Ten Years of Ownership Types,

or

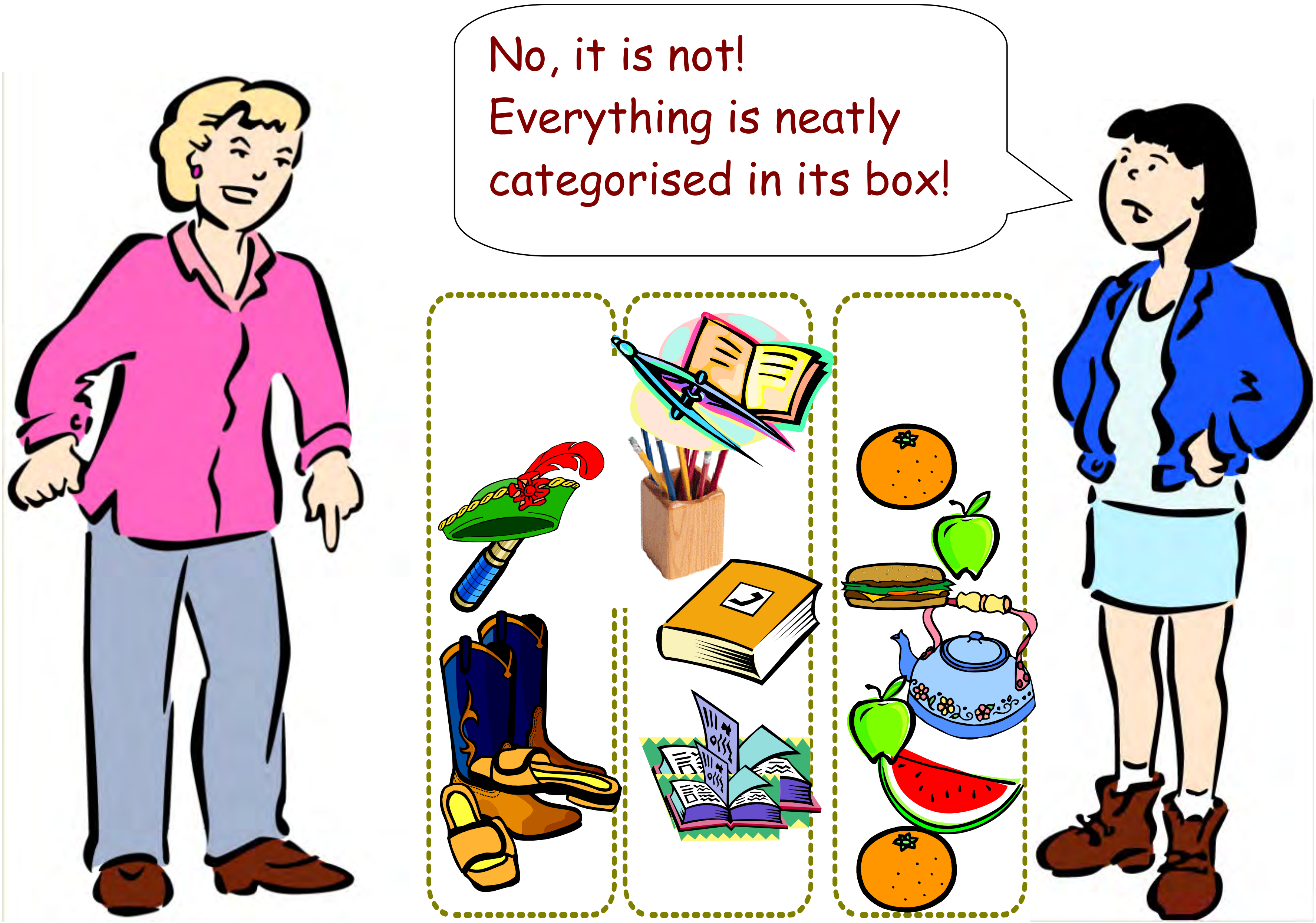
the benefits of Putting Objects into Boxes

Sophia Drossopoulou
Department of Computing, Imperial College London

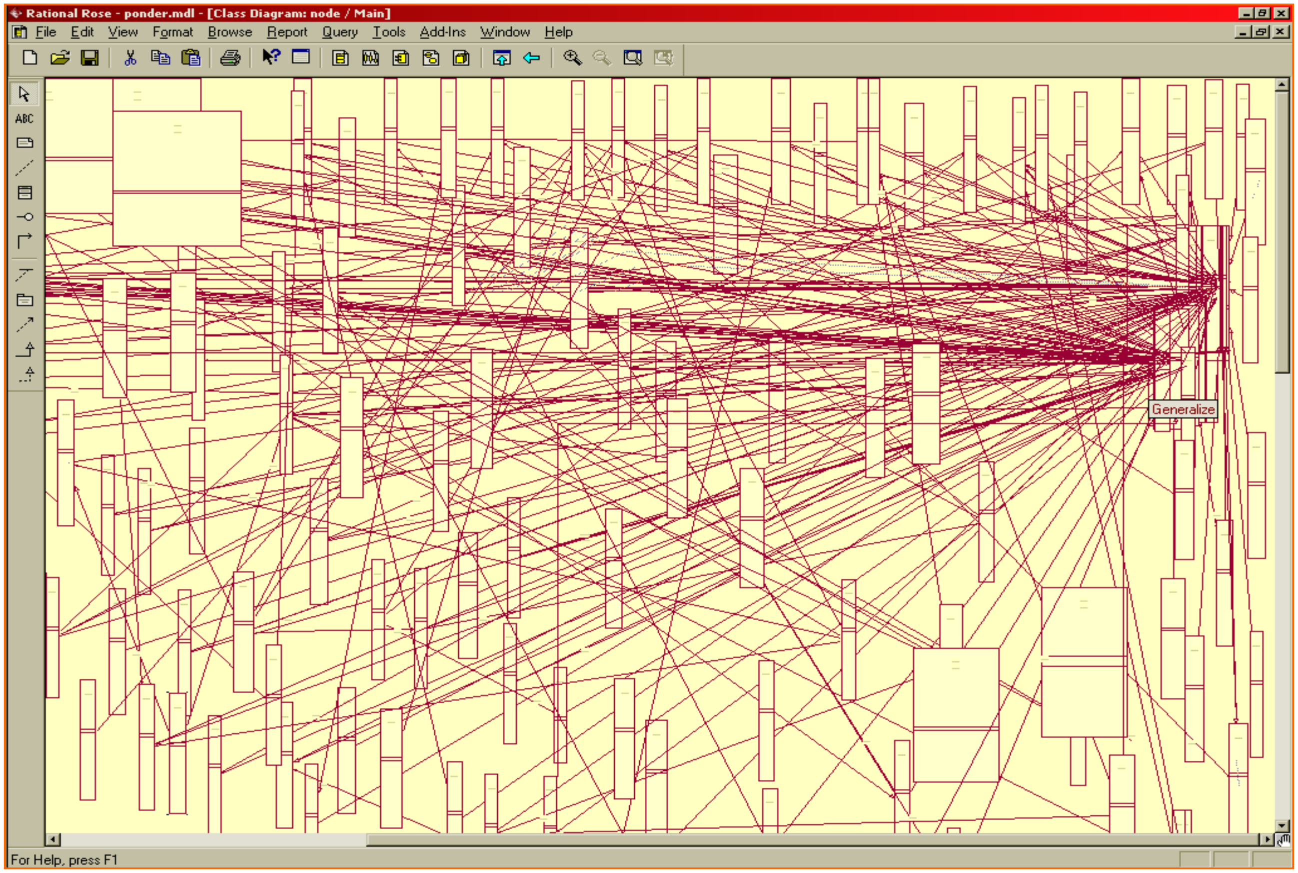
We would like our surroundings*
to be "tidy"

*surroundings = home, or desk, or code, or program heap, ...


This room
is a mess!


No, it is not!
Everything is neatly
categorised in its box!

A common problem in programming



is that code /object topology is far too complex.

A common solution is to organize code/objects into “boxes”.

Over the last decade, several kinds of “boxes” have been suggested with different aims.

Some of this work has concentrated on static type systems.

We shall discuss:

- Survey some of the work on boxes
- The associated heap topology
- MOJO: the need for multiple boxes

Ten years of ownership types …

Research by J. Noble, J. Vitek, J. Potter, D. Clarke, B. Liskov, M. Rinard, C. Boyapati, P. Mueller, J. Aldrich, C. Chambers, M. Schwarzbach, A. Poetzsch-Heffter, J Palsberg, A. Milanova, A. Routlev, P. S. Almeida, B. Bokowski, J. Boyland, S. Drossopoulou, W. Dietl, R. Leino, T. Wrigstad, Y. Lu, T. Zhao, A. Potanin, A. Rudich, , J. Schäfer, M. Smith, A. Wren, A. Buckley, Y. Lu, D, Naumann, A. Bannerjee

… and many, many others.

… it started, ten years ago, at ECOOP 1998

with a paper without implementation, without semantics

Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are hard to understand, maintain, and analyse. *Flexible alias protection* is a conceptual model of inter-object relationships which limits the visibility of changes via aliases, allowing objects to be aliased but mitigating the undesirable effects of aliasing. Flexible alias protection can be checked statically using programmer supplied *aliasing modes* and imposes no run-time overhead. Using flexible alias protection, programs can incorporate mutable objects, immutable values, and updatable collections of shared objects, in a natural object oriented programming style, while avoiding the problems caused by aliasing.

1 Introduction

I am who I am; I will be who I will be.

Object identity is the foundation of object oriented programming. Objects are useful for modelling application domain abstractions precisely because an object's identity always remains the same during the execution of a program —

… it started, ten years ago, at ECOOP 1998

with a paper without implementation, without semantics, but with **very compelling** diagrams

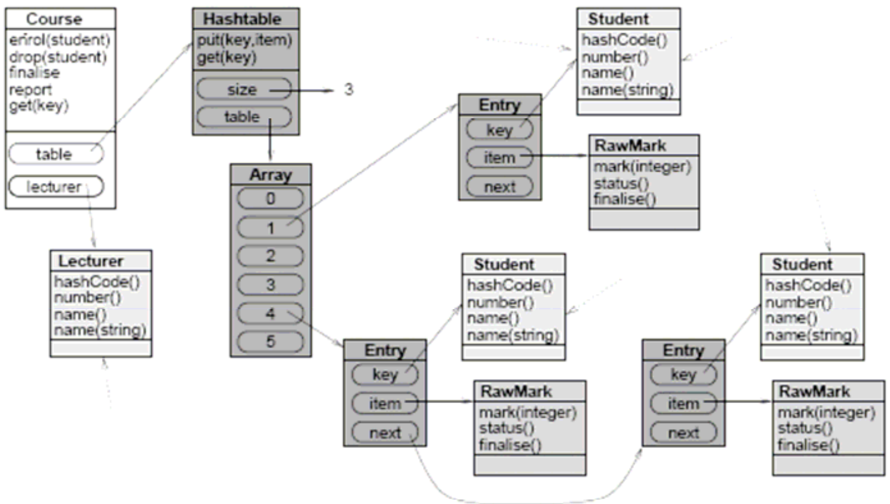


Fig. 4. A Course uses a hashtable as part of its representation (dark grey) while Student and Lecturer objects are the course's arguments (light grey). The hashtable also stores RawMark objects for each student, and these are arguments to the hashtable but part of the Course's representation (mid gray), so cannot be accessed from outside the Course (dotted arrows).

tainer objects. Flexible alias encapsulation separates the objects within an ag-

… and then, at OOPSLA 1998

Ownership Types for Flexible Alias Protection

David G. Clarke, John M. Potter, James Noble

Microsoft Research Institute, Macquarie University, Sydney, Australia
{clad,potter,kjx}@mri.mq.edu.au

Abstract

Object-oriented programming languages allow inter-object

model of flexible alias protection, supported by illustrative examples, and suggested incorporating aliasing modes into programming languages. For flexible alias protection three

… a paper with a formal system

(Red Field Update)

$$\frac{\mathcal{S} \cdot \Delta \vdash e \leadsto o / \mathcal{S}^1 \cdot \Delta^1 \quad \mathcal{S}^1 \cdot \Delta^1 \vdash e' \leadsto v' / \mathcal{S}^2 \cdot \Delta^2}{\mathcal{S} \cdot \Delta \vdash e.fd = e' \leadsto v' / \mathcal{S}^2[o \mapsto \mathcal{F}[fd \mapsto v']] \cdot \Delta^2}$$

where $S^2(o) = \mathcal{F}$

(Red Sequence)

$$\frac{\mathcal{S} \cdot \Delta \vdash e_1 \leadsto v_1 / \mathcal{S}^1 \cdot \Delta^1 \quad \mathcal{S}^1 \cdot \Delta^1 \vdash e_2 \leadsto v_2 / \mathcal{S}^2 \cdot \Delta^2}{\mathcal{S} \cdot \Delta \vdash e_1; e_2 \leadsto v_2 / \mathcal{S}^2 \cdot \Delta^2}$$

D. Clarke and A. Buckley then developed implementations …

Survey - 1

Boxes for Package Encapsulation

Bokowski, Vitek, Grothof, Palsberg,...

Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore

- "box" is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore

- “box” is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

```
package P1 {
   class A{ ... }
   class B{ ... }
   confined class C{ ... }
 }
package P2 {
   class D{ ... }
   confined class E{ ... }
 }
```

Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

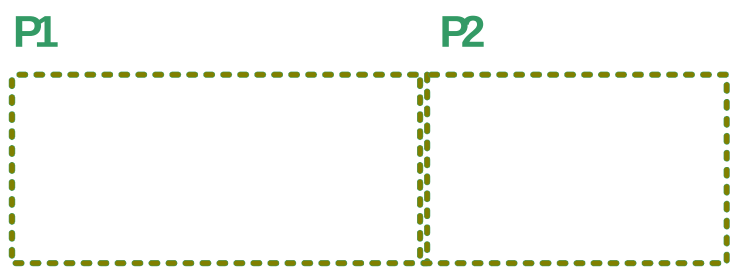
Therefore

- "box" is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

```
package P1 {
   class A{ ... }
   class B{ ... }
   confined class C{ ... }
 }
package P2 {
   class D{ ... }
   confined class E{ ... }
 }
```

with a possible heap:



Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

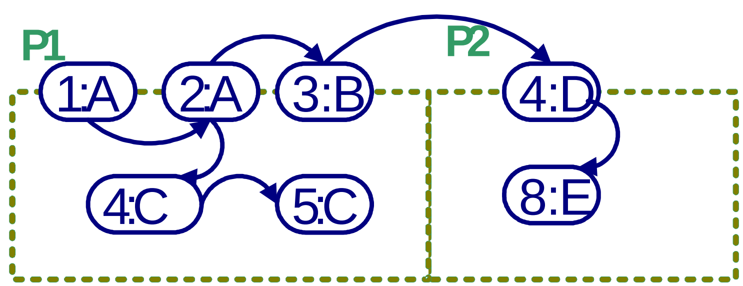
Therefore

- "box" is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

```
package P1 {
   class A{ ... }
   class B{ ... }
   confined class C{ ... }
 }
package P2 {
   class D{ ... }
   confined class E{ ... }
 }
```

with a possible heap:



Boxes for Package Encapsulation

- some classes declared confined within their package
- objects of confined type encapsulated within package

Therefore

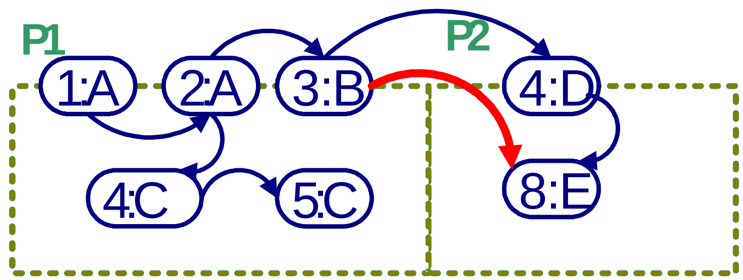
- "box" is a package; static boxes
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

Code from one package won't run on confined objects from another.

```
package P1 {
   class A{ ... }
   class B{ ... }
   confined class C{ ... }
 }
package P2 {
   class D{ ... }
   confined class E{ ... }
 }
```

with a possible heap:



Survey - 2

Boxes for Object Encapsulation

Aldrich, Biddle, Boyapati, Chambers, Clarke, Drossopoulou, Khrishnaswami, Kostadinov, Liskov, Lu, Noble, Potanin, Potter, Vitek, Shrira, Wrigstad, ...

Boxes for Object Encapsulation

– Clarke, Noble, Potter, Vitek,..

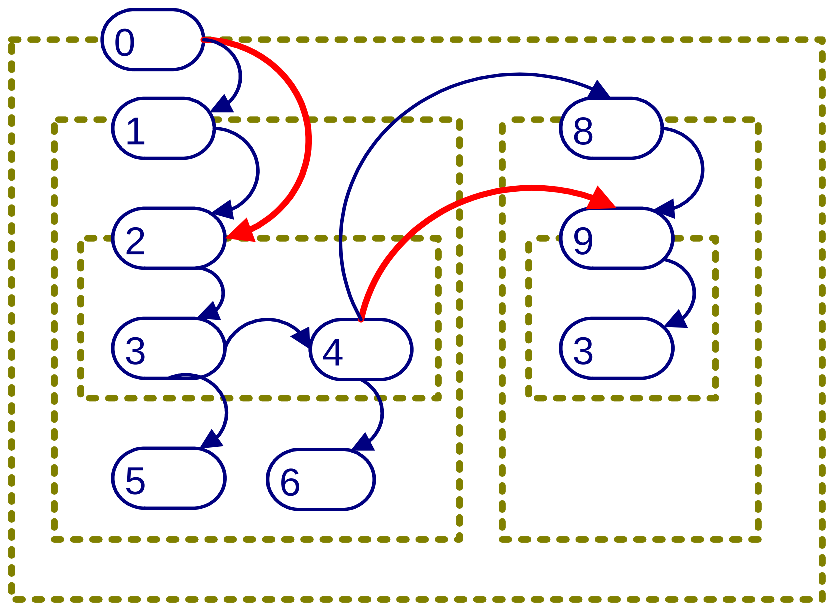
- each object belongs in a box;
- each box is characterized by an object (its owner)
- objects may hold references to objects in enclosing boxes

Therefore

- tree hierarchy of objects
- **owner as dominator**: no incoming references to a box

Properties guaranteed statically

a possible heap:



Boxes for Object Encapsulation – An Example

An employee is responsible for a sequence of tasks. Each task has a duration and a due date.

When an employee is delayed, each of his tasks gets delayed accordingly.

An employee is OK, if all his tasks are within the due dates.

"Java" code

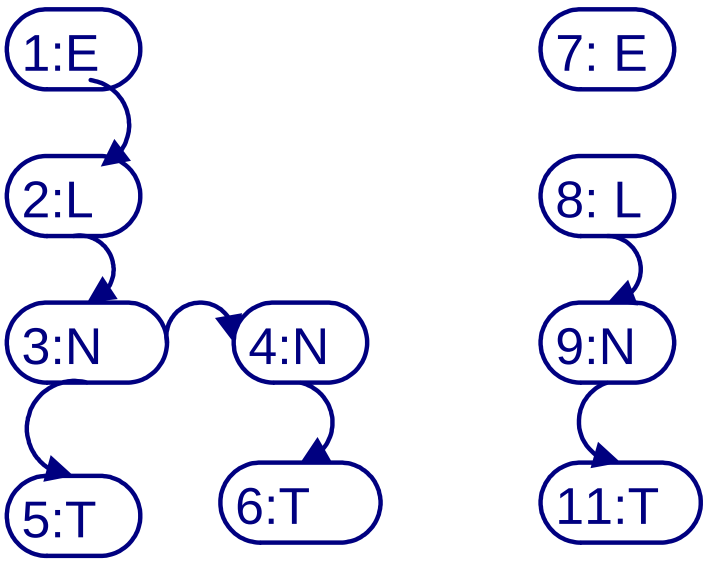
```
class Employee {
   List  tasks;
   void delay( ){ ... }
 }
 class List {
    Node first;
    void delay(){ ... }
 }
 class Node {
    Node next;
    Task task;
    void delay(){ ... }
 }
 class Task {  ...
    void delay(){ ... } }
```

Boxes for Object Encapsulation – An Example

"Java" code

```
class Employee {
   List  tasks;
   void delay( ){ ... }
 }
 class List {
    Node first;
    void delay(){ ... }
 }
 class Node {
    Node next;
    Task task;
    void delay(){ ... }
 }
 class Task {  ...
void delay(){ ... } }
```

possible heap

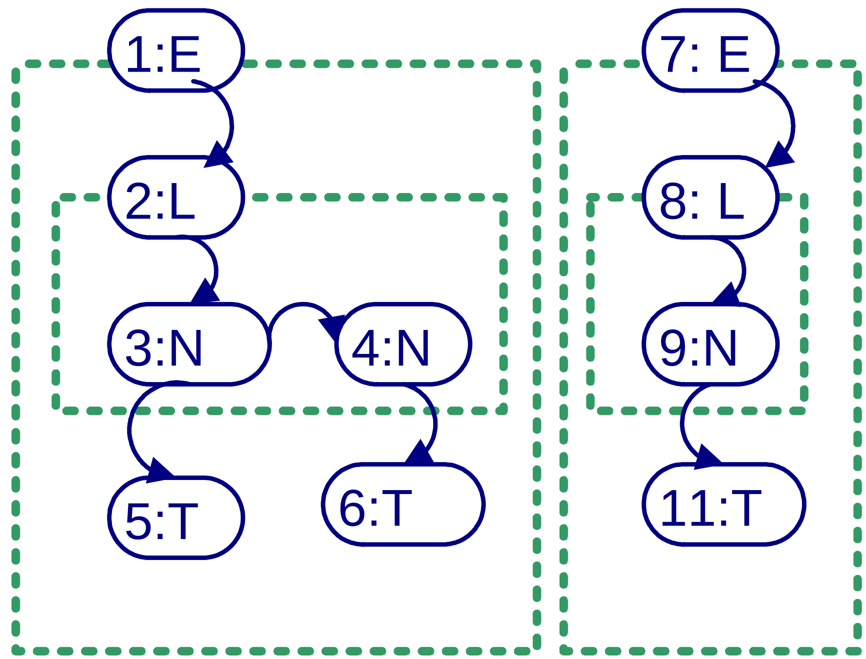


Boxes for Object Encapsulation – An Example

Employee “owns” his tasks, and the list.

The list “owns” its nodes.

with a possible heap:



Boxes for Object Encapsulation – An Example

Each object owned by another, eg 1 owns 2, 5, 6. Thus, classes have owner parameter, eg

```
class List<o>{ ... }
```

and types mention owners, eg

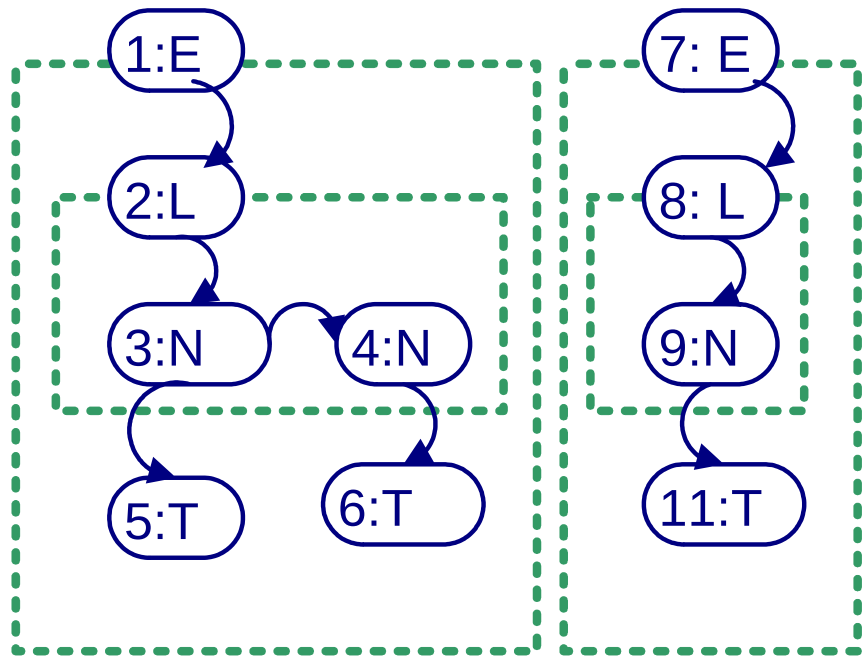
```
List<this>
```

Objects may have fields pointing to enclosing boxes, eg 3.

Classes have as many ownership parameters, as boxes involved

```
class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task;.. }
```

with a possible heap:

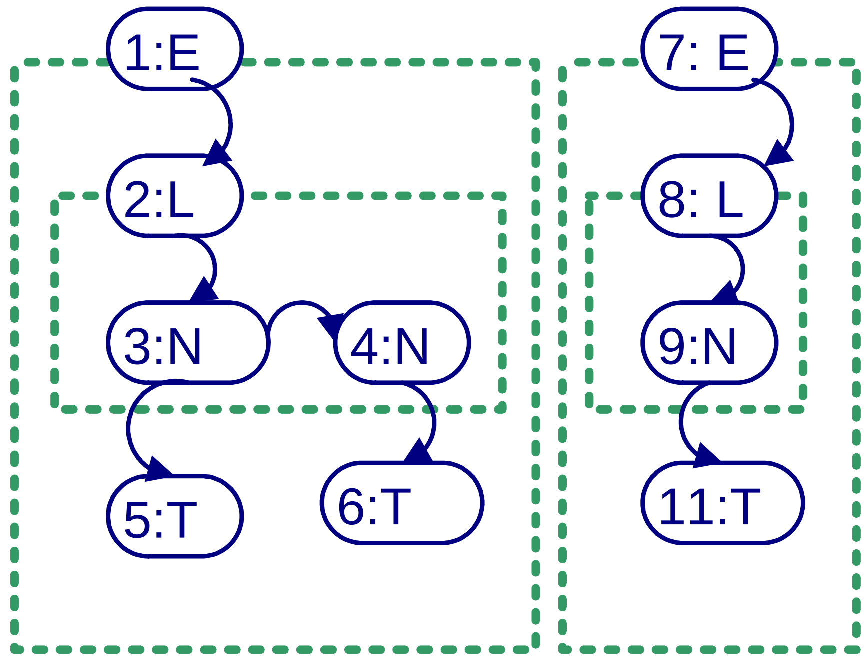


Boxes for Object Encapsulation – An Example

"Java + OT" code

```
class Employee<o> {
   List<this> tasks;
   void delay( ){ ... }
 }
 class List<o1>{
    Node<this,o1> first;
    void delay(){ ... }
 }
 class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task;
    void delay(){ ... }
 }
 class Task<o>{    ...
    void delay(){ ... } }
```

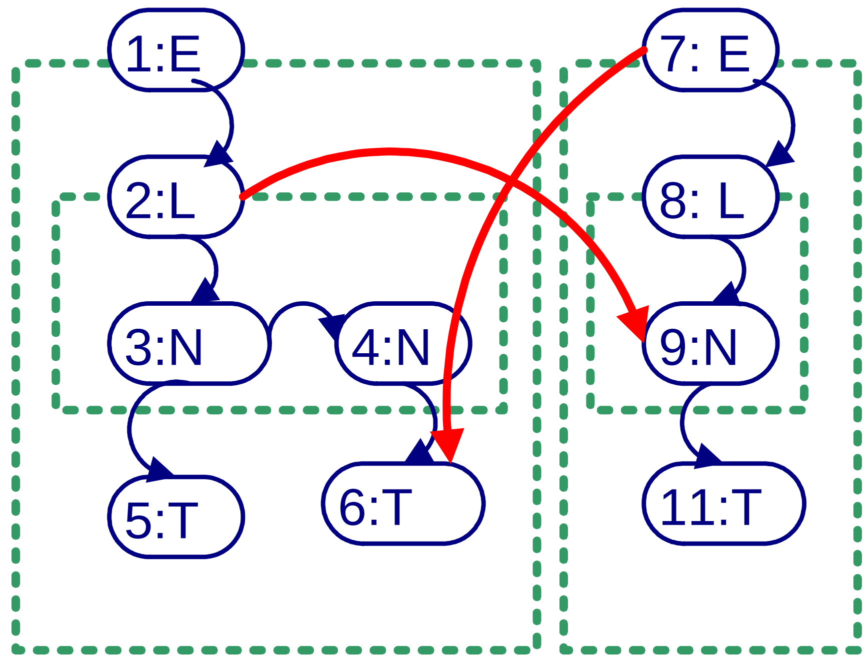
with a possible heap:



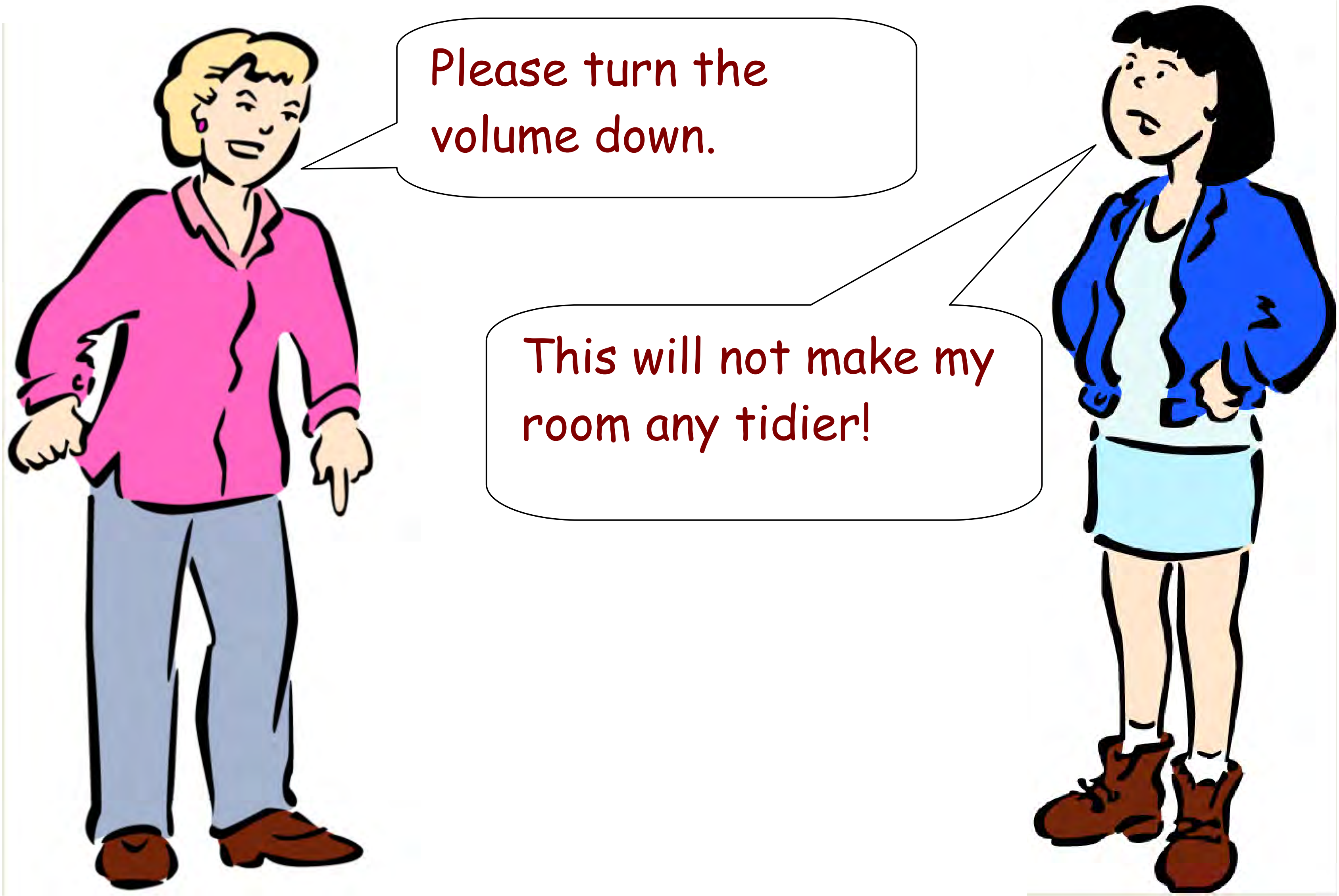
Boxes for Object Encapsulation – An Example

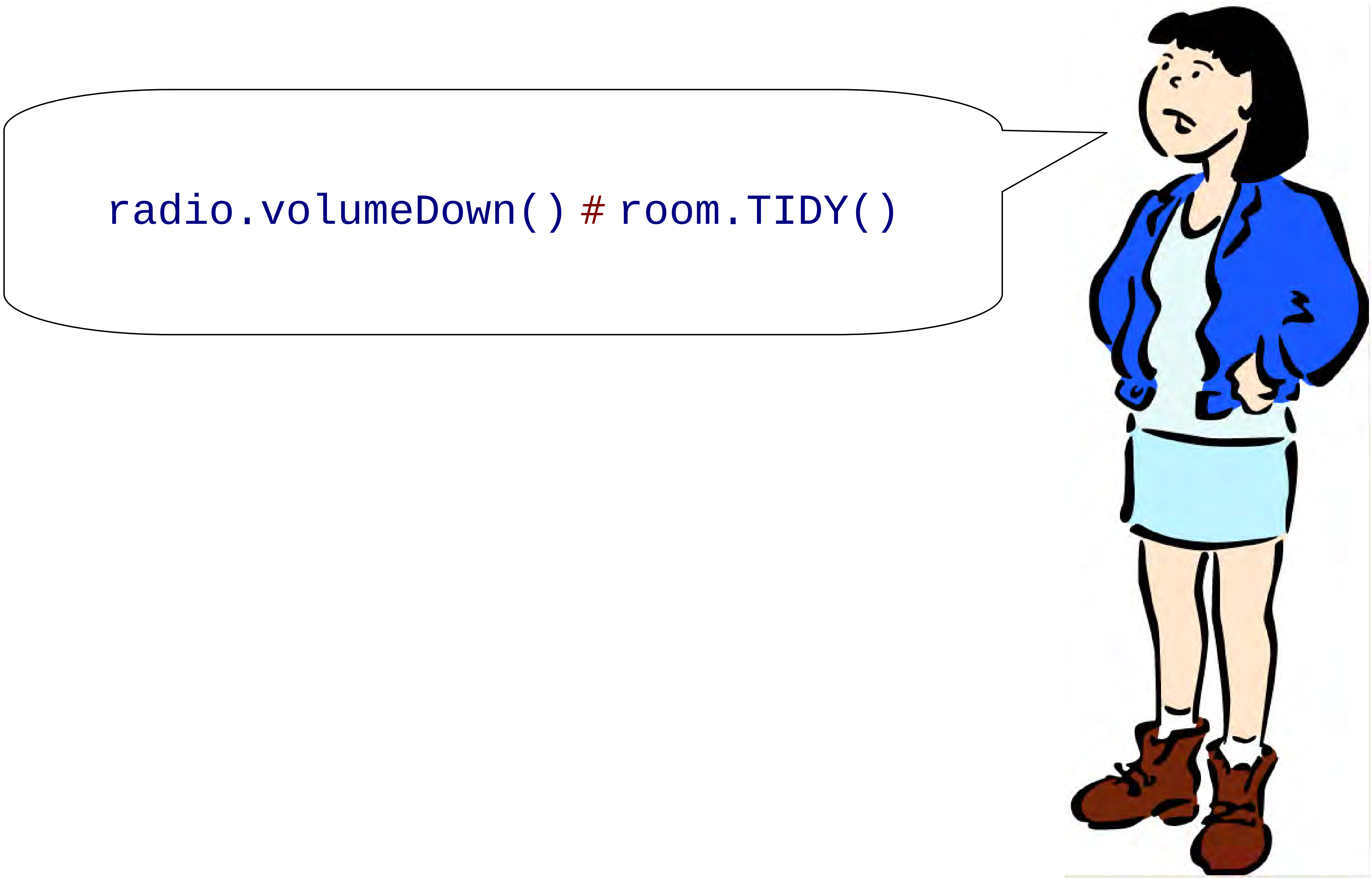
```
class Employee<o> {
   List<this> tasks;
   void delay( ){ ... }
 }
 class List<o1>{
    Node<this,o1> first;
    void delay(){ ... }
 }
 class Node<o1,o2>{
    Node<o1,o2> next;
    Task<o2> task;
    void delay(){ ... }
 }
 class Task<o>{     ...
    void delay(){ ... } }
```

with a possible heap:



Employee "controls" its tasks; list controls its links.


Please turn the volume down.
This will not make my room any tidier!


radio.volumeDown() # room.TIDY()

Boxes for Property Encapsulation

Clarke, Drossopoulou, Smith

We want to be able to argue for "different" employees `e1`, `e2`:

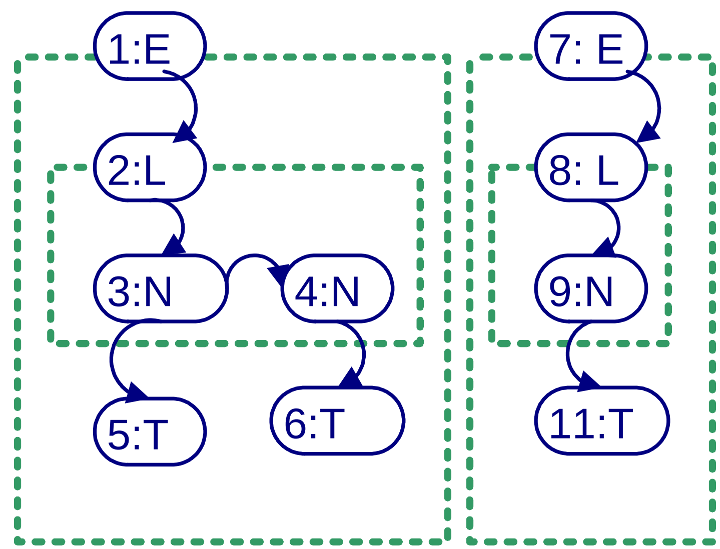
$$\texttt{e1} \nparallel \texttt{e2} \vdash \texttt{e1.delay()} \# \texttt{e2.OK()}$$

Approach: Boxes characterize the parts of heap affecting/ed by some execution/property.

For example:

```
1.delay() : 1.under
7.OK() : 7.under
```

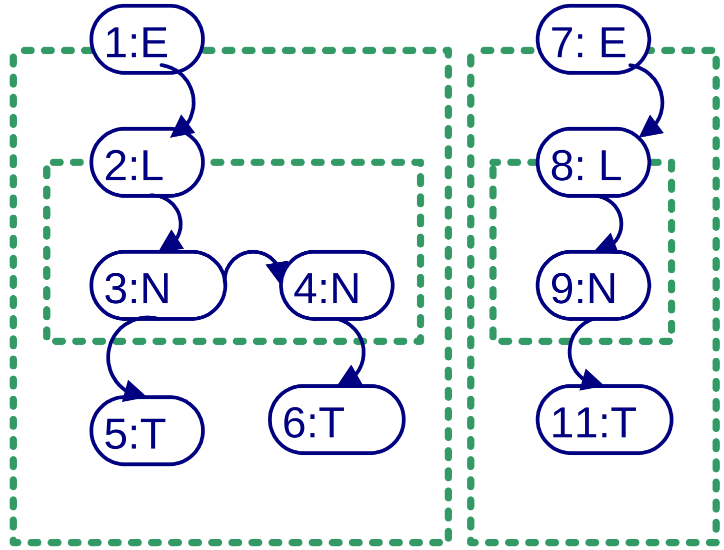
Disjoint boxes ⇒ independence



Boxes for Property Encapsulation – An Example

Approach: we add effects to methods:

```
class Employee<o> {...
   void delay( ):this.under
}
class List<o1>{...
    void delay( ):o1.under
}
class Node<o1,o2>{...
    void delay():o2.under
}
class Task<o>{  ...
    void delay():o.under
}
```



Therefore, `e1.delay()` : **`e1.under`**

`e2.OK()` : **`e2.under`**

Because $e1 \nparallel e2 \vdash$ **`e1.under`** # **`e2.under`**

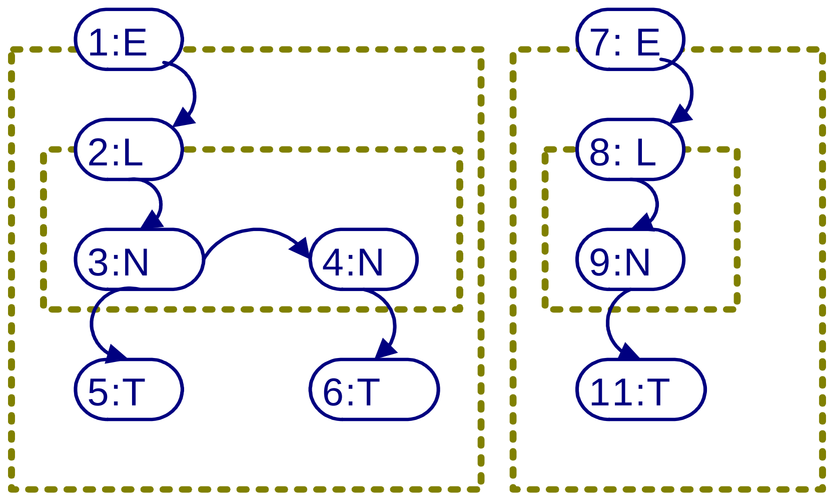
we have $e1 \nparallel e2 \vdash$ `e1.delay()` # `e2.OK()`

Boxes for Scoped Memory

Zhao, Noble, Vitek, Sacianu, Boyapati, Beebee, Rinard

Exploit owners as dominators property, to reclaim whole memory areas rather than individual objects, in presence of multithreading

Here, 2, 3, and 4 belong in one memory scope and reclaimed together. Then, 1, 5 and 6 belong to the parent memory scope.



Memory areas organized hierarchically. Threads enter/leave memory scopes consistent with the hierarchy.

Scoped memory used in unmanned airplanes (Vitek & Noble)

Survey - 3

Boxes for Concurrency

Boyapati, Lee, Liskov, Rinard, Salcianu, Shrira, Whaley, ...

and also

Abadi, Flanagan, Freund, Qadeer,

.

and also

Cunningham, Eisenbach, Drossopoulou

Boxes for Concurrency

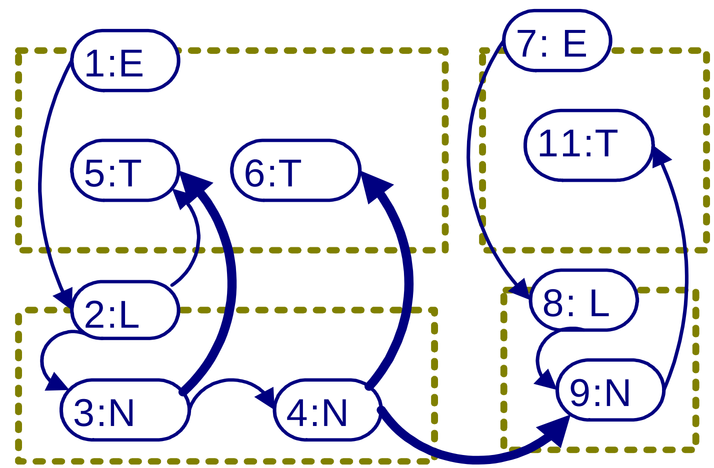
To avoid races/guarantee atomicity, a thread must have acquired the lock to an object before accessing it. The owner of a box stands for the lock of all the contained objects.

A thread must lock 1 before accessing 1, 5, or 6 – ie **no need to lock objects individusally.**

Threads must lock 2 before accessing 2, 3, or 4.

Note

- no nesting of boxes
- owners **not** dominators
- owners as locks.



Survey - 4

Boxes for Program Verification

Barnett, Bannerjee, Darvas, DeLine, Dietl, Faehndrich, Jacobs, Leavens, Leino, Logozzo, Mueller, Naumann, Parkinson, Piessens, Poetzsch-Heffter, Schulte ...

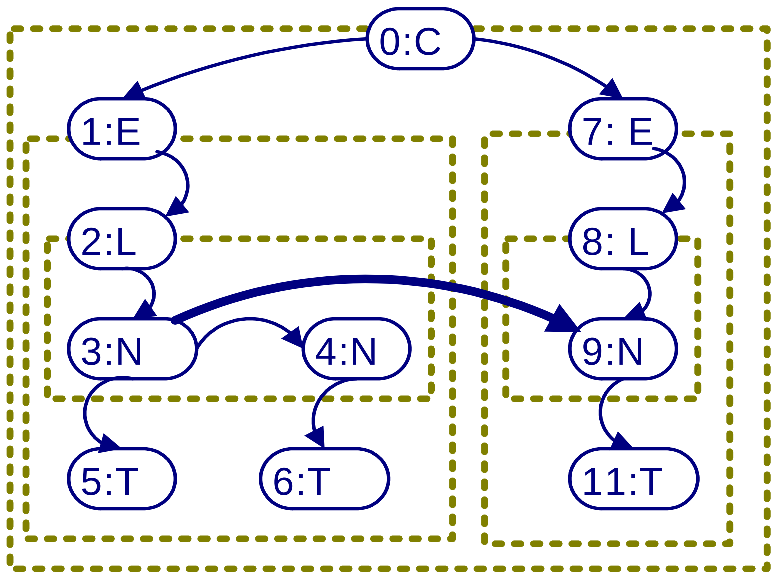
Boxes for Verification

An object "owns" other objects; the owner's invariant depends on the properties of the owned object.

A company is OK, if all its employees are OK. An employee is OK, if all his tasks are on time.

Note:

- owners may change; (5 may move to 7)
- no owners as dominators; (3 may have reference to 9)
- owner as modifier (3 may not change 9)



Survey - 5

Boxes for Program Architecture

Chambers, Aldrich, Krisnaswami, ….

Boxes as Ownership Domains

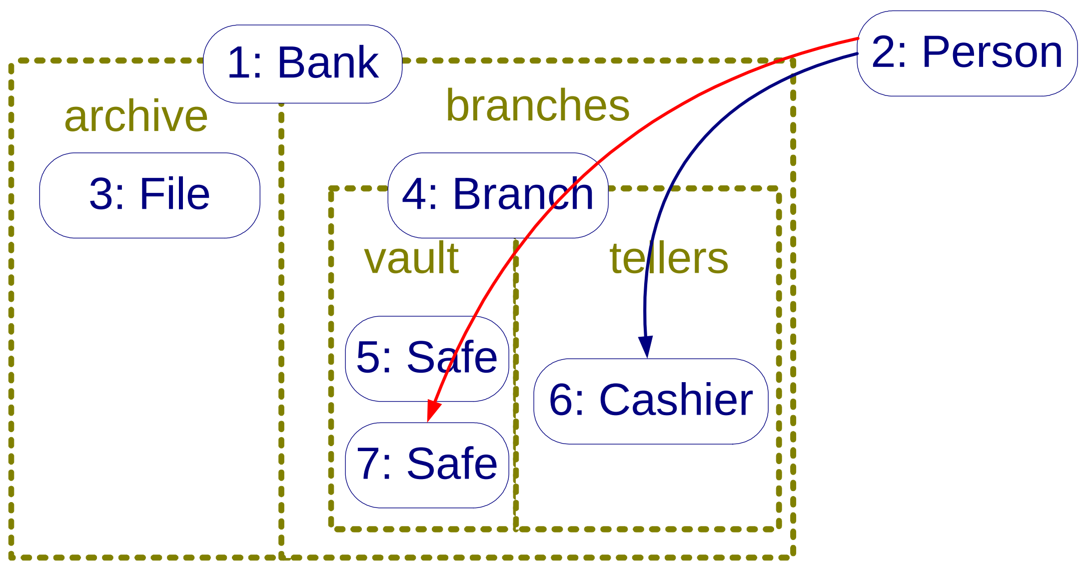
Objects "own" boxes. Link statements allow references across boxes

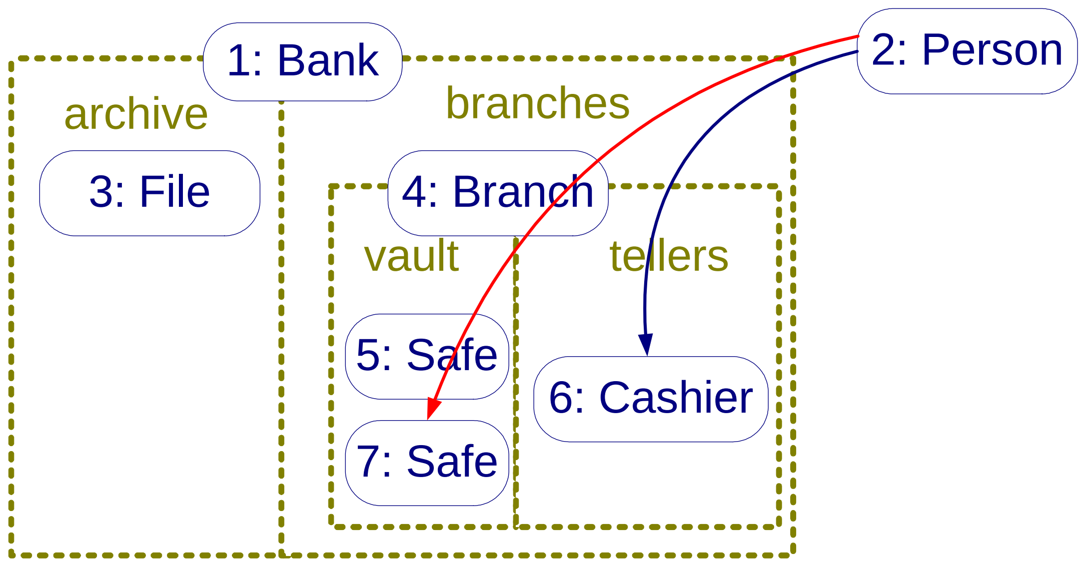
A Bank has several branches and an archive. A branch has tellers and a vault. Customers are allowed access to the tellers, but not the vaults.

```
class Bank<b> {
    domain branches, archive;
    Branch<branches> br1, br2;
    Data<archive> d1, d2, d3;
    link b.customer & branches…
 }
 class Branch<b1>{
    domain vault, tellers;
    Safe<vault> s1, s2;
    Cahier<teller> c1, c3;
    link b1 & teller;

class Main<b> {
    domain customer, shop;
    Bank<shop> b1;
    Person<customer> p1, p2 };
```

```
class Bank<b> {
   domain branches, archive;
   Branch<branches> br1, br2;
   Data<archive> d1, d2, d3;
   link b.customer & branches
}

;
```

```
class Branch<b1>{
   domain vault, tellers;
   Safe<vault> s1, s2;
   Cashier<teller> c1,c2;  link
b1 & teller;
```





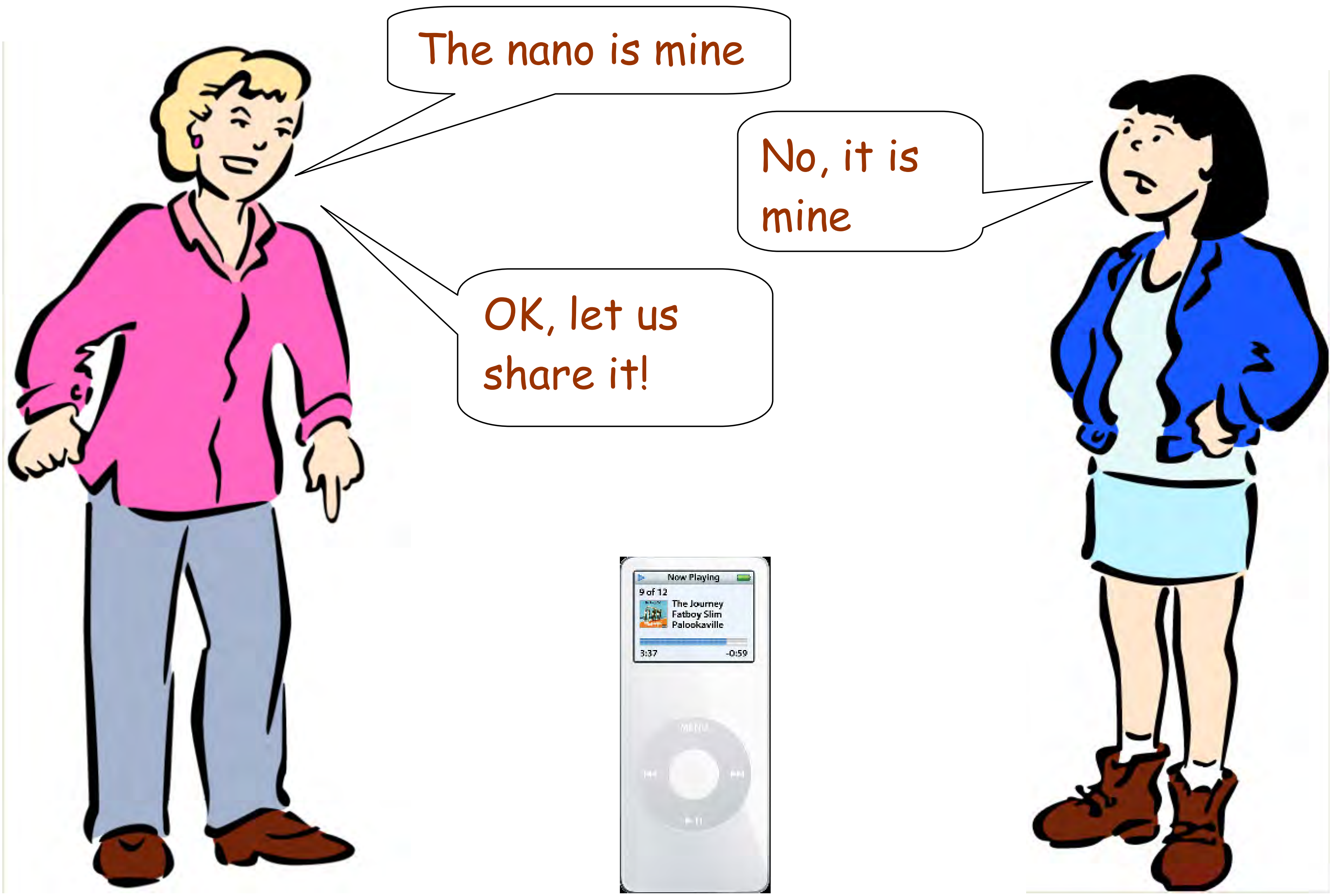
Aldrich at al have developed tools with extract such architectural descriptions from the program code

Survey – Summary

	Box is a	M/D	nest dpth	TR	Obj has ? boxes	Obj in boxes
Confined types	package	OAD	1	no	1	1
Object Encapsulation	object	OAD	n	some	1	1
Locking	object or thread	none	n	no	1	1
Universes Boogie	object	OAM	n	yes	1	1
Ownership domaind	"object"	none	n	no	n	1

Where OAM = owners as modifier; OAD = owner as dominator;
TR = ownership transfer

However . . .



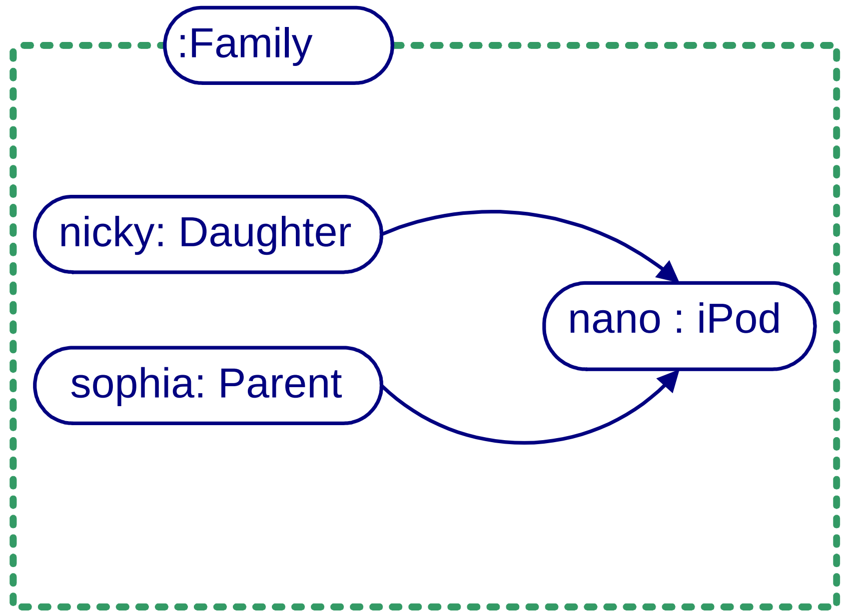
The nano is mine
No, it is mine
OK, let us share it!
Now Playing
9 of 12
The Journey
Fatboy Slim
Palookaville
3:37
-0:59

Common Ownership - The Classic Way

Put the nano in the most enclosing inner box.

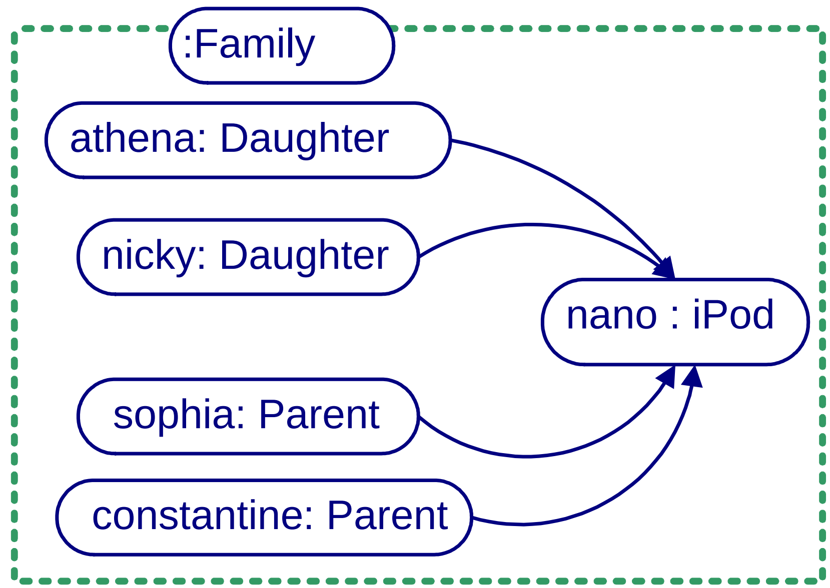
```
class Family<o> {...
    iPod<this> nano;
    Daughter<this> nicky;
    Parent<this> sophia;
    ...
 }
```

then:



Common Ownership - The Classic Way - Limitations

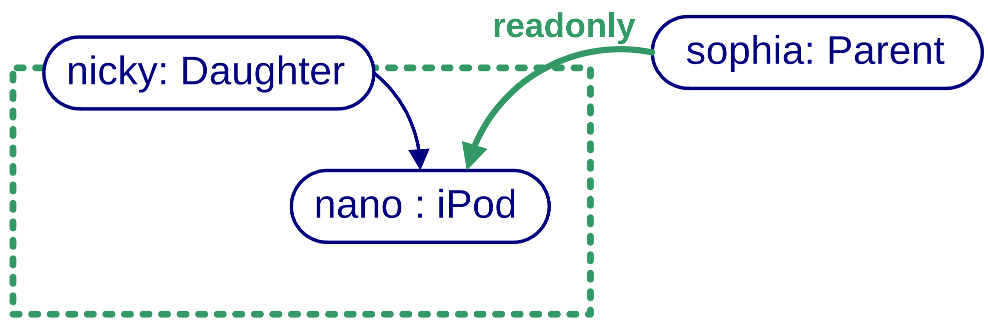
However, the family also includes athena and constantine. Therefore, they too will get their hands on the nano....



Common Ownership - The Universes Way

Give sophia a readonly reference to the nano.

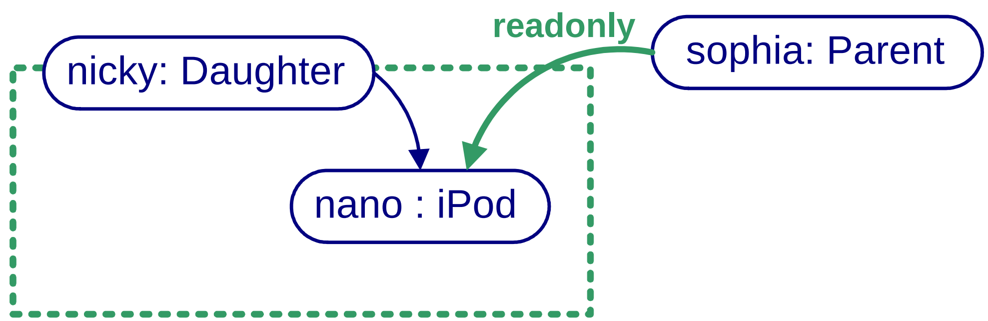
```
class Daughter {...
   rep iPod nano;
   ...
 }

class Parent {...
   readonly iPod nano;
   ...
 }
```



then, sophia can listen to the nano.

Common Ownership - the Universes Way - Limitations

However, then, sophia cannot switch the nano on or off!

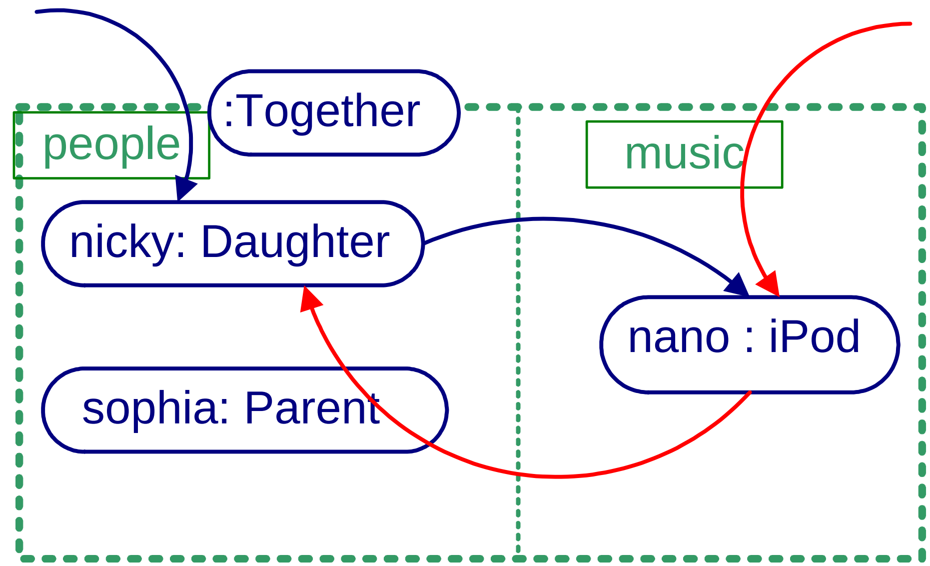


Common Ownership – Ownership Domains Way

Put sophia and nicky in the same ownership domain, with access to the domain containing nano.

```
class Daughter { ... }

class Parent { ... }

class Together {
   public domain people;
   domain music;
   link people->music;
   people Daughter nicky;
   people Parent sophia;
   music iPod nano;  }
```

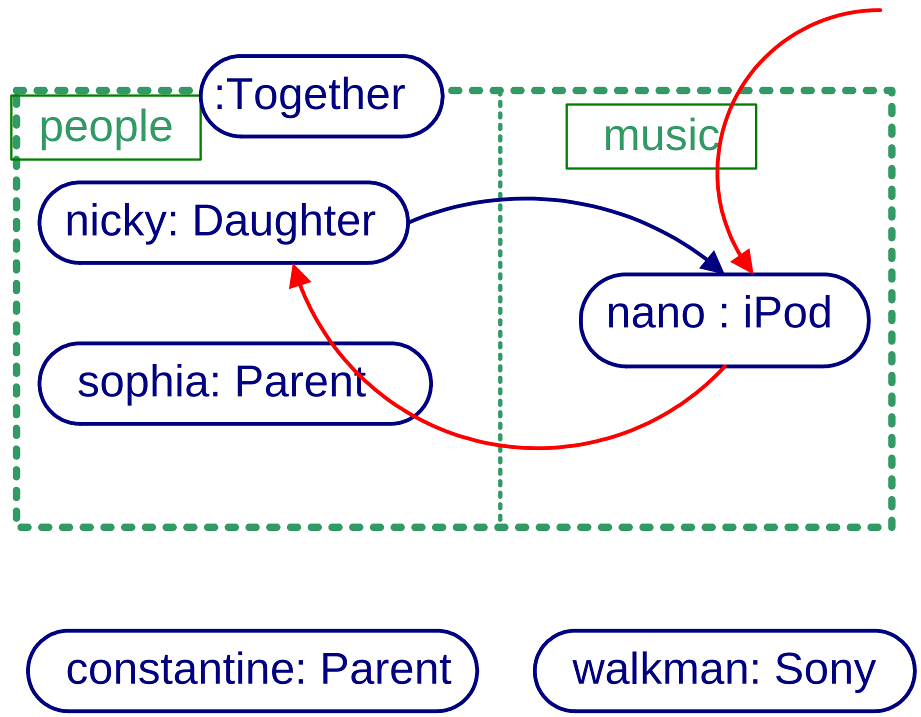
then, only sophia and nicky can manipulate nano.



Common Ownership - Ownership Domains Way - Limitations

However, what if sophia wanted to

- share the nano with nicky, and also
- share the walkman with constantine?



We developed

MOJO, Multiple Ownership for Java Objects

Multiple Ownership -- OOPSLA 2007

- allow *more than one* hierarchy
- allow *more than one* owner

Multiple Ownership

Nicholas R Cameron,
Sophia Drossopoulou
Department of Computing,
Imperial College London, UK
{ncameron, sd}@doc.ic.ac.uk

James Noble *
School of Mathematics, Statistics
& Computer Science,
Victoria University of Wellington,
New Zealand
kjx@mcs.vuw.ac.nz

Matthew J Smith
Department of Computing,
Imperial College London, UK
mjs198@doc.ic.ac.uk

Abstract

Existing ownership type systems require objects to have precisely one primary owner, organizing the heap into an ownership tree. Unfortunately, a tree structure is too restrictive for many programs, and prevents many common design patterns where multiple objects interact.

1. Introduction

> *We're tired of trees... We should stop believing in trees, roots, and radicles.*
>
> Deleuze and Guattari, **A Thousand Plateaus** [17]

Multiple Ownership – An Example

Tasks and employees as before.

A project consists of a sequence of tasks.

When a project is delayed, its tasks get delayed accordingly.

A project is OK, if all its tasks are within their due dates.

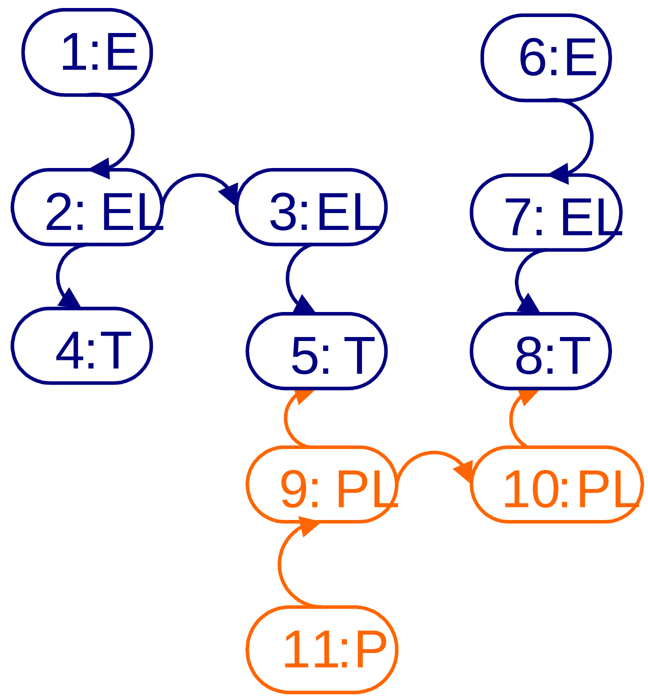
In the code we omit Node class.

"Java" code

```
class Employee {
    EList tasks;
    void delay( ){ ... } }
class Project {
    List tasks;
    void delay( ){ ... } }

class List {
    List next;
    Task task;
    void delay( ){ ... } }
class Task {  ...
    void delay(){ ... }; }
```

Multiple Ownership – An Example

```
class Employee {
   List tasks;
   void delay( ){ ... } }
class Project {
   List tasks;
   void delay( ){ ... } }

class List {
   List next;
   Task task;
   void delay( ){ ... } }
class Task {  ...
   void delay(){ ... }; }
```

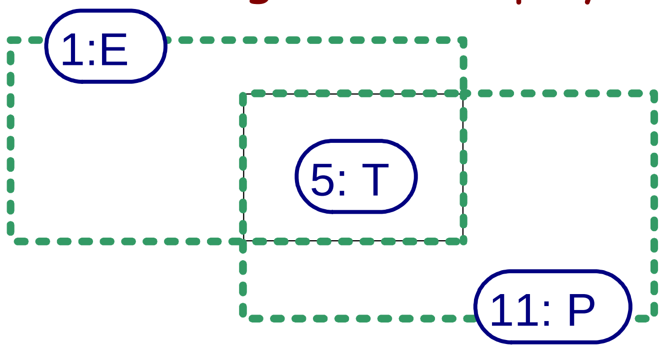


We want:

$e1 \nparallel e2 \vdash e1.delay() \# e2.OK()$

$p1 \nparallel p2 \vdash p1.delay() \# p2.OK()$

Need to express that a task belongs to an employee *and* a project, e.g.



task 5 is owned by Employee 1, *and* Project 11.

Here, `Task< 1&11 >`

In general, we allow types like

`A<o1`**&o2**`,o3,o5`**&o6**`>`

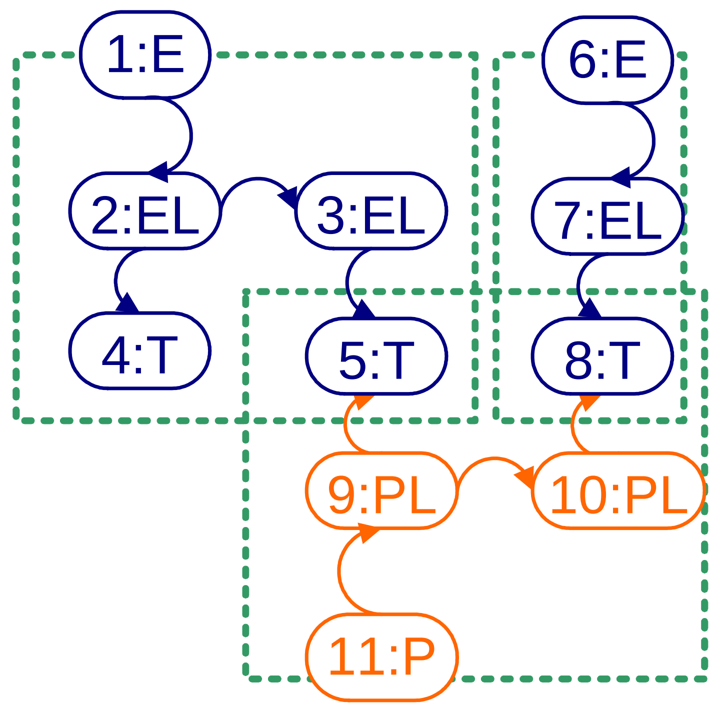
or

`A<o1`**&any**`,o3,`**any**`>`

In a type, we say **any**, when actual owner unknown (cf **readonly**).

Multiple Ownership

```
class Employee<o> {
    List<this,this> tasks;
    void delay( ){ ... } }
class Project<o> {
    List<this,this> tasks;
    void delay( ){ ... } }

class List<o1,o2> {
    List<o1,o2> next;
    Task<o1,o2& any> task;
    void delay( ){ ... } }
class Task<o1>{  ...
    void delay(){ ... }; }
```



NOTE: ☺ `Task` class unaware of number of owners. ☺

NOTE: ☹ `List` class aware of number of owners. ☹

BEGIN ASIDE: the meaning of **any**

any = corresponding owner is unknown, but fixed;

We have to distinguish the *don't know* from the *don't care* use of any.

```
class List<o1,o2> {
    …
    List<o1,o2> next;
}
     …
List<o4,o5&any> l1;              // any as don't care
```

BEGIN ASIDE: the meaning of any

any = corresponding owner is unknown, but fixed;

We have to distinguish the *don't know* from the *don't care* use of any.

```
class List<o1,o2> {
    …
    List<o1,o2> next;
}

    …
List<o4,o5&any> l1;                  // any as don't care
l1 = new List<o4,o5&o6>;             // OK
l1 = new List<o4,o5&o7&o8>;          // OK

l1.next;                             :  List<o4,o5&any>
                                     // any as don't know

l1.next:= new List<o4,o5&o6>;        : ERROR
l1.next:= l1;                        : ERROR
```

To distinguish the don't care from the don't know, we employ different field lookup functions upon field read and upon field write,

For field read

$$\frac{\Gamma \vdash e : t \qquad fType(t, f, e, \Gamma) = t'}{\Gamma \vdash e.f : t'}$$

$$fType(c{<}\overline{Q}{>}, f, e, \Gamma) = [\overline{Q/p}](t^{\Gamma \cdot e}) \qquad \text{where } t = fType^{aux}(c{<}\overline{p}{>}, f)$$

gives `l1.next; : List<o4,o5&any>`

To distinguish the don't care from the don't know, we employ different field lookup functions upon field read and upon field write,

For field read

....

gives `l1.next; : List<o4,o5&any>`

On the other hand, for field write

$$\frac{\Gamma \vdash e : t \qquad fType^{strict}(t, f, e, \Gamma) = t' \qquad \Gamma \vdash e' : t'}{\Gamma \vdash e.f{=}e' : t'}$$

$$fType^{strict}(c{<}\overline{Q}{>}, f, e, \Gamma) = [\overline{Q/p}]^{strict} t', \qquad \text{where} \quad t = fType^{aux}(c{<}\overline{P}{>}, f) \;\; \text{and} \; t' = t^{\Gamma \cdot e}$$

gives `l1.next:= new List<o4,o5&o6>; : ERROR`

END ASIDE

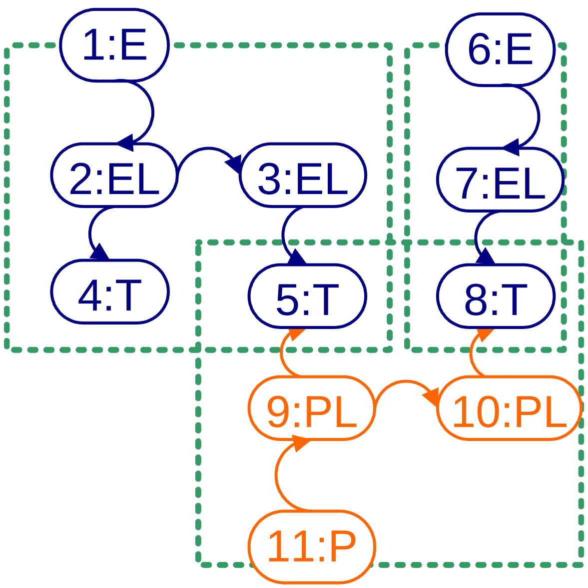
We want to be able to argue:

$$\texttt{e1} \not\parallel \texttt{e2} \vdash \texttt{e1.delay()} \ \# \ \texttt{e2.OK()}$$

We first define when an object is "inside" another object, i.e. $\iota \ll \iota'$ as the minimal reflexive, transitive relation, such that

if one of the owners of ι is ι' then $\iota \ll \iota'$

Therefore



$5 \ll 5$

$5 \ll 1$

$5 \ll 11$

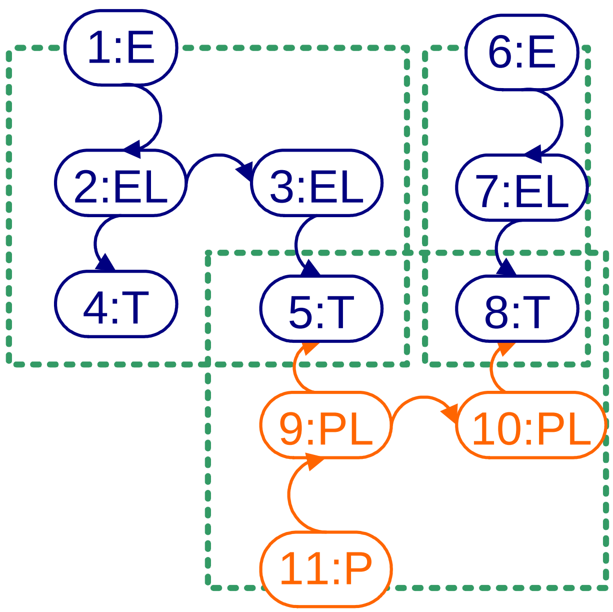
Define *run-time* effects: $\chi ::= \iota \mid \chi.\mathtt{undr} \mid \chi \;\&\; \chi$

meaning:

$$[\![\iota]\!] = \{ \iota \}$$

$$[\![\chi.\mathtt{undr}]\!] = \{ \iota \mid \iota \ll [\![\chi]\!] \}$$

$$[\![\chi \;\&\; \chi']\!] = [\![\chi]\!] \cap [\![\chi']\!]$$



$[\![$ 1 $]\!] = \{$ 1 $\}$

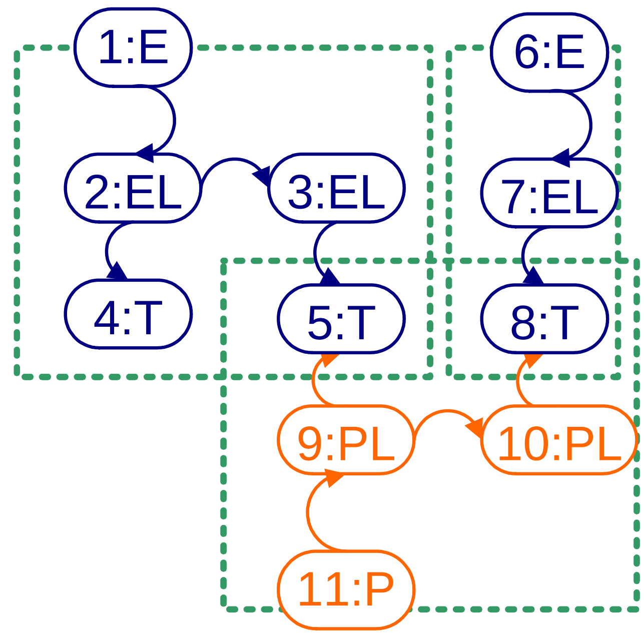
$[\![$ 1.under $]\!] = \{$ 1, 2, 3, 4, 5 $\}$

$[\![$ 1.under & 11.under$]\!] = \{$ 5 $\}$

$[\![$ 1.under & 6.under$]\!] = \emptyset$

Define also an effects annotation system, which gives

```
class Employee<o> {
   List<this,this > tasks; void
   void delay()this&any.undr {..}
}
class Project<o> {
   ...
   void delay() this&any.undr {..}
}

class List<o1,o2> {
   ...
   void delay()o2.undr {..} }
}
class Task<o>{  ...
   void delay()this.undr{..}
}
```



For stack s and heap h, define $[[\phi]]_{s,h}$ the obvious way.

Define judgement $\Gamma \vdash \phi \# \phi'$ to denote disjointness of effects

Lemma:

$$\Gamma \vdash s, h \qquad \Gamma \vdash \phi \# \phi' \qquad \Rightarrow \qquad [[\phi]]_{s,h} \cap [[\phi']]_{s,h} = \varnothing$$

Execution of an expression does not require/modify more than what is described by the read/write effects:

Theorem:

$$\left.\begin{array}{l} \Gamma \vdash_{rd} e : \phi_1 \quad \Gamma \vdash_{wr} e : \phi_2 \\ \Gamma \vdash s, h \\ e, s, h \rightsquigarrow v, h' \end{array}\right\} \Rightarrow \left\{\begin{array}{l} h = [[\phi_1]]_{s,h} * h_2 \\ [[\phi_1]]_{s,h} = [[\phi_2]]_{s,h} * h_3 \\ h' = h'' * h_3 * h_2 \\ e, s, [[\phi_2]]_{s,h} * h_3 \rightsquigarrow v, h'' * h_3 \\ \text{for some } h_2, h_3, h'' \end{array}\right.$$

Thus, `e1.delay() : (e1&any).under`

`e2.OK() : (e2&any).under`

Because $\texttt{e1} \nparallel \texttt{e2} \vdash \texttt{(e1\&any).under} \;\#\; \texttt{(e2\&any).under}$

we have $\texttt{e1} \nparallel \texttt{e2} \vdash \texttt{e1.delay()} \;\#\; \texttt{e2.OK()}$

Similarly, $\texttt{p1} \nparallel \texttt{p2} \vdash \texttt{p1.delay()} \;\#\; \texttt{p2.OK()}$

☺

Can I preserve owners as dominators?

Yes, in a way, if we

- require that in each type definition the actual owner parameters are "within" the actual context parameters,
- define a program "slice", Pi, where each class as a "selected" ownership parameter out of the may ownership parameters.
- For each slice, we filter the heap, by dropping any field whose selected owner is not "outside" the selected owner parameter of the defining class.

Can I preserve owners as dominators? yes, partly

Yes, in a way, if we

- require that in each type definition the actual owner parameters are "within" the actual context parameters,
- define a program "slice", Pi, where each class as a "selected" ownership parameter out of the may ownership parameters.
- For each slice, we filter the heap, by dropping any field whose selected owner is not "outside" the selected owner parameter of the defining class.

Then

- For each of the slices, the selected owners are dominators in the correspondingly filtered heap.

Preserving owners as dominators - partly - P1 slice

Selected owner **higlighted**,

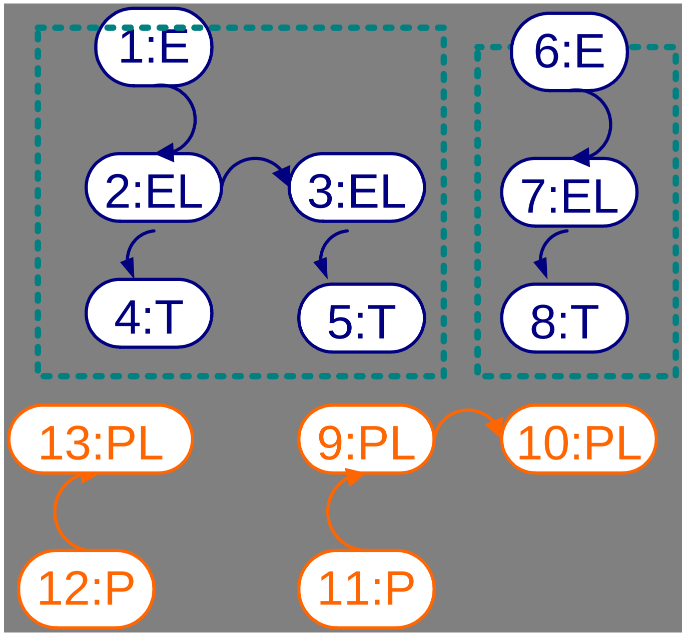
```
class Task<o1,o2:>{  ...  }
class Employee<o:> {
    EList<this:> tasks;
.. }
class EList<o:> {
    EList<o:> next;
   Task<o,any:> task;
... }
class Project<o:> {
    PList<this:> tasks; ...  }
class PList<o:> {
    PList<o:> next;
    Task<any,o:> task;
  ...  }
```

Preserving owners as dominators - partly - P1 slice

Selected owner **higlighted**,

// and fields filtered out

```
class Task<o1,o2:>{  ...  }

class Employee<o:> {
    EList<this:> tasks;
.. }

class EList<o:> {
    EList<o:> next;
   Task<o,any:> task;
... }

class Project<o:> {
    PList<this:> tasks; ...  }

class PList<o:> {
    PList<o:> next;
    // Task<any,o:> task;
  ...  }
```



Preserving owners as dominators - partly - P2 slice

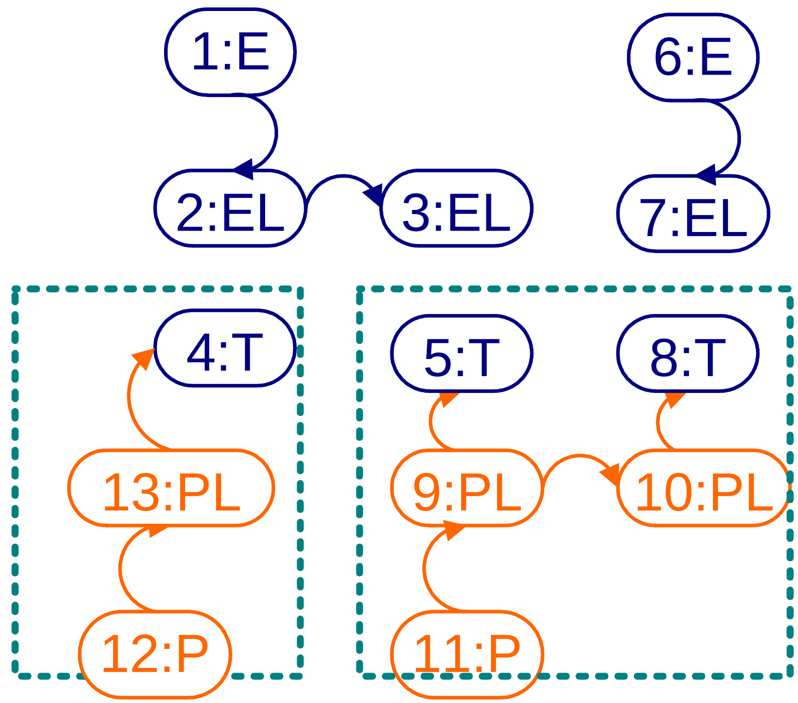
Selected owner **higlighted**

```
class Task<o1,o2:>{  ...  }
class Employee<o:> {
    EList<this:> tasks;
.. }
class EList<o:> {
    EList<o:> next;
    Task<o,any:> task;
... }
class Project<o:> {
    PList<this:> tasks; ...  }
class PList<o:> {
    PList<o:> next;
    Task<any,o:> task;
  ...  }
```

Preserving owners as dominators - partly - P2 slice

Selected owner **higlighted**,
// and fields filtered out

```
class Task<o1,o2:>{  ...  }

class Employee<o:> {
    EList<this:> tasks;
.. }

class EList<o:> {
    EList<o:> next;
    // Task<o,any:> task;
... }
class Project<o:> {
    PList<this:> tasks; ...  }

class PList<o:> {
    PList<o:> next;
    Task<any,o:> task;
  ...  }
```



Multiple Owners and Aspects

Aside: I started tackling this problem (independence of actions and assertions in the presence of "overlapping topologies") unsuccessfully by filtering out fields in and off for the three years. Multiple owners was the missing link, and in particular the idea of intersection – *remember basic set theory?*

Looking for an AOP view, where
the program is

$$P = P1 \oplus P2 \oplus \dots \oplus Pn$$

the heap is

$$h = h1 \oplus \dots \oplus hn$$

and execution of P consists of the combination of execution of P1,P2,..., Pn, and preserves some of the properties established in the context of Pi.

$h1 \oplus h2 = h0 * h3 * h4$ where $h1 = h0 * h3$ and $h2 = h0 * h4$

Multiple Ownership - Conclusions

- multiple owners are possible,
- multiple owners describe realistic object topologies, and thus document programmer's intuitions,
- multiple owners can be used to argue disjointness.

Multiple Ownership – Further Work

- refine type system (**any** as existential, refine scope),
- apply to concurrency and verification,
- *AOP*: combine two programs into one program with multiple ownership hierarchies.

Putting MOJO into Context

	Box is a	M/D	nest dpth	TR	Obj has ? boxes	**Obj in boxes**
Confined types	package	OAD	1	no	1	**1**
Object Encapsulation	object	OAD	n	some	1	**1**
Locking	object or thread	none	n	no	1	**1**
Universes Boogie	object	OAM	n	yes	1	**1**
Ownership domains	"object"	none	n	no	n	**1**
MOJO	"object"	none	n	no	1	**n**

where OAM = owners as modifier; OAD = owner as dominator;
TR = ownership transfer

The Benefits of Putting Objects into Boxes
Conclusions

- “boxes” express and preserve a topology in the object heap;
- topology exploited for different goals, eg encapsulation, memory management, program verification, concurrency
- different goals impose slightly different constraints and notations – a unification would be nice (pluggable types).
- notation heavy in some cases; some nice simplifications exist, more are currently being developed.
- type inference exists for some systems, more would be good.

… not bad for a paper without implementation, without semantics, but with **compelling diagrams**,

Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch

Abstract. Aliasing is endemic in object oriented programming. Because an object can be modified via any alias, object oriented programs are

Ten years, later, we have many implementations, many semantics, vibrant research, diverse application areas, many further publications (eg 3 in ECOOP 07, 3 in OOPSLA 07), and recognition (J. Aldrich awarded the Junior Dahl/Nygaard Prize in 2007).

Thank you!