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

Abstraction
- procedures/methods
- classes

Modularity
- subclasses
- inheritance
- interfaces
- polymorphism
- aspects

Correctness
- type systems
Example of an Actor-Oriented Framework: Simulink

Observation

By itself, hierarchy is a very weak abstraction mechanism.
Tree Structured Hierarchy

Does not represent common class definitions. Only instances.

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

Alternative Hierarchy: Roles and Instances

One definition, multiple containers

role hierarchy
("design-time" view)

instance hierarchy
("run time" view)
Role Hierarchy

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

This makes hierarchical components more like leaf components.

A Motivating Application: Modeling Sensor Networks

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.
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?

Example Using AO Classes

This model illustrates the mechanisms in Ptolemy II for defining classes and subclasses with inheritance.

This is an instance of the above class definition. Look inside to see the subclass definition.

This is an instance of the base class for the above class definition.

This is an instance of NoiseySinewave.
Inner Classes

Local class definitions are important to achieving modularity.
Encapsulation implies that local class definitions can exist within class definitions.

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.

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

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.
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 $c^+ \subseteq D \times D$ 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.
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 $p^+ \subseteq D \times D$ 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):
  
  $$(x, y) \in p^+ \Rightarrow (y, x) \not\in p^+$$

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

Labeling

- Let \(L\) be a set of identifying labels.
- Let \(l: D \to L\) 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).
Derived Relation

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

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

or

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

$x$ 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

containment relation

parent-child relation

this object is derived from more than one other object: multiple inheritance.

Lee 17: 20
Implied Objects and the Derivation Invariant

We say that \( y \) is implied by \( z \) in \( D \) if
\[
(y, z) \in d \quad \text{and} \quad (y, z) \notin p^+.
\]

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.
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 $y$

Persistent Representation

This is all that is required to be stored in a file to represent the model. All other objects are implied.
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).

Advanced Topics

- Interfaces and interface refinement
- Types, subtypes, and component composition
- Abstract actors
- Aspects
- Recursive containment
Defining Actor Interfaces:
Ports and Parameters

parameters:
\[ a_1 = \text{value} \]
\[ a_2 = \text{value} \]

Example:

Actor Subtypes

Example of a simple type lattice:
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.

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?
Composing Actors

A connection implies a type constraint. Can:

- check compatibility
- perform conversions
- infer types

The Ptolemy II type system does all three.

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: 31

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?

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.

*Event* is a peculiar type that can yield a token of any type. It is the bottom of the type lattice.
A Model Using Such an Actor

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?
Recursive Containment
Can Hierarchical Classes Contain Instances of Themselves?

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

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…