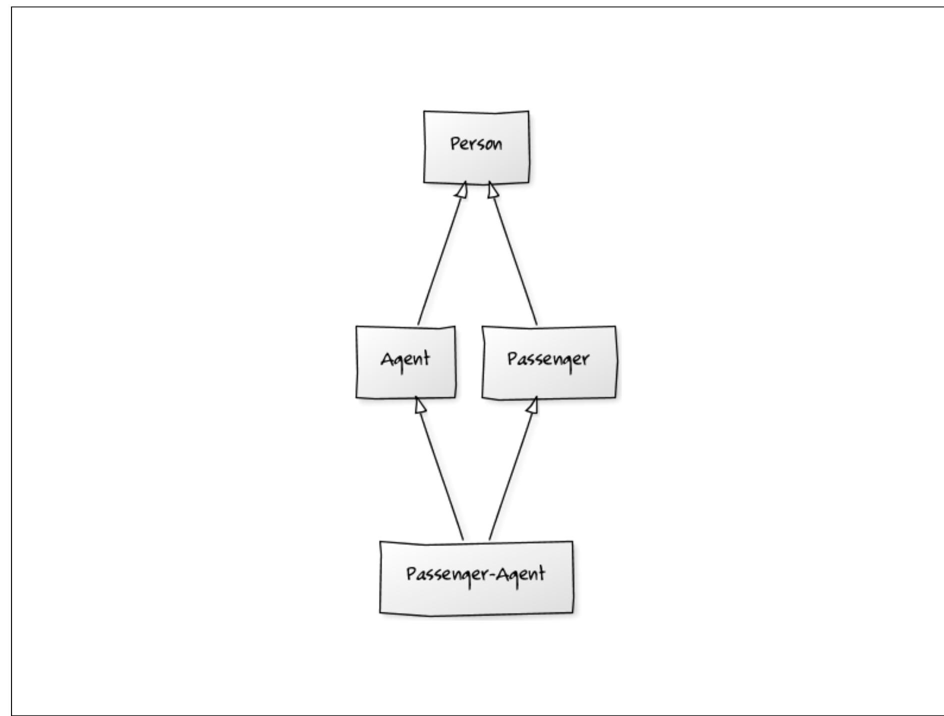
a simple design question

A passenger is a person.

An agent is a person.

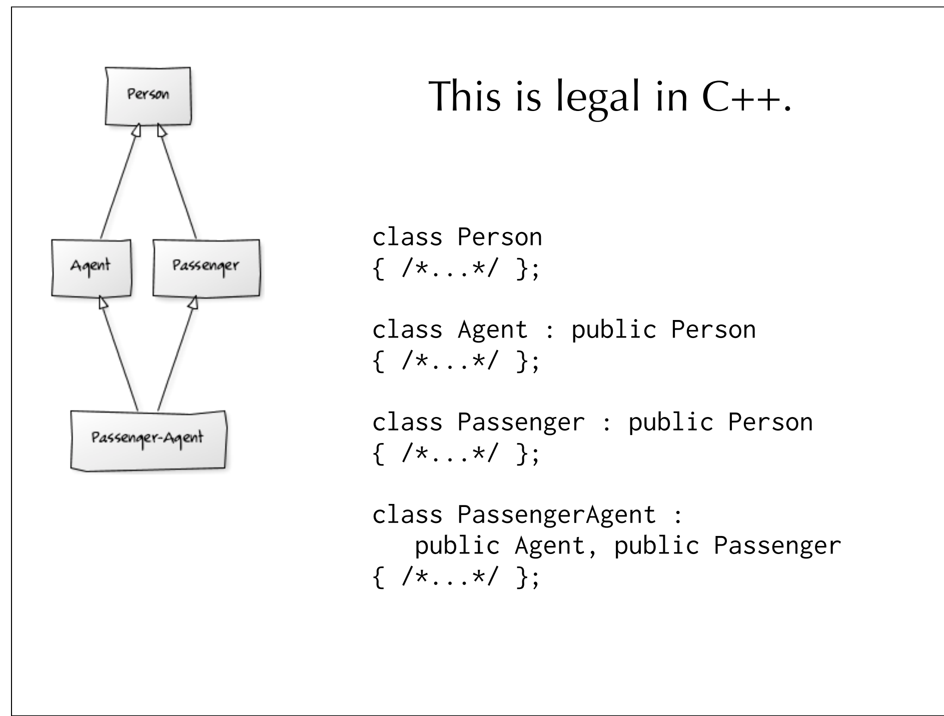
Some people are agents and passengers.

So...

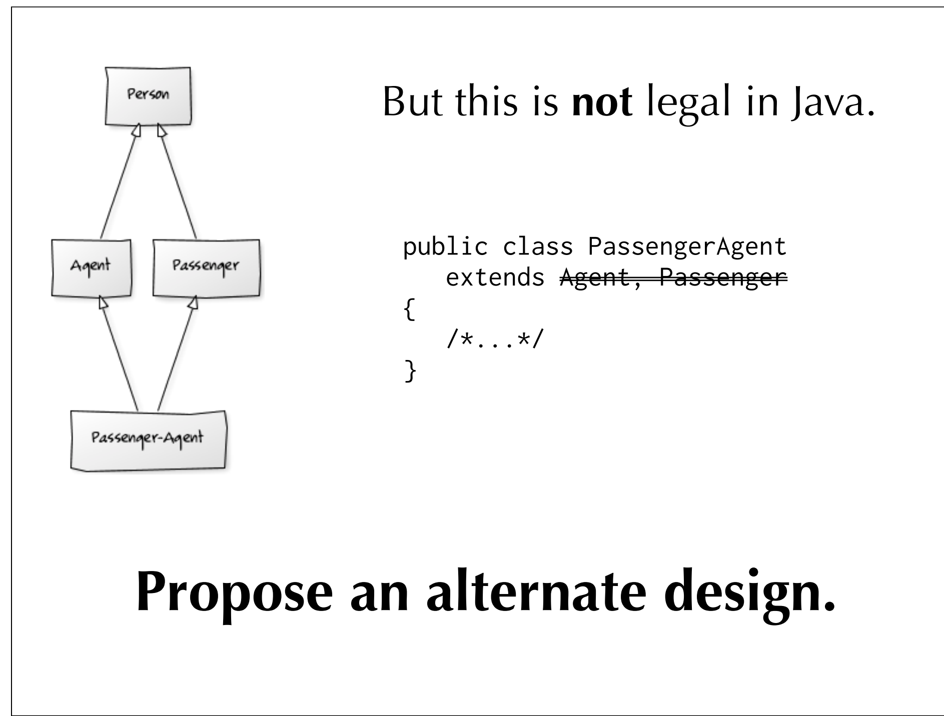


A simple class diagram showing the inheritance relationships among Person, Passenger, Agent, and Passenger-Agent.

I generated the simple class diagrams for this session using YUML, a nice on-line tool at http://yuml.me/

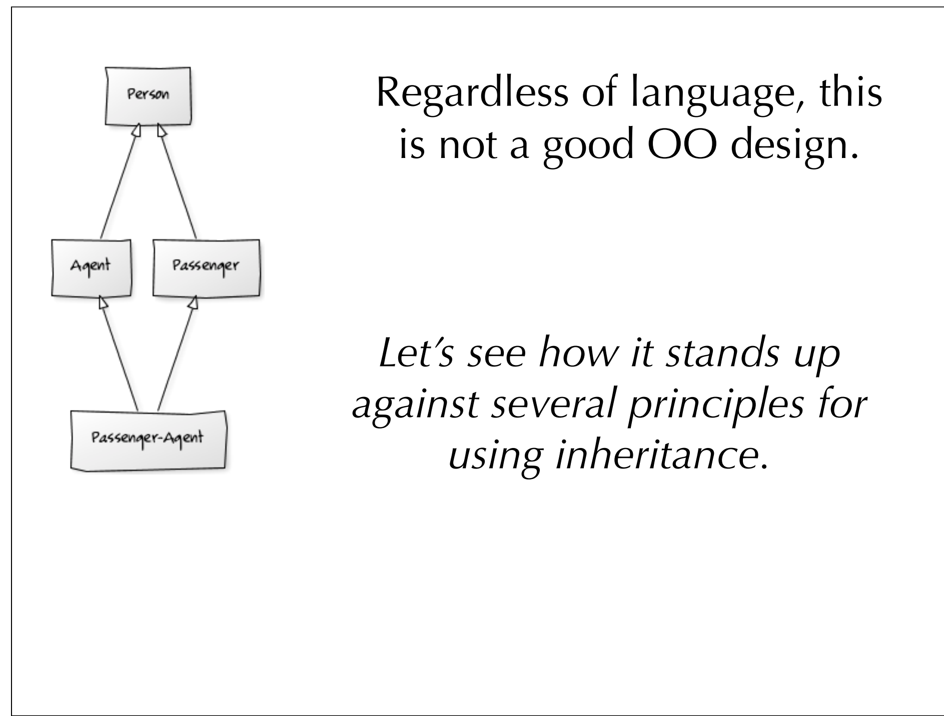


In C++, we can create a subclass with two superclasses using a comma-separated list of base classes.

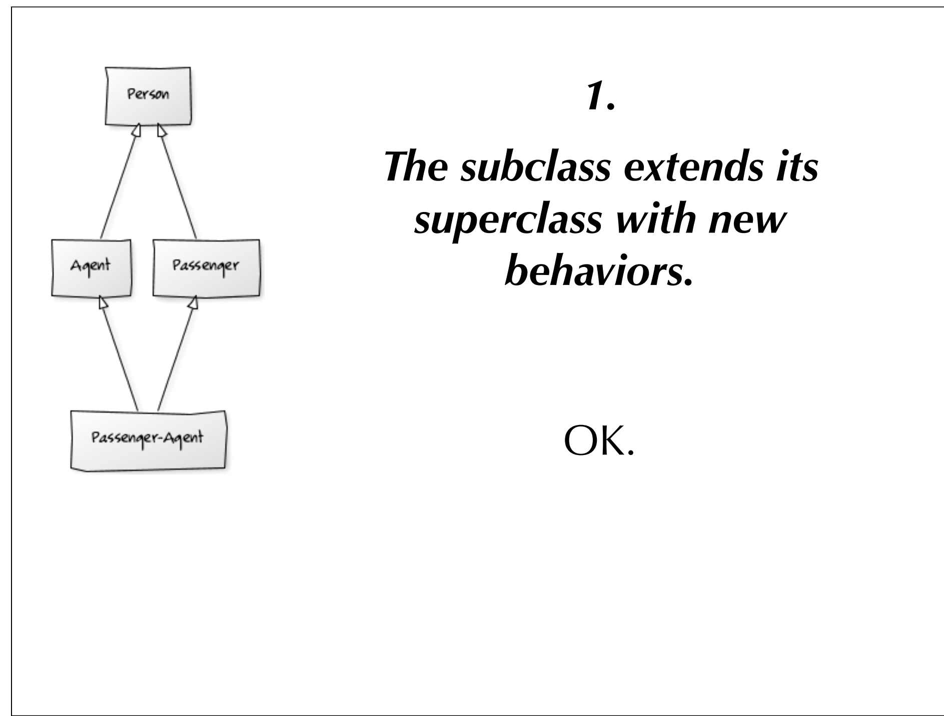


This is not legal in Ada95, either. See http://www.adaic.org/learn/tech/multin.html

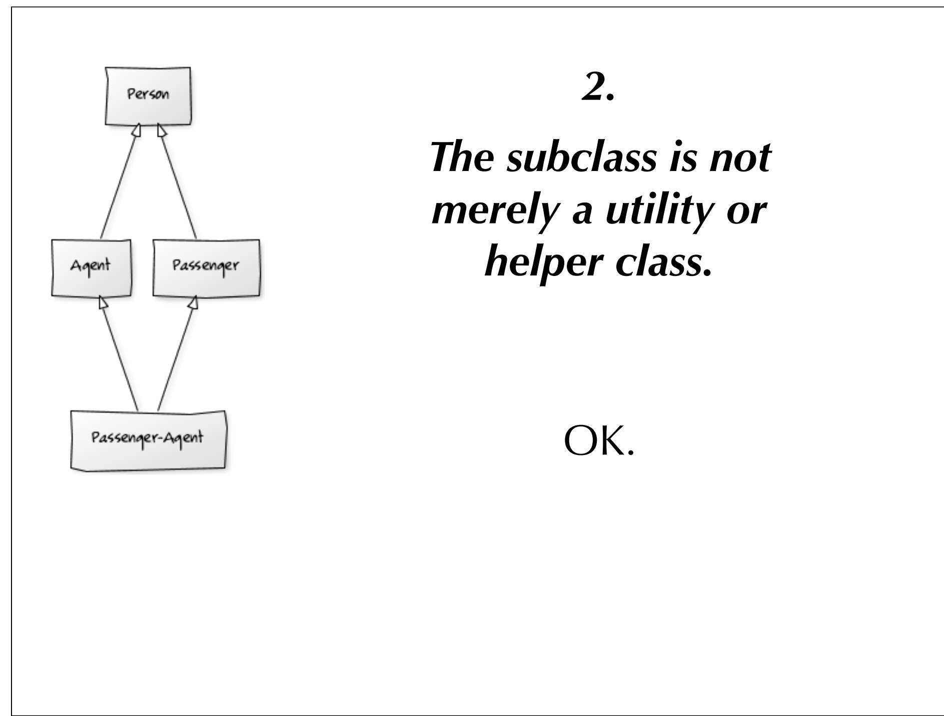
Many designers prefer not to use multiple inheritance anyway. They think it complicates the semantics of programs and can make it harder to modify code. Others use it sparingly but to good effect in complex applications.



Let's see how it stands up against several principles for using inheritance.



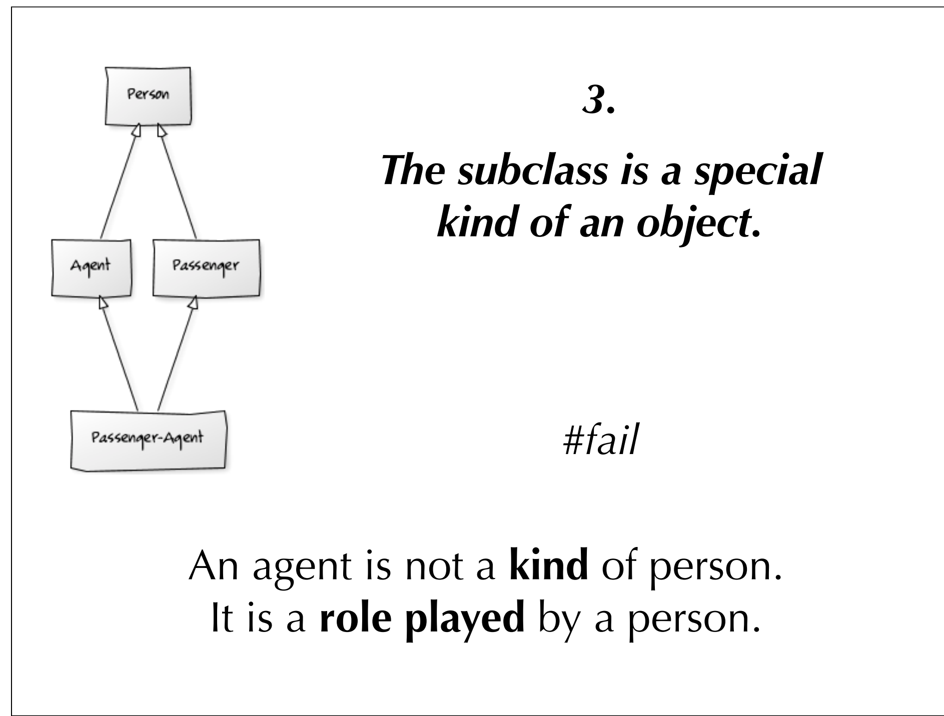
Passenger-Agent does not add new behavior, but at least it doesn't subtract behavior.



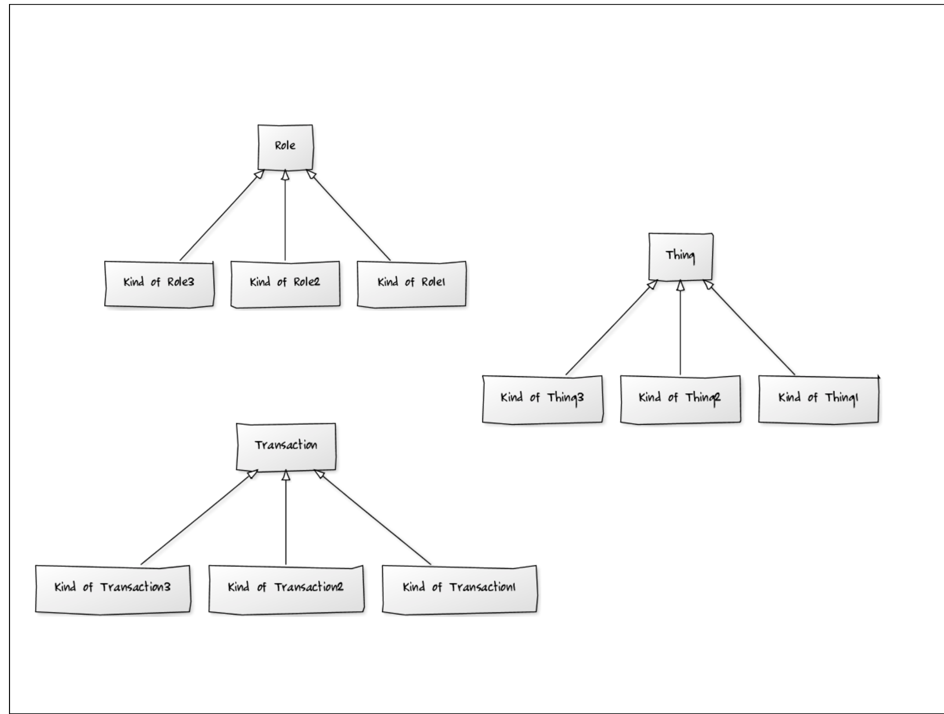
Passenger and Agent do real work. They have attributes and behavior:

Agent
password, authorizationLevel
isAuthorized()

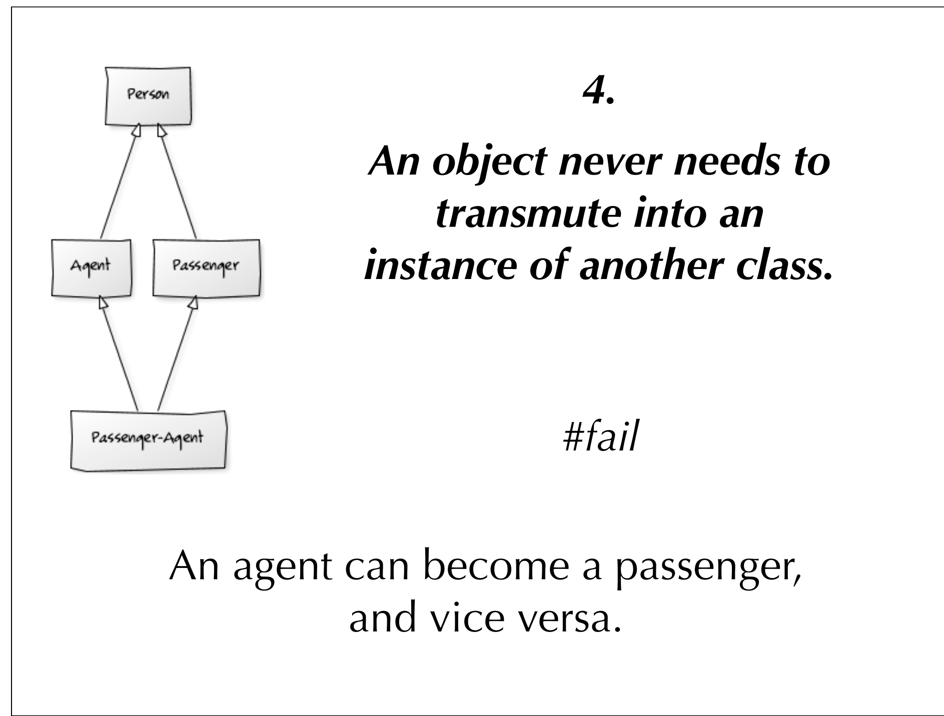
Passenger
type
assessPreferences()



Role played is a common abstraction when modeling problem domains.



Roles and transactions are both common abstractions. Consider inheritance for kinds of things -- the more general abstraction -- after considering more specific kinds of relationships.



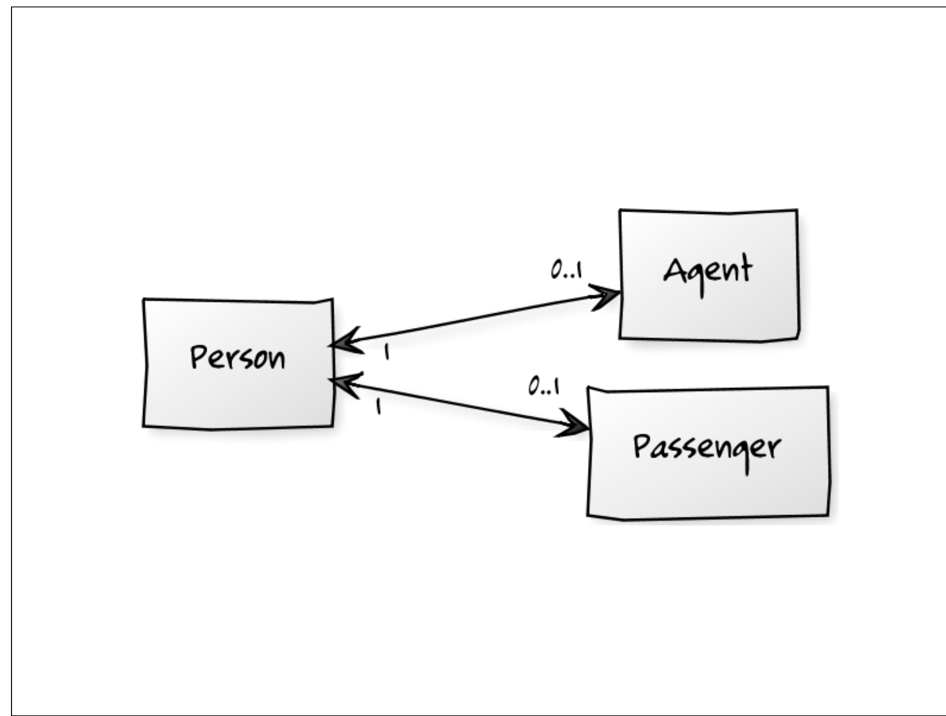
The PassengerAgent class indicates that an agent can even become a passenger without stopping being an agent!

Inheritance is useful,

but

composition is the norm.

So let's use composition with role objects...



A person can be play an agent role, or not. That means it can have 0 or 1 agent role objects. Likewise for the passenger role.

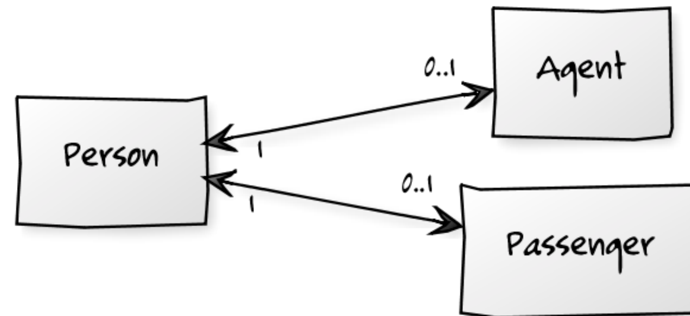
```
public class Person {
   // ...
   private Agent     agentRole;
   private Passenger passengerRole;

   //...

   public void startFlightAs( Passenger p ) {
      passengerRole = p;
   }

   public void stopFlight() {
      passengerRole = null;
   }
   //...
}
```

(If null becomes a problem, what could we do?)

```
public class Passenger {
   // ...

   private Person person;

   //...

   public Passenger( Person p ) {
      person = p;
   }

   //...
}
```

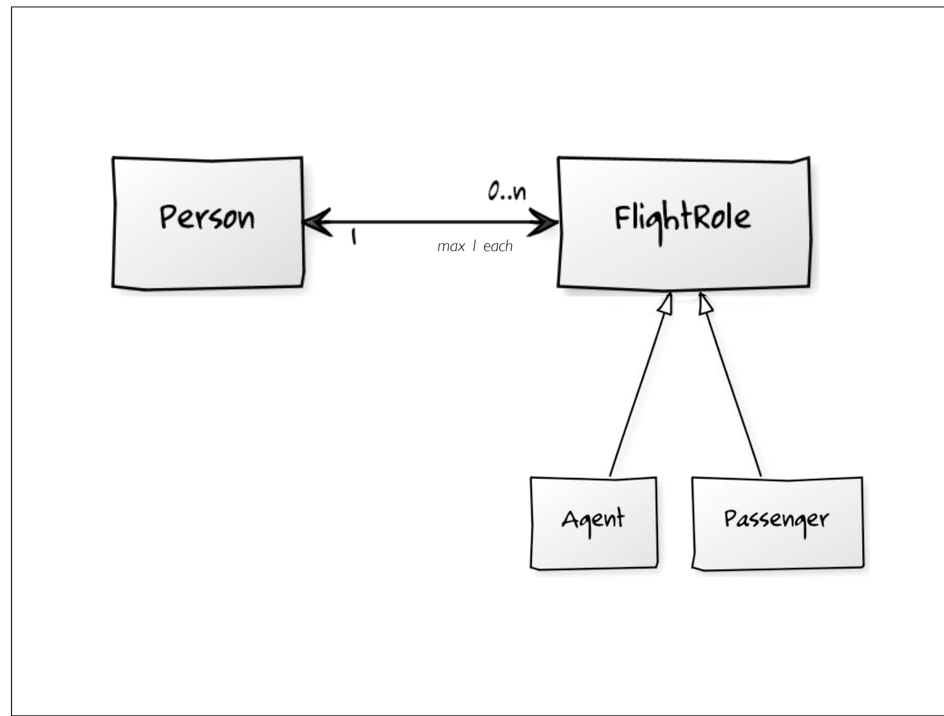
Note that there is no default constructor. A passenger **must** be related to a person.

But...

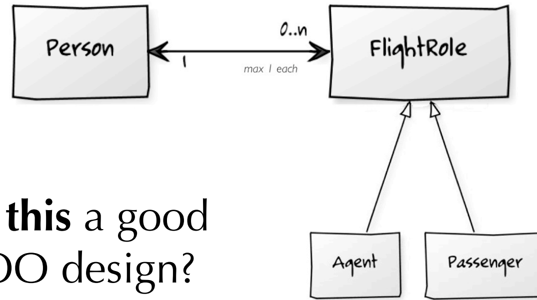


Aren’t **Agent** and **Passenger** special kinds of roles played by a person?

Yes! There is a place for inheritance here...



Inheritance is often used in conjunction with composition. We design small, focused hierarchies of specific kinds of objects, and use them as components in larger-scale design.



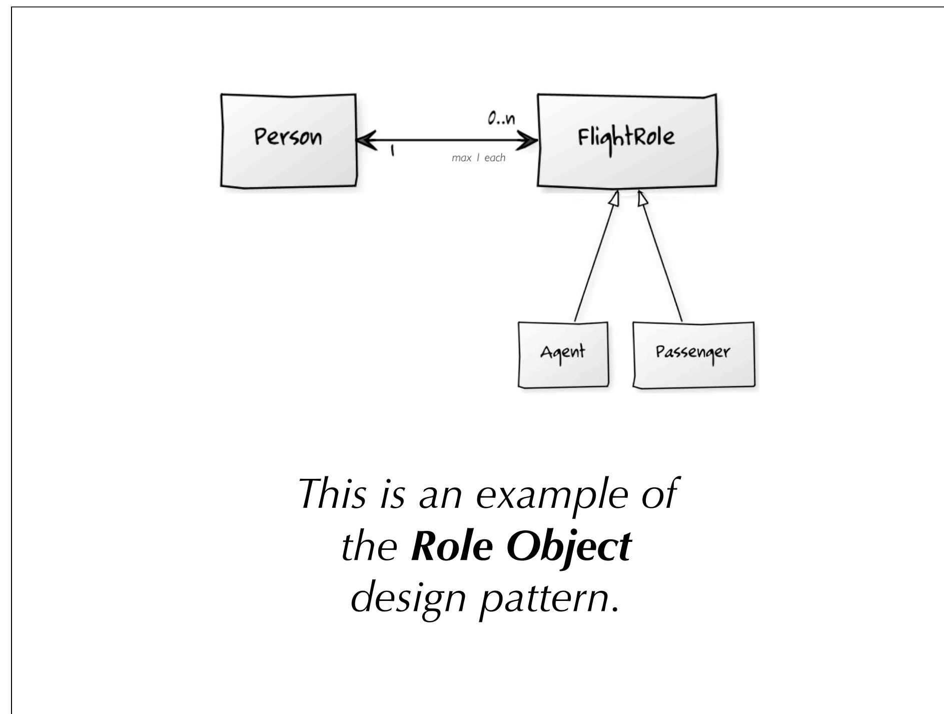
Is **this** a good OO design?

Let's see how it stands up against the same principles of inheritance.

1. The subclass extends its superclass with new behaviors.	Check.
2. The subclass is not merely a utility or helper class.	Check.
3. The subclass is a special kind of an object.	Check.
4. An object never needs to transmute into an instance of another class.	Check.

You don’t need to settle for designs that feel wrong.

There is often a better design to be found.



http://www.cix.co.uk/~smallmemory/almanac/BaumerEtc99.html

Published in **Pattern Languages of Program Design 4**, pages 15-31.
An earlier version appeared at PLoP 1997:
http://hillside.net/plop/plop97/Proceedings/riehle.pdf

these four principles
of inheritance
are examples of
design heuristics

What does **heuristic** mean? If you have had AI, then you should have a guess...

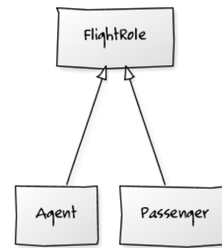
There are many design heuristics.

a commonsense rule intended
to **increase the probability** of
solving some problem

It's not a recipe. It cannot guarantee an answer. It guides us in the direction of right answers.

Note: a design pattern cannot guarantee an answer, either. It guides us in the direction of right answers. It is a tool to be used. It can be used well, and it can be used poorly.

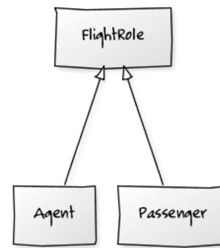
There are many design heuristics. There are many even heuristics involving inheritance. Here are a few examples from a great book, **Object-Oriented Design Heuristics** by Arthur Riel.



A superclass should not know anything about its subclasses.

Why?

Violating this rule leads to the Fragile Base Class Problem.



All data in a superclass should be private.

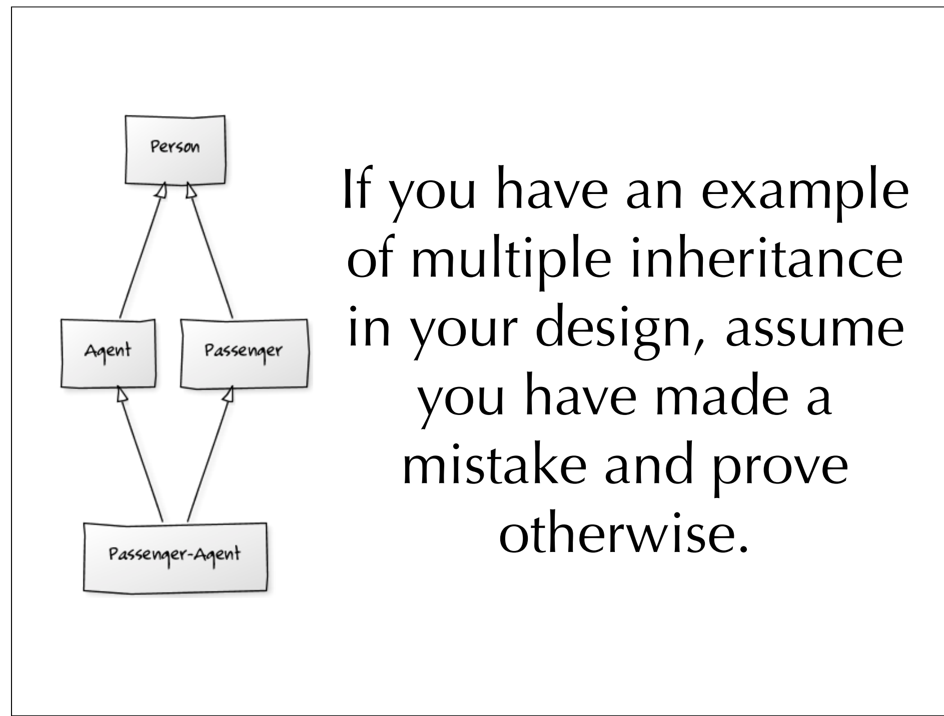
Why?

Violating this rule leads to the Fragile Base Class Problem.

I subscribe to a stricter guideline: All instance variables are private. In all classes. Period.

Why?

OO systems are designed around behavior -- the responsibilities of each object. Data are implementation detail. If one object can know anything about another object's data, then it can be designed based on the data, not around the behavior.



“Assume you have made a mistake and proved otherwise” is good advice in many design cases. Whenever you do something that is not the norm, have a good reason!

dates of interest

10/26/09 → 10/30/09

10/27/09 → 11/03/09

10/29/09 → 11/05/09

*** postponing things by a week ***

Friday: project designs are due (or: iteration 2 is due)

Tuesday: discuss designs in class ... informal presentations

Thursday: midterm exam