

# Concurrent Models of Computation for Embedded Software

Edward A. Lee

Professor, UC Berkeley

EECS 219D

*Concurrent Models of Computation*

Fall 2011

Copyright © 2009-2011, Edward A. Lee, All rights reserved

Lecture 17: Actor-Oriented Type Systems

Does Actor-Oriented Design Offer Best-Of-Class SW Engineering Methods?

## Abstraction

- procedures/methods
- classes

## Modularity

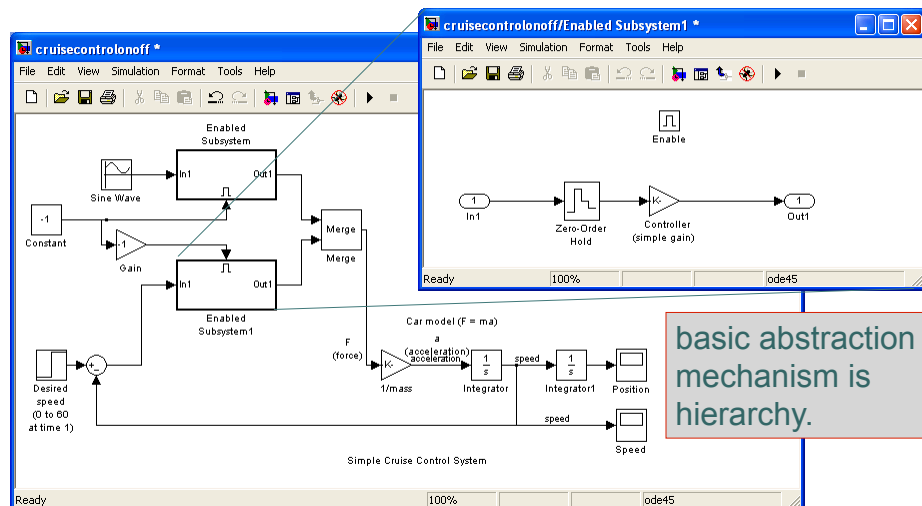
- subclasses
- inheritance
- interfaces
- polymorphism
- aspects

## Correctness

- type systems

Lee 17: 2

## Example of an Actor-Oriented Framework: Simulink



Lee 17: 3

## Observation

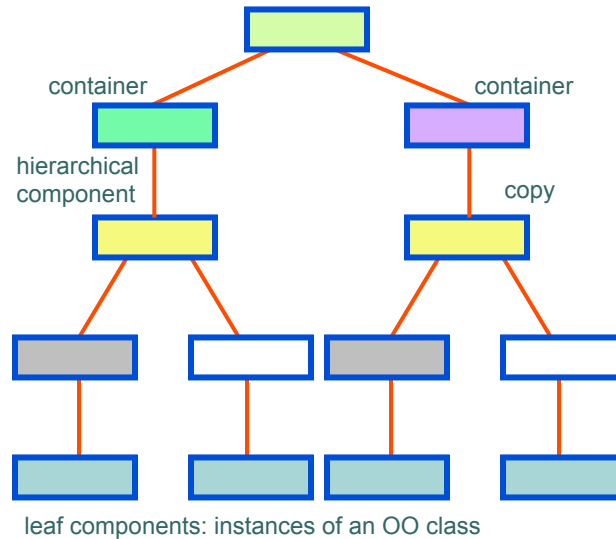
By itself, hierarchy is a very weak abstraction mechanism.

Lee 17: 4

## Tree Structured Hierarchy

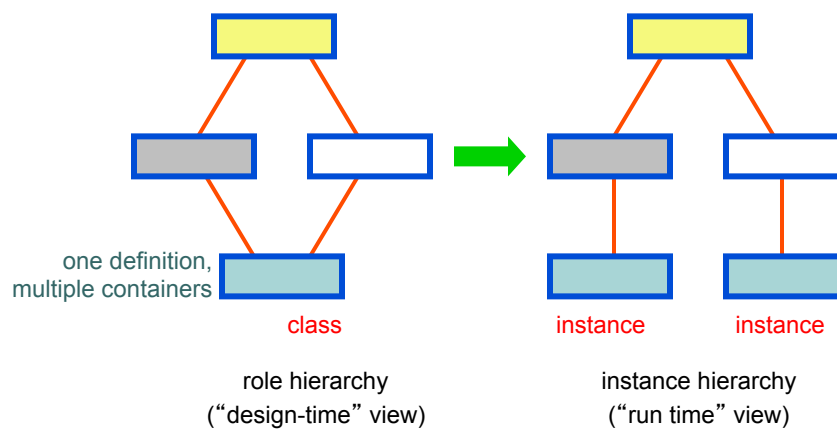
Does not represent  
common *class*  
definitions. Only  
instances.

Multiple instances  
of the same  
hierarchical  
component are  
*copies*.



Lee 17: 5

## Alternative Hierarchy: Roles and Instances

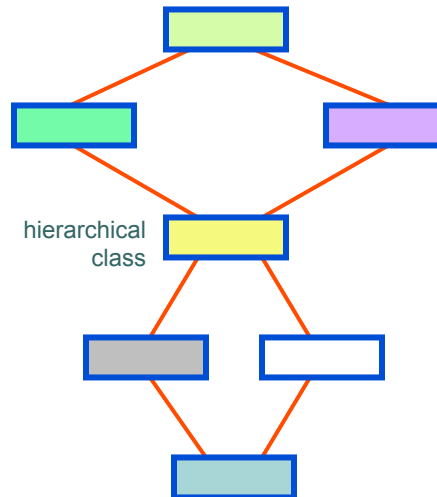


Lee 17: 6

## Role Hierarchy

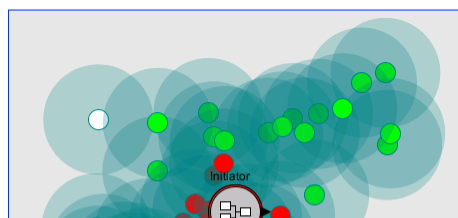
Multiple instances of the same hierarchical component are represented by *classes* with multiple containers.

This makes hierarchical components more like leaf components.



Lee 17: 7


## A Motivating Application: Modeling Sensor Networks

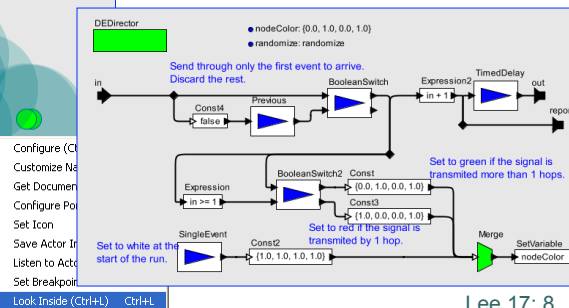


Model of Massimo Franceschetti's "small world" phenomenon with 49 sensor nodes.

These 49 sensor nodes are actors that are instances of the same class, defined as:

*Making these objects instances of a class rather than copies reduced the XML representation of the model from 1.1 Mbytes to 87 kBytes, and offered a number of other advantages.*

Channel  This channel has range given by the "range" parameter and probability of delivery given by the "probability" parameter.



Lee 17: 8

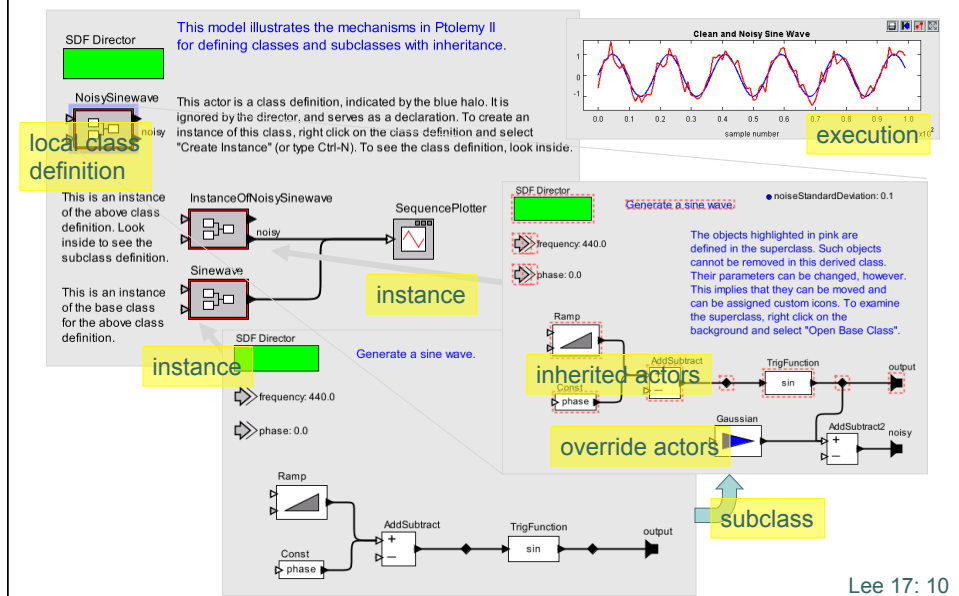
## Subclasses, Inheritance? Interfaces, Subtypes? Aspects?

Now that we have classes, can we bring in more of the  
**modern programming world?**

- subclasses?
- inheritance?
- interfaces?
- subtypes?
- aspects?

Lee 17: 9

## Example Using AO Classes



Lee 17: 10

## Inner Classes

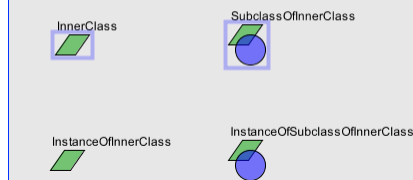
Local class definitions are important to achieving modularity.

Encapsulation implies that local class definitions can exist within class definitions.

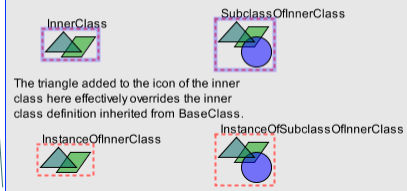
This model illustrates classes, subclasses, inner classes and inheritance, using custom icons to make it visually clear how inheritance works.

A key issue is then to define the semantics of inheritance and overrides.

The BaseClass definition includes an inner class and a subclass of that inner class, plus instances of each.

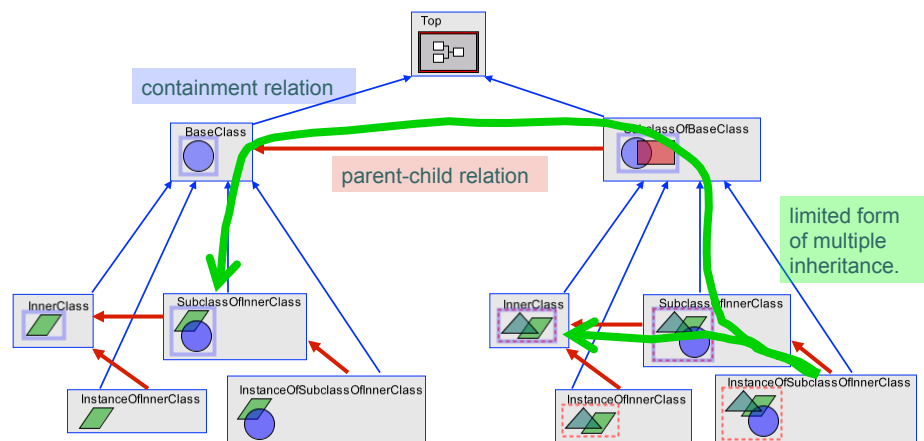


The BaseClass definition includes an inner class and a subclass of that inner class, plus instances of each.



Lee 17: 11

## Ordering Relations



Mathematically, this structure is a *doubly-nested diposet*, the formal properties of which help to define a clean inheritance semantics. The principle we follow is that *local* changes override *global* changes.

Lee 17: 12

## Formal Structure: Containment

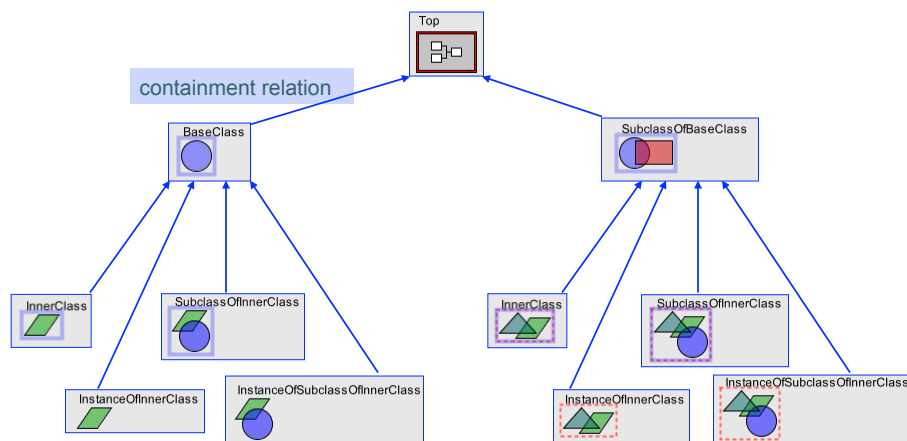
- Let  $D$  be the set of *derivable objects* (actors, composite actors, attributes, and ports).
- Let  $c: D \rightarrow D$  be a partial function (*containment*).
- Let  $c^+ \subset D \times D$  be the transitive closure of  $c$  (*deep containment*). When  $(x, y) \in c^+$  we say that  $x$  is **deeply contained by**  $y$ .
- Disallow circular containment (anti-symmetry):

$$(x, y) \in c^+ \Rightarrow (y, x) \notin c^+$$

So  $(D, c^+)$  is a strict poset.

Lee 17: 13

## Containment Relation



Lee 17: 14

## Formal Structure: Parent-Child

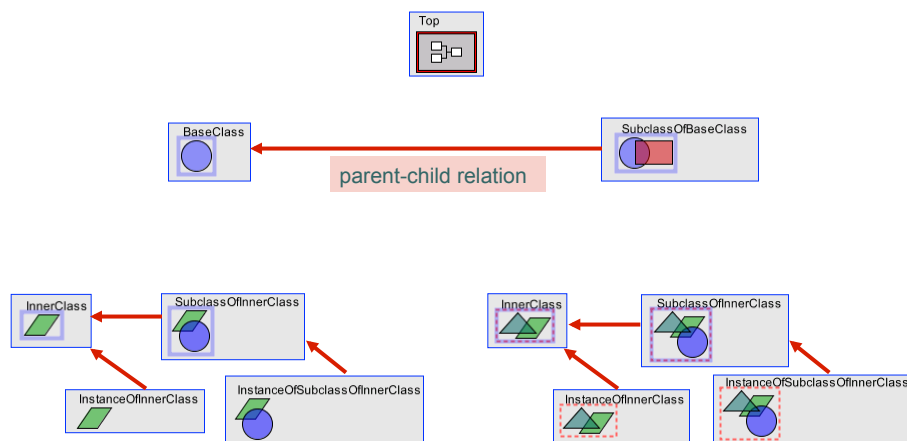
- Let  $p: D \rightarrow D$  be a partial function (*parent*).
- Interpret  $p(x) = y$  to mean  $y$  is the parent of  $x$ , meaning that either  $x$  is an instance of class  $y$  or  $x$  is a subclass of  $y$ . We say  $x$  is a child of  $y$ .
- Let  $p^+ \subset D \times D$  be the transitive closure of  $p$  (*deep containment*). When  $(x, y) \in p^+$  we say that  $x$  is descended from  $y$ .
- Disallow circular containment (anti-symmetry):  

$$(x, y) \in p^+ \Rightarrow (y, x) \notin p^+$$

Then  $(D, p^+)$  is a strict poset.

Lee 17: 15

## Parent-Child Relation



Lee 17: 16



## Structural Constraint

We require that

$$(x, y) \in p^+ \Rightarrow (x, y) \notin c^+ \text{ and } (y, x) \notin c^+$$

$$(x, y) \in c^+ \Rightarrow (x, y) \notin p^+ \text{ and } (y, x) \notin p^+$$

That is, if  $x$  is deeply contained by  $y$ , then it cannot be descended from  $y$ , nor can  $y$  be descended from it.

Correspondingly, if  $x$  is descended from  $y$ , then it cannot be deeply contained by  $y$ , nor can  $y$  be deeply contained by it.

This is called a *doubly nested diposet* [Davis, 2000]

Lee 17: 17

## Labeling

- Let  $L$  be a set of identifying labels.
- Let  $l: D \rightarrow L$  be a labeling function.
- Require that if  $c(x) = c(y)$  then  $l(x) \neq l(y)$ .  
(Labels within a container are unique).

Labels function like file names in a file system, and they can be appended to get “full labels” which are unique for each object within a single model (but are not unique across models).

Lee 17: 18

## Derived Relation

- Let  $d \subset D \times D$  be the least relation so that  $(x, y) \in d$  implies either that:

$$(x, y) \in p^+$$

or

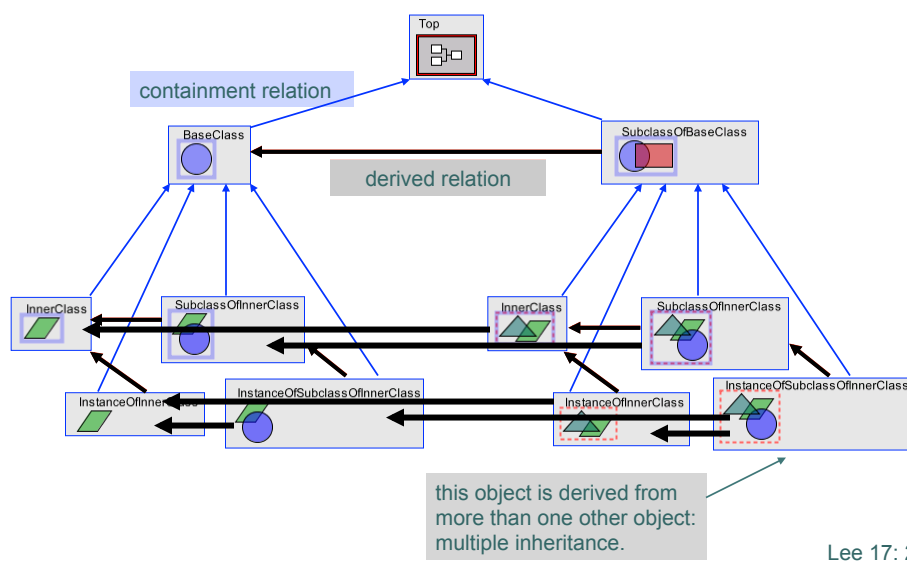
$$(c(x), c(y)) \in d \text{ and } l(x) = l(y)$$

$x$  is derived from  $y$  if either:

- $x$  is descended from  $y$  or
- $x$  and  $y$  have the same label and the container of  $x$  is derived from the container of  $y$ .

Lee 17: 19

## Derived Relation



Lee 17: 20

## Implied Objects and the Derivation Invariant

We say that  $y$  is implied by  $z$  in  $D$  if

$$(y, z) \in d \text{ and } (y, z) \notin p^+.$$

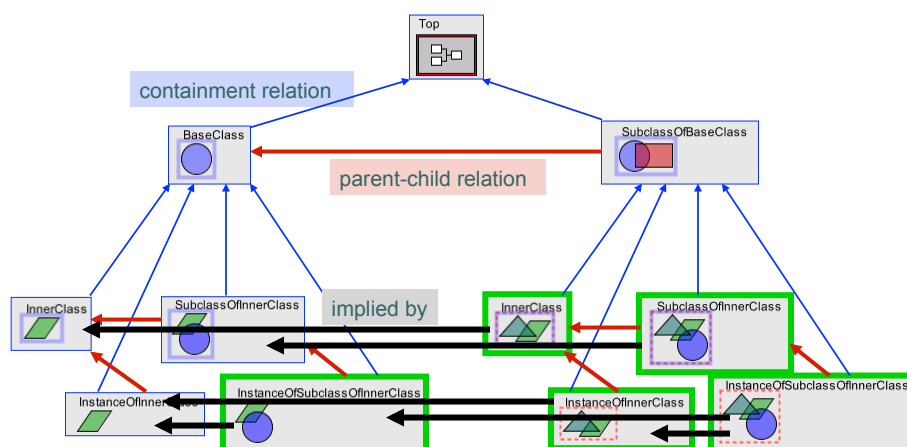
I.e.,  $y$  is implied by  $z$  if it is derived but is not a descendant.

Consequences:

- There is no need to represent implied objects in a persistent representation of the model, unless they somehow *override* the object from which they are derived.

Lee 17: 21

## Implied Objects



Lee 17: 22

## Derivation Invariant

If  $x$  is derived from  $y$  then for all  $z$  where  $c(z) = y$ , there exists a  $z'$  where  $c(z') = x$  and  $l(z) = l(z')$  and either

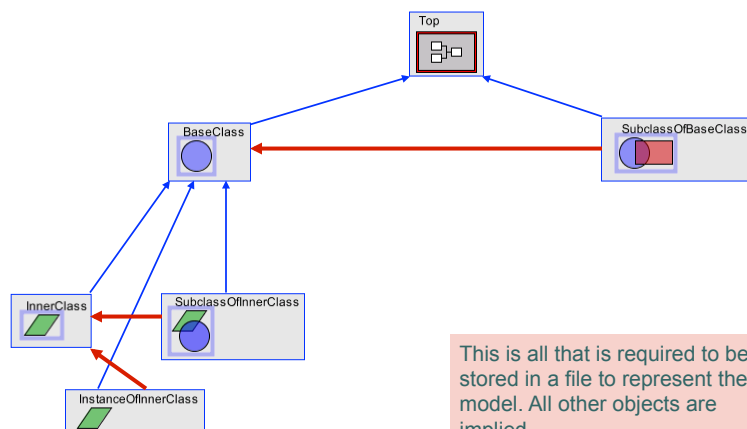
1.  $p(z)$  and  $p(z')$  are undefined, or
2.  $(p(z), p(z')) \in d$ , or
3.  $p(z) = p(z')$  and both  $(p(z), y) \notin c^+$  and  $(p(z'), x) \notin c^+$

I.e.  $z'$  is implied by  $z$ , and it is required that either

1.  $z'$  and  $z$  have no parents
2. the parent of  $z$  is derived from the parent of  $z'$  or
3.  $z'$  and  $z$  have the same parent, not contained by  $x$  or  $y$

Lee 17: 23

## Persistent Representation



Lee 17: 24

## Values and Overrides

- Derived objects can contain more than the objects from which they derive (but not less).
- Derived objects can override their *value*.
- Since there may be multiple derivation chains from one object to an object derived from it, there are multiple ways to specify the value of the derived object.
- A reasonable policy is that more local overrides supercede less local overrides. Ensuring this is far from simple (but it is doable! see paper and/or Ptolemy II code).

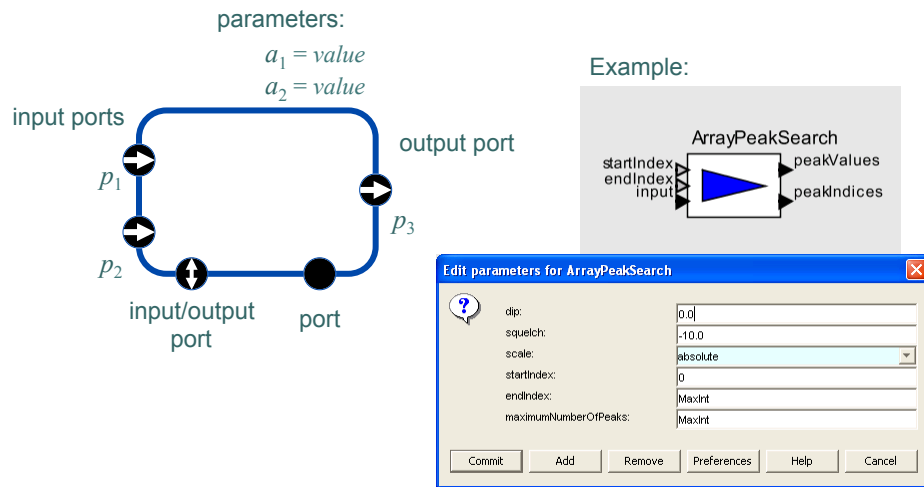
Lee 17: 25

## Advanced Topics

- Interfaces and interface refinement
- Types, subtypes, and component composition
- Abstract actors
- Aspects
- Recursive containment

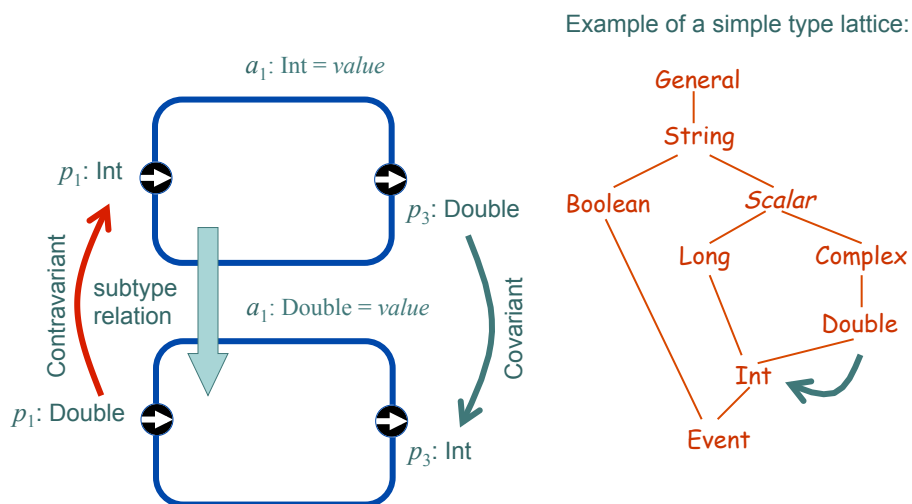
Lee 17: 26

## Defining Actor Interfaces: Ports and Parameters



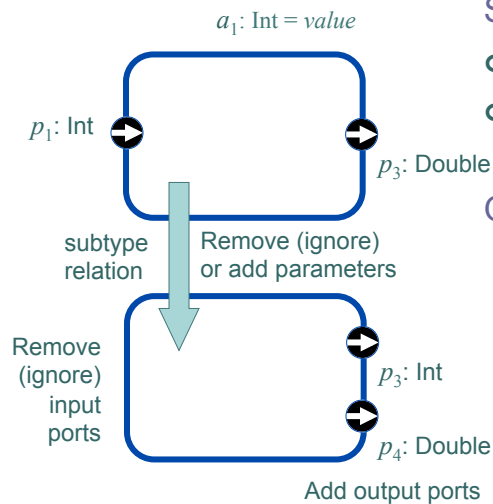
Lee 17: 27

## Actor Subtypes



Lee 17: 28

## Actor Subtypes (cont)



Subtypes can have:

- Fewer input ports
- More output ports

Of course, the types of these can have co/contravariant relationships with the supertype.

Lee 17: 29

## Observations

- Subtypes can remove (or ignore) parameters and also add new parameters because parameters always have a default value (unlike inputs, which a subtype cannot add)
- Subtypes cannot modify the types of parameters (unlike ports). Co/contravariant at the same time.
- PortParameters are ports with default values. They can be removed or added just like parameters because they provide default values.

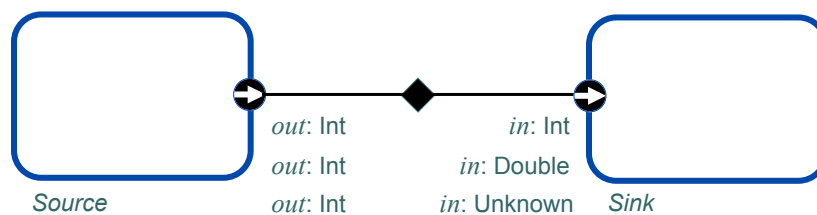
Are there similar exceptions to co/contravariance in OO languages?

Lee 17: 30

## Composing Actors

A connection implies a type constraint. Can:

check compatibility  
perform conversions  
infer types

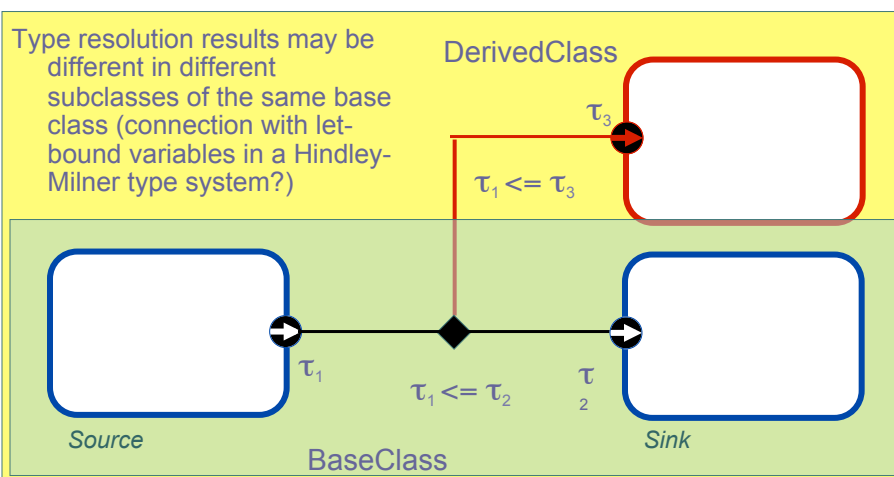


The Ptolemy II type system does all three.

Lee 17: 31

## What Happens to Type Constraints When a Subclass Adds Connections?

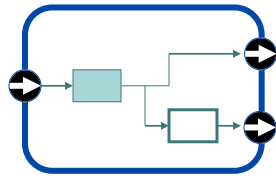
Type resolution results may be different in different subclasses of the same base class (connection with let-bound variables in a Hindley-Milner type system?)



Lee 17: 32



## Abstract Actors?



Suppose one of the contained actors is an interface only. Such a class definition cannot be instantiated (it is abstract). Concrete subclasses would have to provide implementations for the interface.

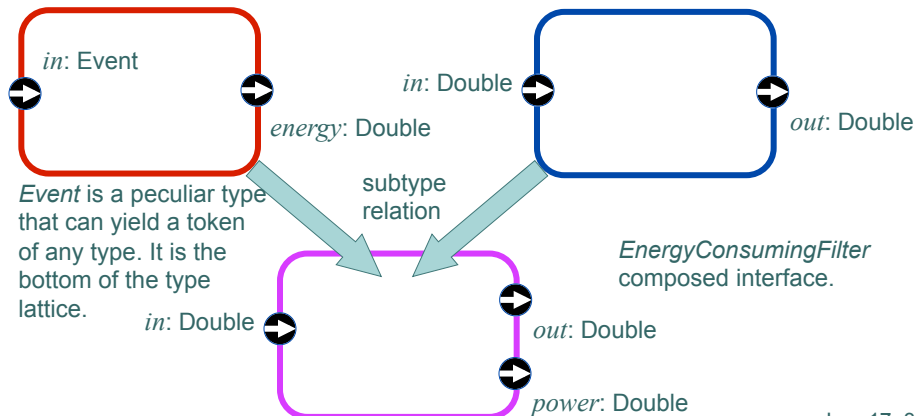
Is this useful?

Lee 17: 33

## Implementing Multiple Interfaces An Example

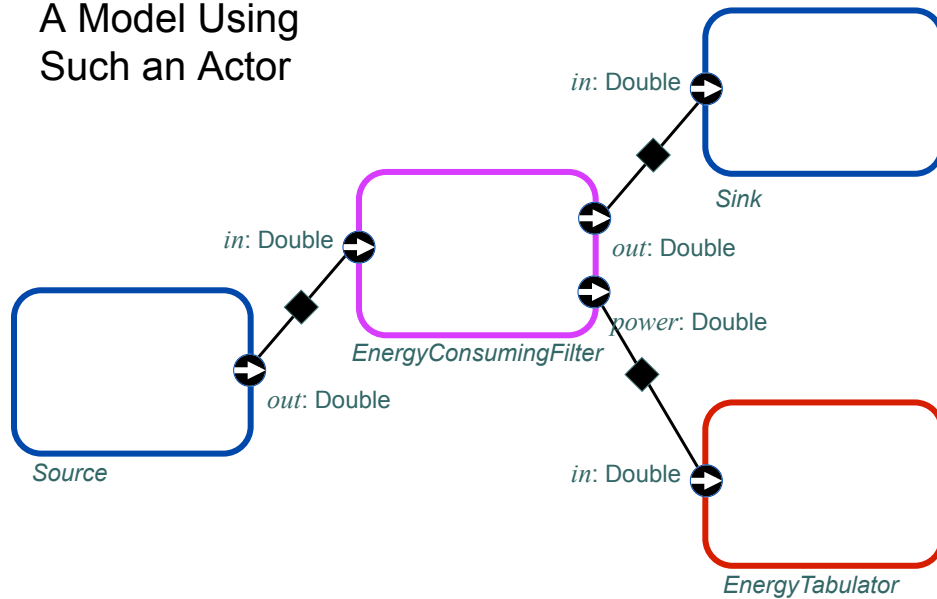
*EnergyConsumer* interface has a single output port that produces a Double representing the energy consumed by a firing.

*Filter* interface for a stream transformer component.



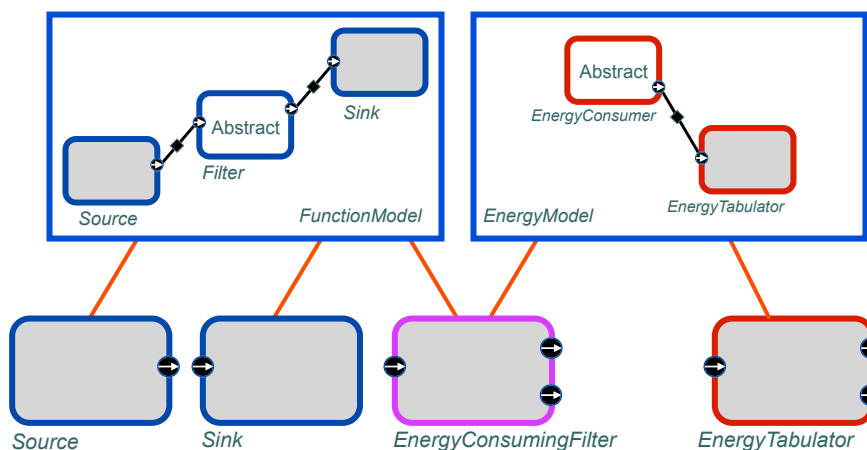
Lee 17: 34

## A Model Using Such an Actor



Lee 17: 35

## Heterarchy? Multi-View Modeling? Aspects?



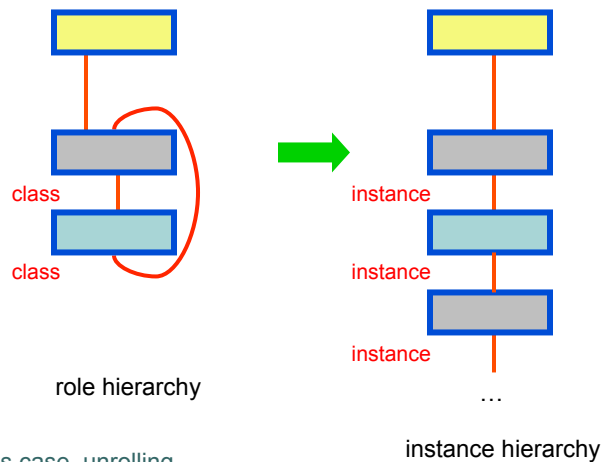
This is *multi-view modeling*, similar to what GME (Vanderbilt) can do.

Is this an *actor-oriented* version of *aspect-oriented* programming?

Is this what Metropolis does with function/architecture models?

Lee 17: 36

## Recursive Containment Can Hierarchical Classes Contain Instances of Themselves?

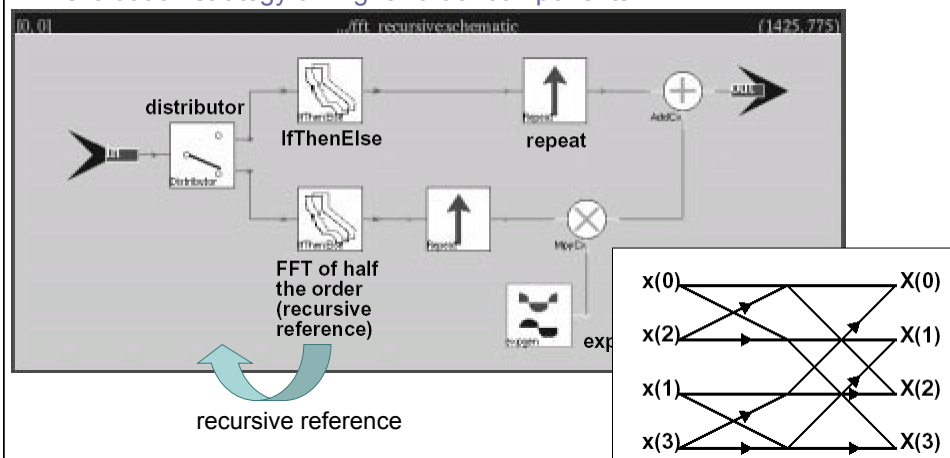


Note that in this case, unrolling cannot occur at “compile time”.

Lee 17: 37

## Primitive Realization of this in Ptolemy Classic

FFT implementation in Ptolemy Classic (1995) used a partial evaluation strategy on higher-order components.



## Conclusion

- Actor-oriented design remains a relatively immature area, but one that is progressing rapidly.
- It has huge potential.
- Many questions remain...

Lee 17: 39