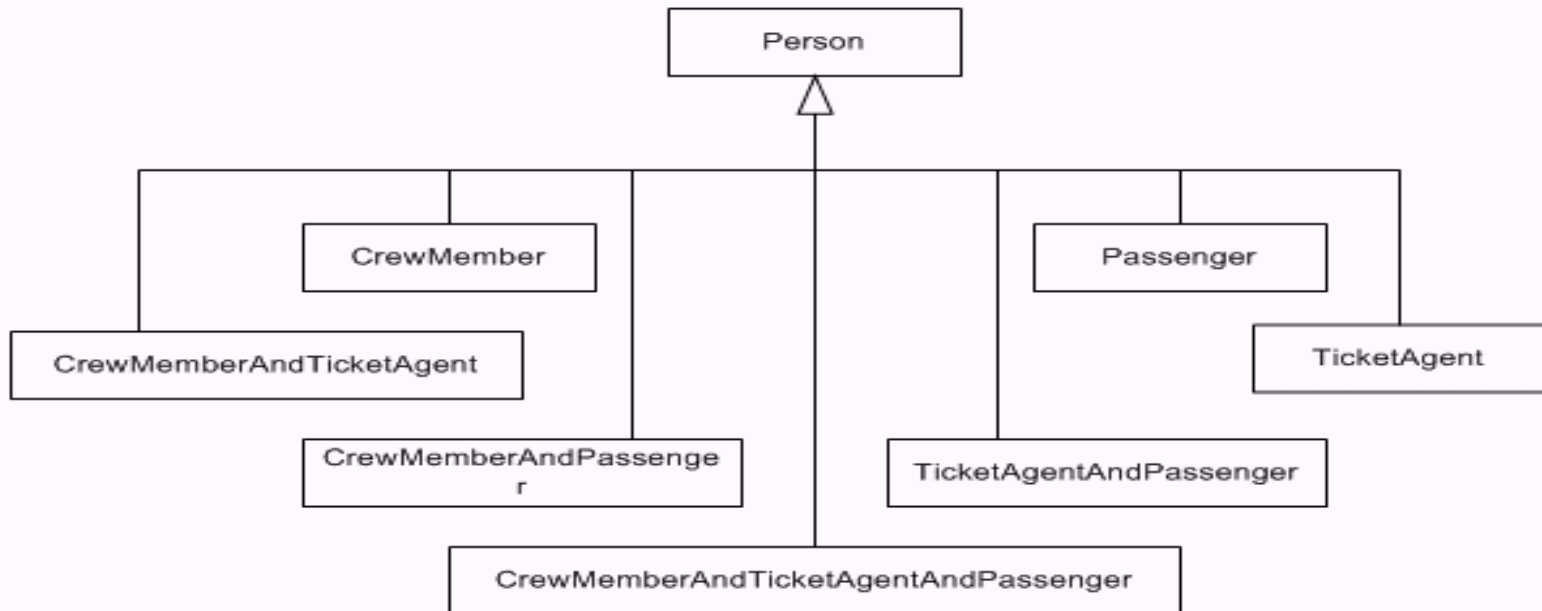
Inheritance vs delegation: changing roles

- ▶ Don't use inheritance where roles interchange.
 - ▶ For example, an airline reservation system may include such roles as passenger, ticket selling agent and flight crew.
 - ▶ A class called Person may use subclasses corresponding to each of these roles.



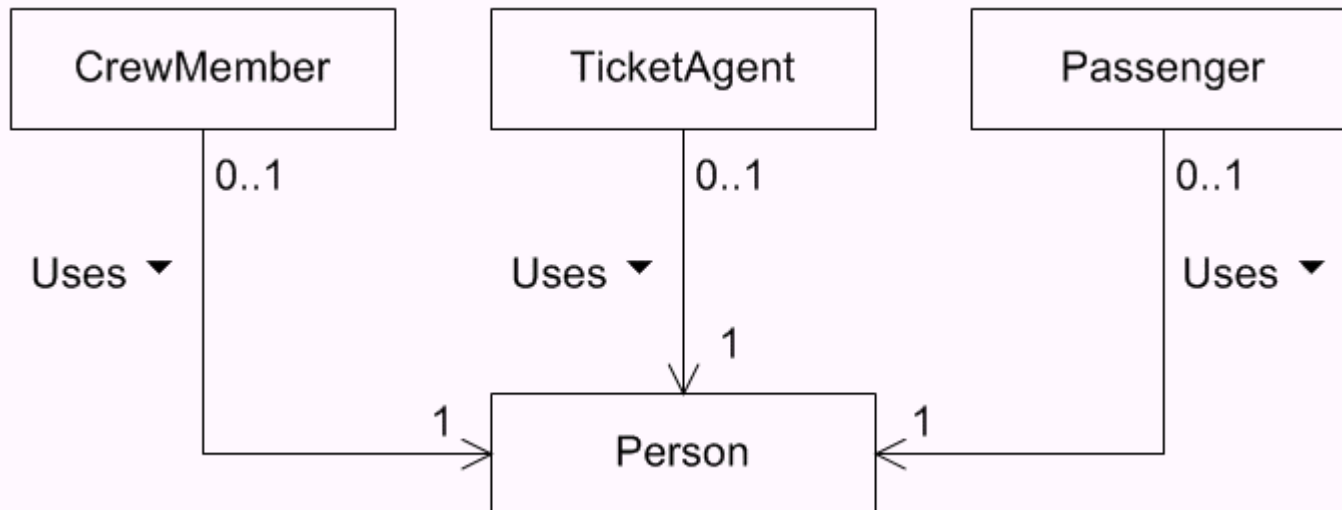
Example (cont'd)

- ▶ The problem is that the same person can fill more than one of these roles.
 - ▶ A person who is normally part of a flight crew can also be a passenger...
 - ▶ This way, the number of subclasses would increase exponentially.

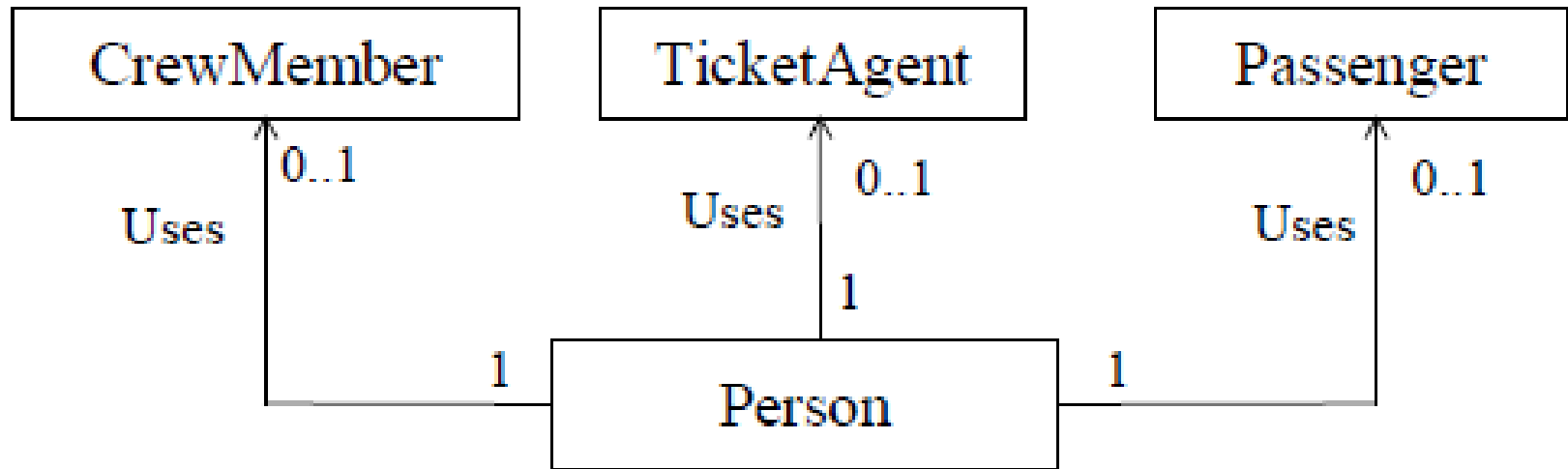


Example (cont'd)

- ▶ If person A, CrewMember, becomes now also a Passenger, a new object Passenger is created, referring A.



Example (cont'd)



- ▶ But then problems with using the specific methods, which were unforeseen.

Inheritance vs delegation:languages

- ▶ In Java or C#, an object cannot change its type once it has been instantiated.
- ▶ So, if your object need to appear as a different object or behave differently depending on an object state or conditions, then use Composition
- ▶ Refer to State and Strategy Design Patterns.
- ▶ If the object need to be of the same type, then use Inheritance or implement interfaces

Inheritance vs delegation: hiding

- ▶ Don't use inheritance if you end up in a situation where a class is trying to hide a method or variable inherited from a superclass.
- ▶ If you define a field in a subclass that has the same name as an accessible field in its superclass, the subclass's field *hides* the superclass's version.
 - ▶ E.g., if a superclass declares a public field, subclasses will either inherit or hide it. (You can't override a field.)
- ▶ If a subclass hides a field, the superclass's version is still part of the subclass's object data; however methods in the subclass can access the superclass's version only by using the super keyword, as in `super.fieldName`.

Inheritance vs delegation: utility classes

- ▶ **Don't use inheritance of a utility class**
 - ▶ you're not in control of the parent class and it may change scope later (inheriting `java.util.Vector` is a very, very bad idea since sun may later declare methods deprecated).
 - ▶ It's always easier to replace changing a class you just use – than one you inherit from.
 - ▶ Besides, inheritance exposes a subclass to details of its parent's class implementation, that's why it's often said that inheritance breaks encapsulation (in a sense that you really need to focus on interfaces only not implementation, so reusing by subclassing is not always preferred).

Places where not to use inheritance (but rather delegation) (continued)

- ▶ Don't use inheritance from a class, which is written very specifically to a narrow problem - because that will make it more difficult to inherit from another class later.
- ▶ Client classes that use the problem domain class may be written in a way that assumes the problem domain class is a subclass of the utility class.
- ▶ If the implementation of the problem domain changes in a way that results in its having a different superclass, those client classes that rely on its having its original superclass will break.

Potential Drawbacks of 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 impose **any disciplined structure** on the design.