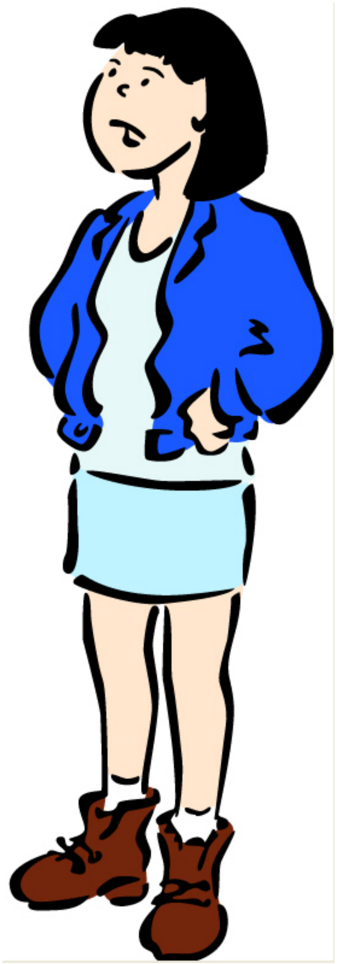
The Benefits of Putting Objects into Boxes

- using pencil and paper -

Sophia Drossopoulou
Department of Computing, Imperial College London

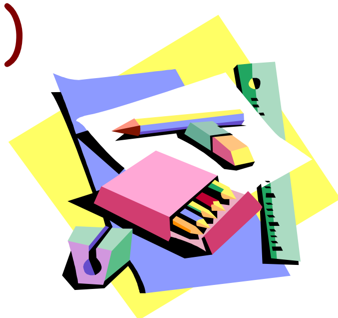


... **1972**, Scientific Gymnasium, Athens,
20 boys, 4 girls

... trying to decide what to study,

... came across basic set theory, where

$$(P * Q) + R = (P + R) * (Q + R)$$



... found it intriguing, because, eg

$$(2 * 1) + 5 \neq (2 + 5) * (1 + 5)$$

... so, settled on "Informatik"

1973, Fakultaet fuer Informatik, Univ. Karlruhe

10 Professors (age 35-45), no women,

40 academic staff, 4 women

very enthusiastic, none knew much computer science

600 students, 40% women

Professor Krueger:

"computer science will be golden opportunity for women because

1) women are more precise,
2) they will be able to work from home"

The “Rechenzentrum” had

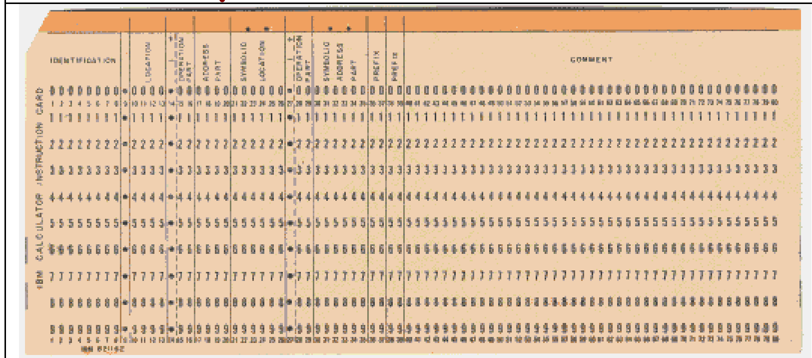
IBM 360/370

Burroughs 9000

PDP 9/11

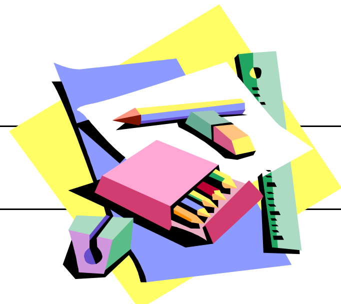
Univac 1108

used punched cards ...



it was very slow ....

pencil and paper much faster...

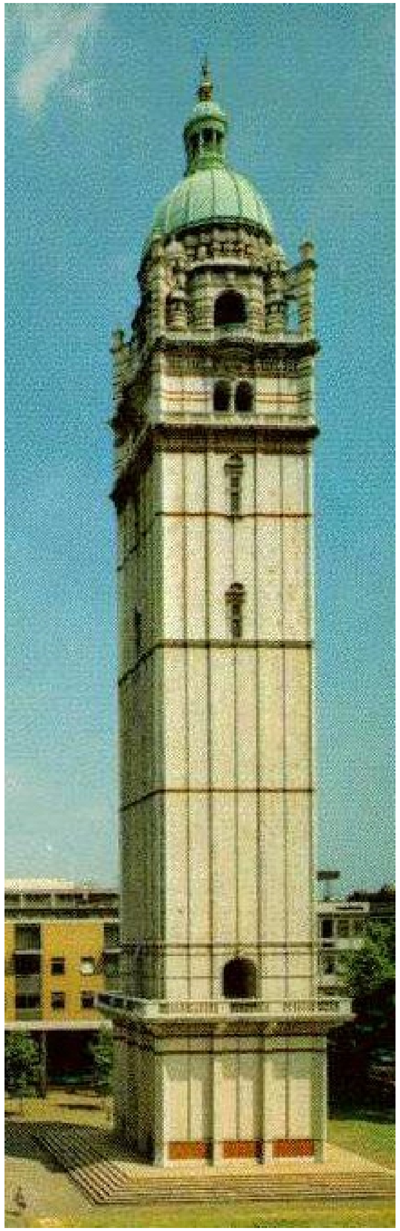


1979 obtained Dimplom

joined Prof. Goos group - 20 people and 2 women;
worked on Modula-2 and Ada compilers;
attribute grammars

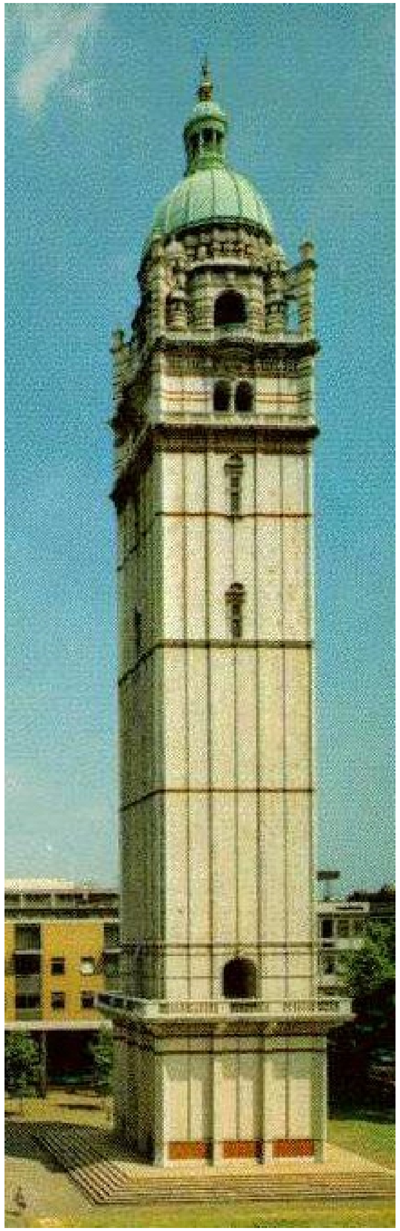
1982 PhD, Karlsruhe

work on parser generators theory; theory lead to new speedups
(30th PhD of Department,
10th foreigner, 1st woman)



1986 Department of Computing, Imperial College
roughly 30 academic staff,
of which 1 woman senior lecturer,
3 women lecturer

worked on the Flagship project on functional programming; and discovered the joys of object oriented programming



1986 Department of Computing, Imperial College
roughly 30 academic staff,
of which 1 woman senior lecturer,
3 women lecturers

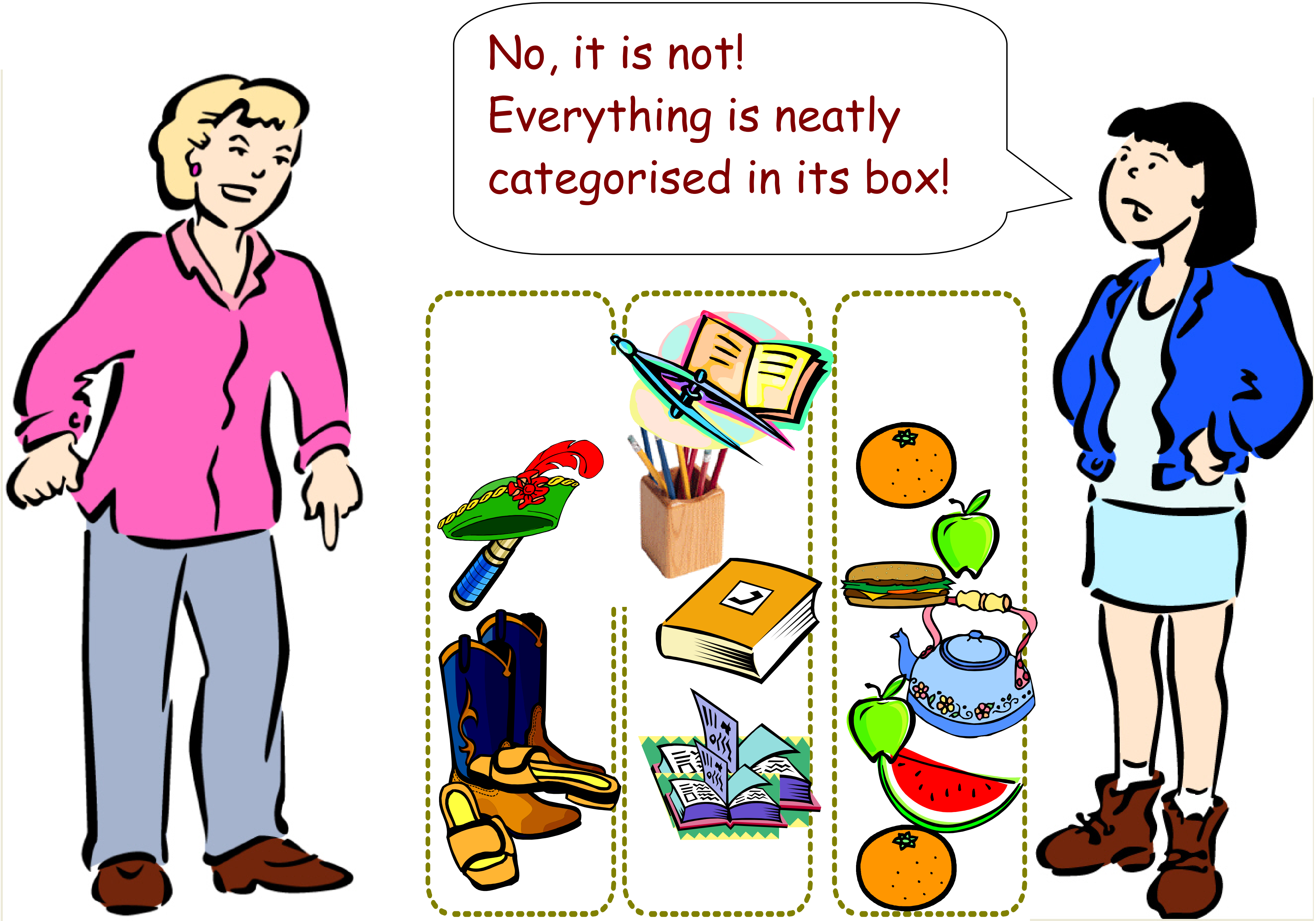
and 20 years later,
22 Professors (of which 1 woman)
6 Readers (of which 3 women)
15 Senior Lecturers (of which 6 women)
10 Lecturers (of which 1 woman)

In those 20 years, I worked on

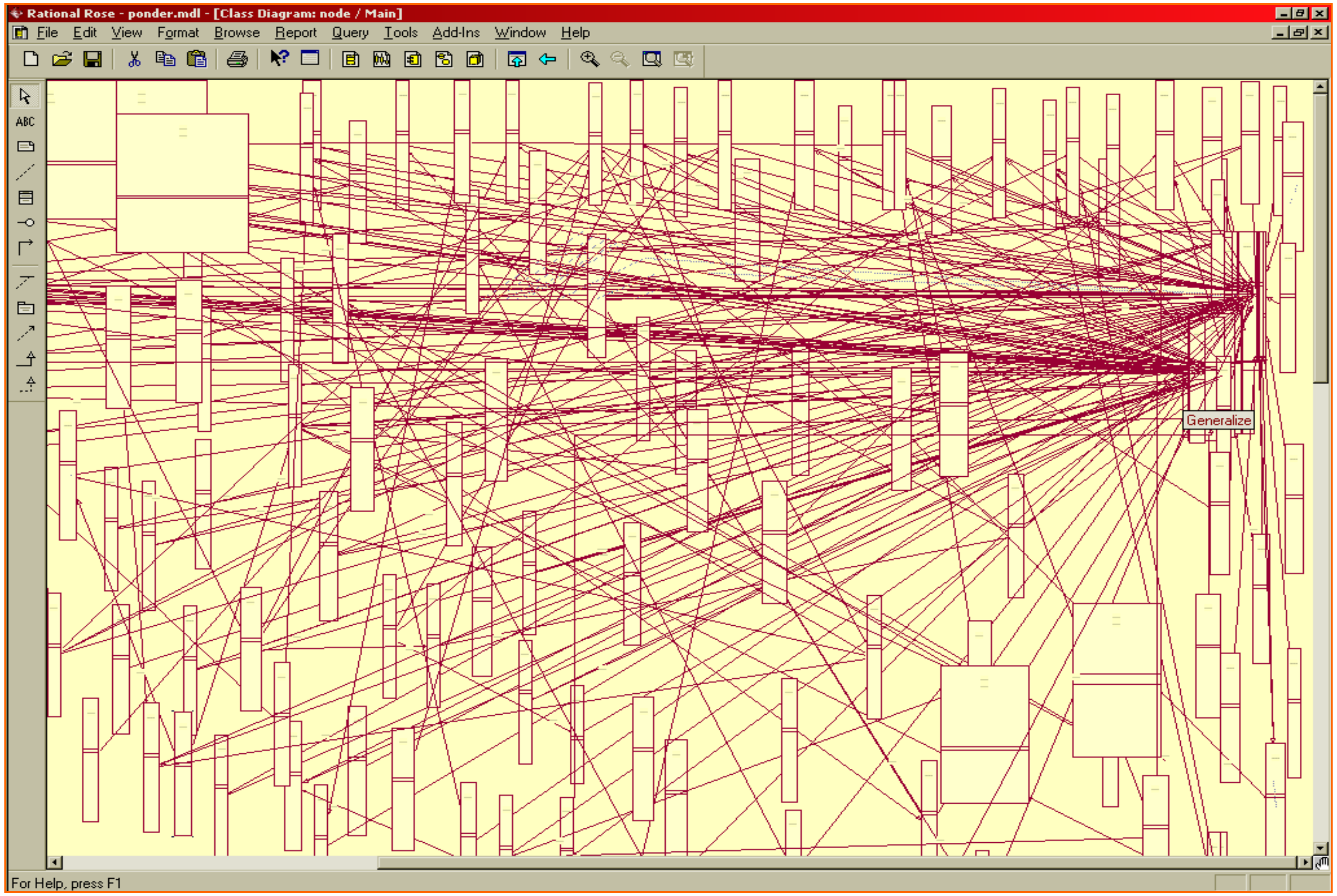
- Separate Compilation and Dynamic Linking
- Semantics of Java
- Dependent Classes
- Ownership Types
- Traits
- Generic Algorithms and Combinatorial Optimization
- Object Reclassification
- Types for Scripting Languages

and *family* Constantine, Sophia, Athena, Nicky ….


This room is a mess!


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

A common problem in programming



is that code structure/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 (4 strands),
- One further issue on boxes.

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

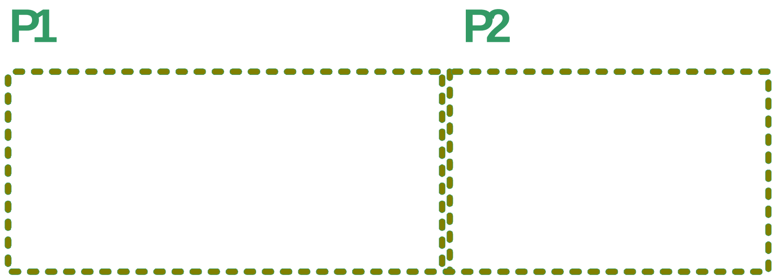
Therefore

- "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

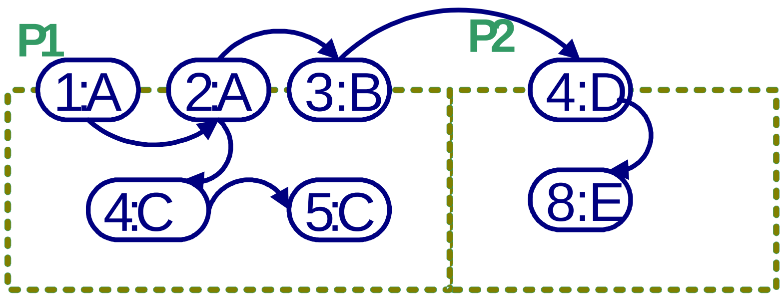
Therefore

- "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

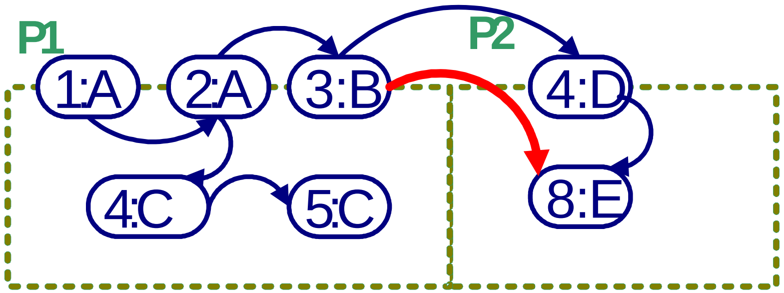
- "box" is a package; static boxes
- **owner as dominator**: no incoming references to a box

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,..

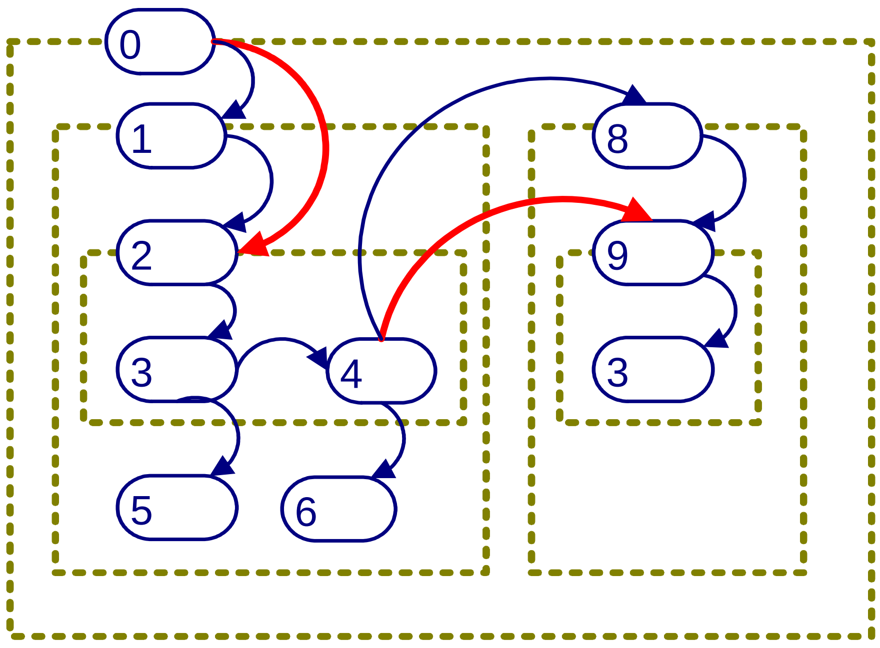
- each object belongs in a box;
- each box is characterized by an object (its owner)
- objects may hold references to objects in enclosing boxes

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

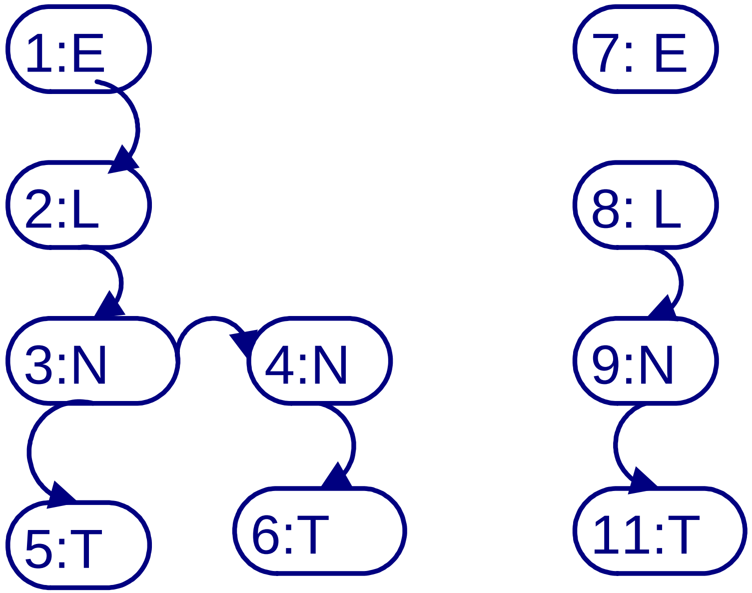
```
class Employee {
   List  tasks;
   void delay( ){ ... }
 }
 class List {
    Node first;
    void delay(){ ... }
 }
 class Node {
    Node next;
    Task task;
    void delay(){ ... }
 }
 class Task {  ...
    void delay(){ ... } }
```

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

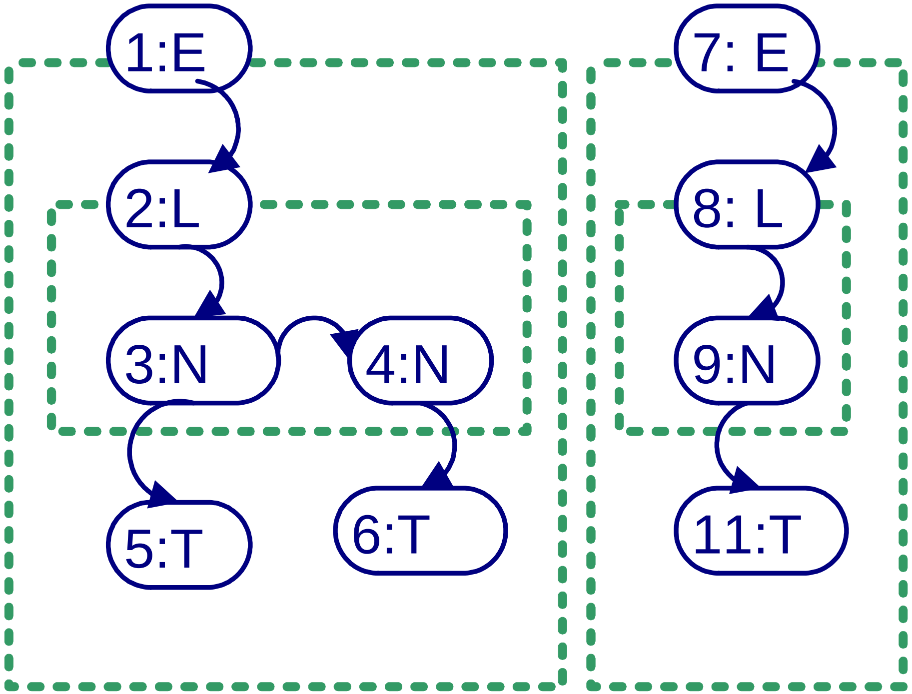


Boxes for Object Encapsulation – An Example

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

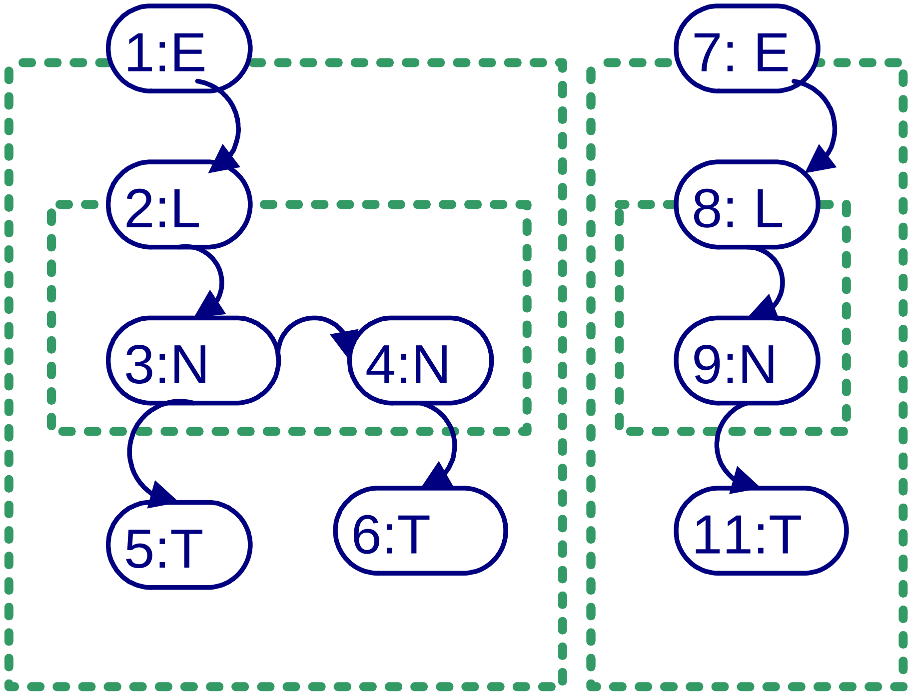
```
List<this>
```

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:

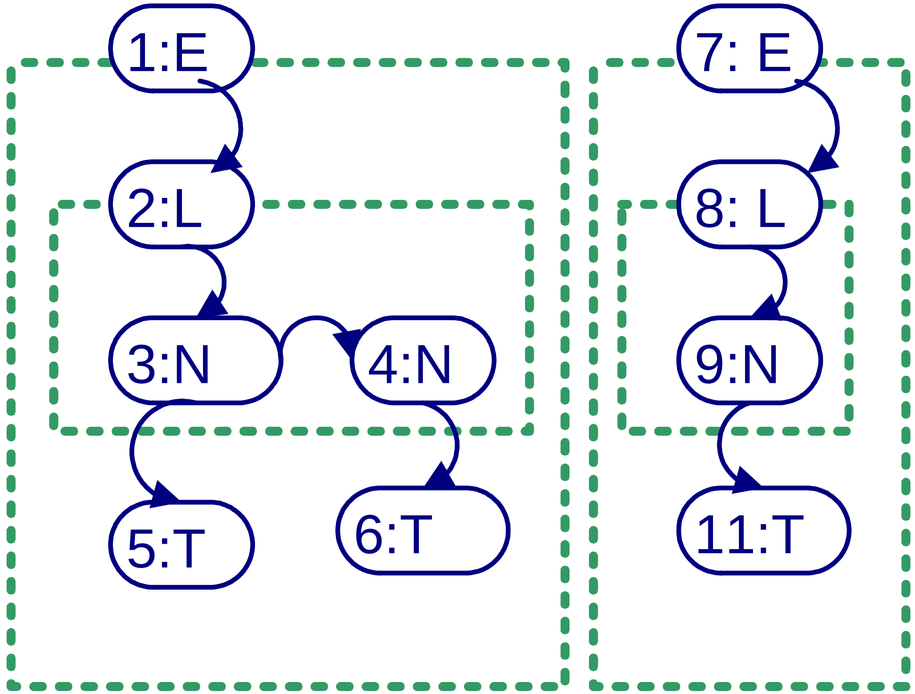


Boxes for Object Encapsulation – An Example

"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(){ ... } }
```

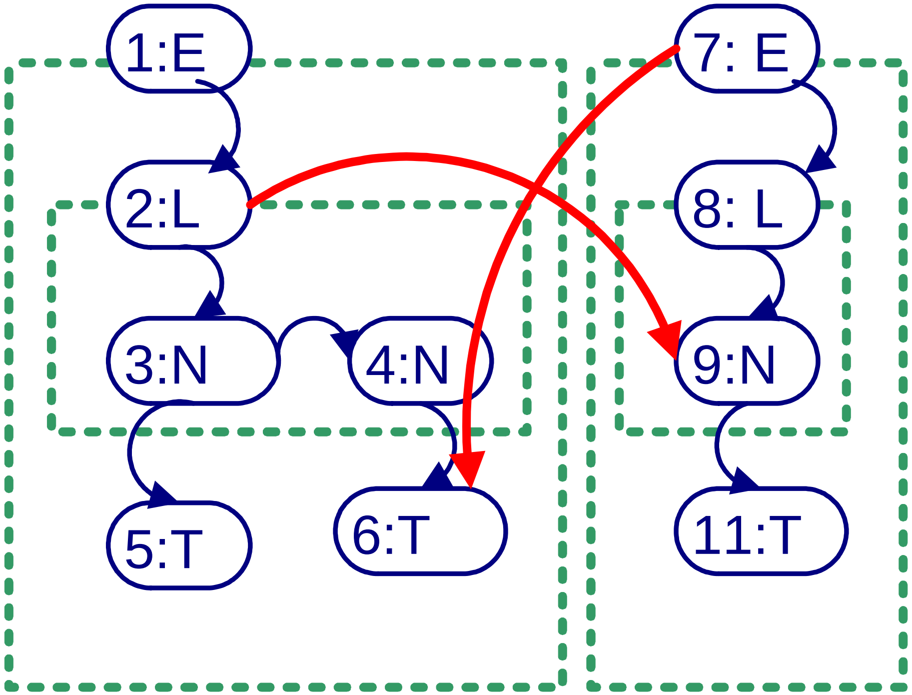
with a possible heap:



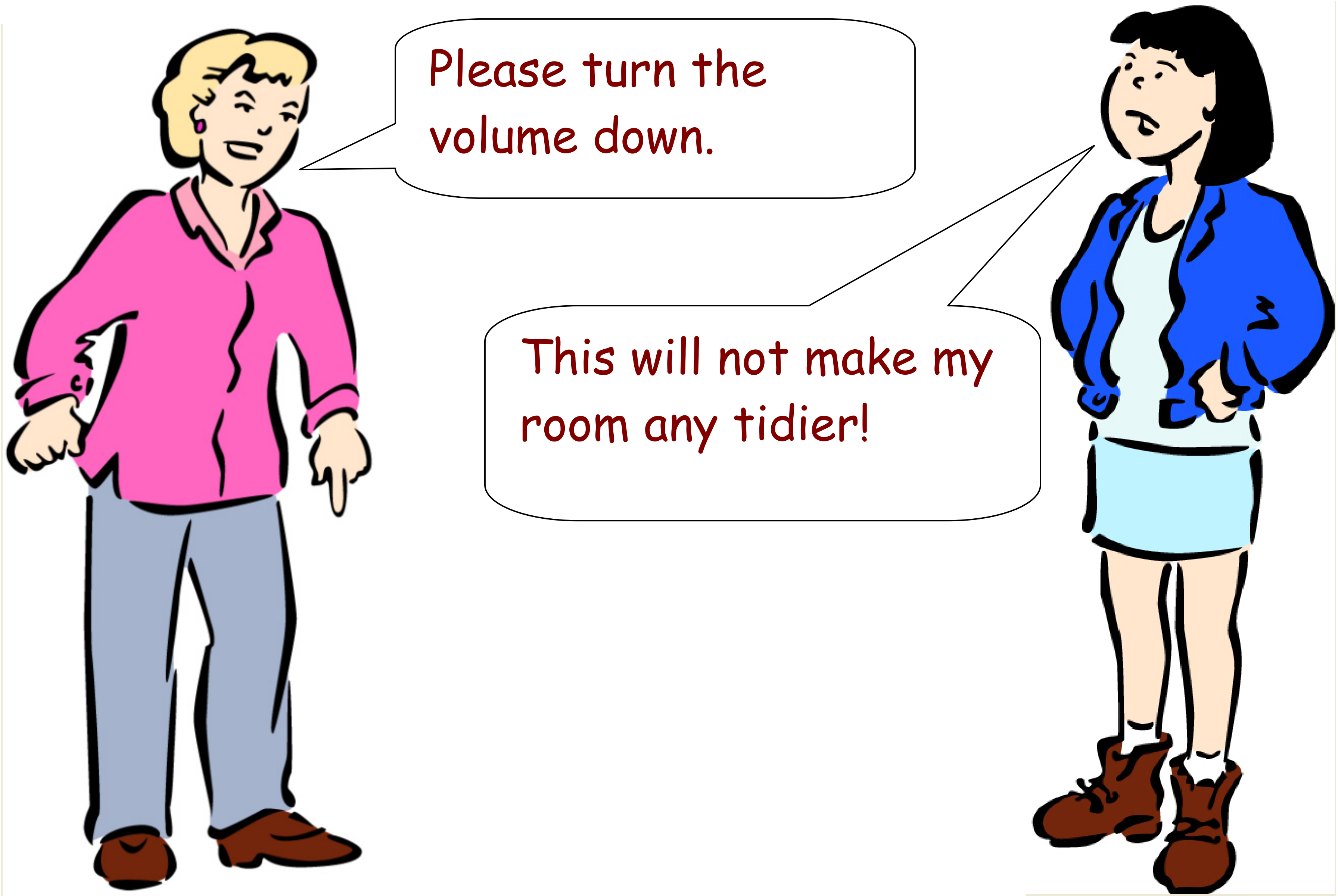
Boxes for Object Encapsulation – An Example

```
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 Object/Property Encapsulation

Clarke, Drossopoulou, Smith

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

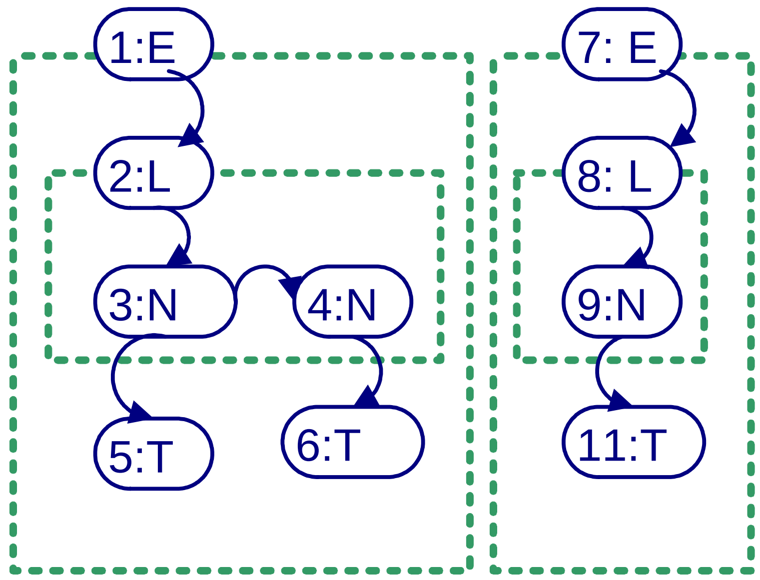
$$\texttt{e1} \not\parallel \texttt{e2} \vdash \texttt{e1.delay()} \,\#\, \texttt{e2.OK()}$$

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

For example:

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

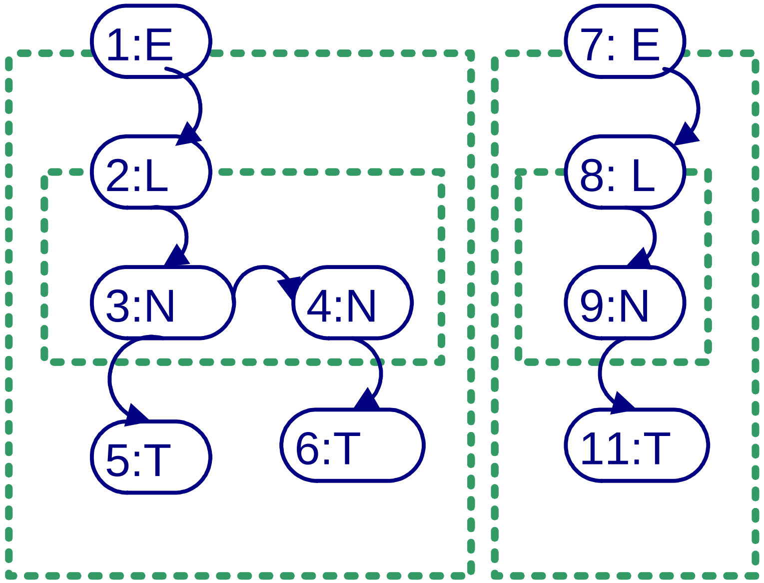
Disjoint boxes ⇒ independence



Boxes for Object/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

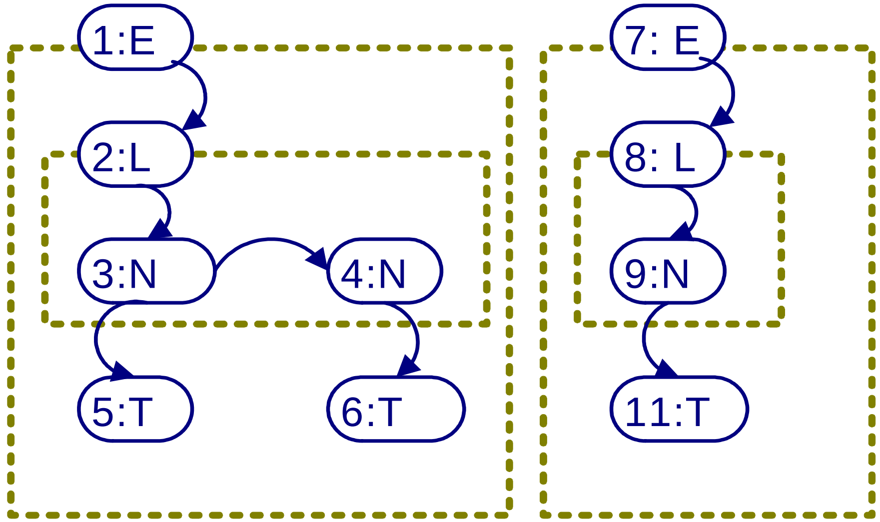
Zhao, Noble, Vitek, ...

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.



Survey - 3

Boxes for Concurrency

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

and also

Abadi, Flanagan, Freund, Qadeer, ...

Boxes for Concurrency

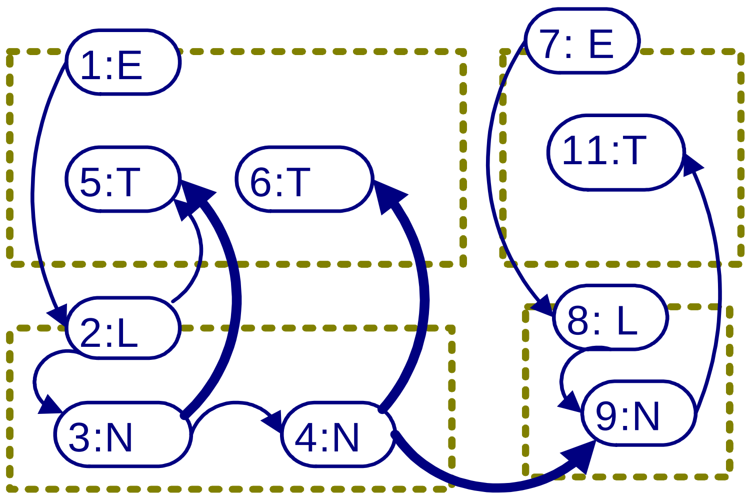
To avoid races/guarantee atomicity, a thread must have acquired the lock to an object before accessing it. The owner of a box stands for the lock of all the contained objects.

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 ...

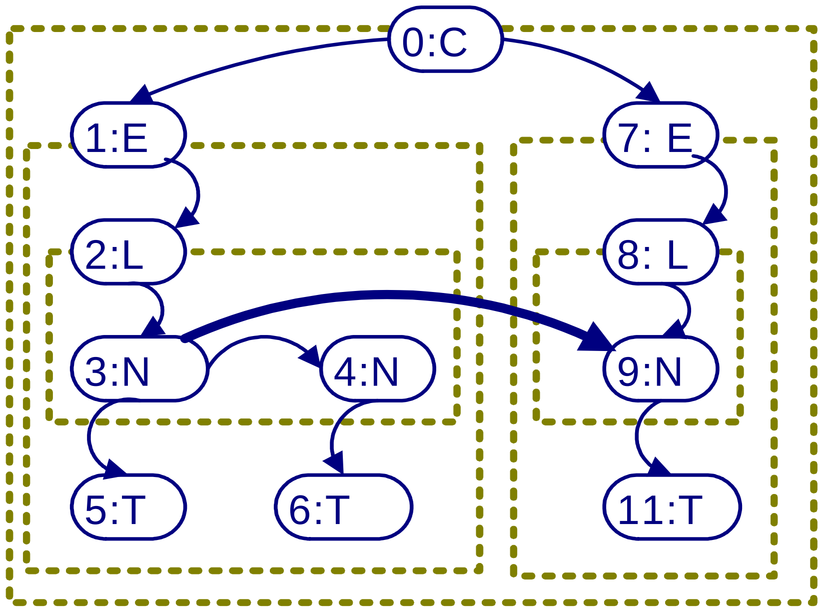
Boxes for Verification

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:

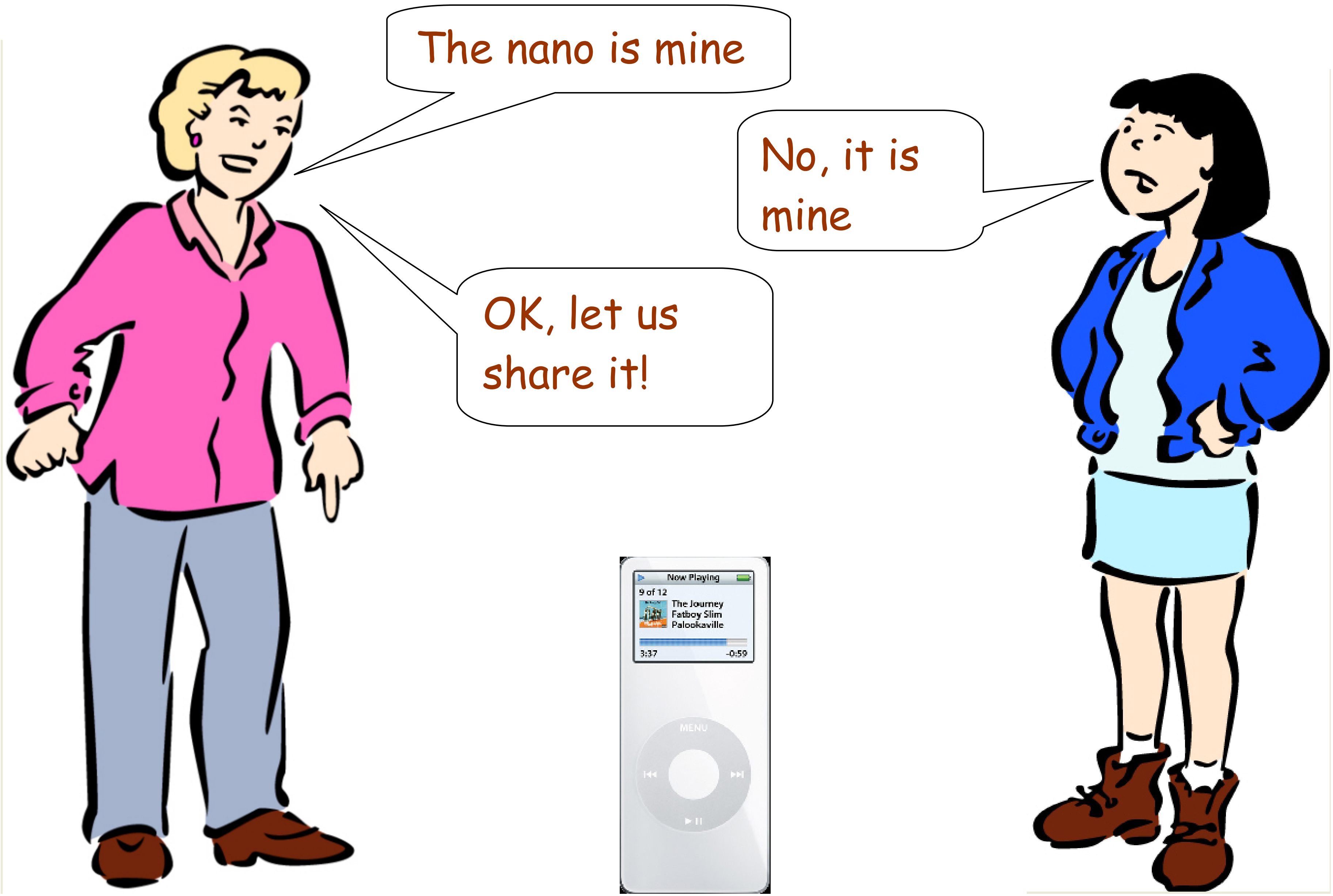
- owners may change; (5 may move to 7)
- no owners as dominators; (3 may have reference to 9)
- owner as modifier (3 may not change 9)



Survey - Summary

	owner is ...	owner as dominator?	benefit
Confined types	package – *static* number of owners	yes	object encapsulated in package
Object Encapsulation	an object	yes	object encapsulated in objects, scoped memory, visualization, independence
Locking	object or thread, holds "logic lock" to owned objects	no no nesting	guarantee race-free, or atomicity
Universes/ Boogie	an object; owner's properties depend on owned objects' state –	no; modifier owners may change	modular verification

However . . .

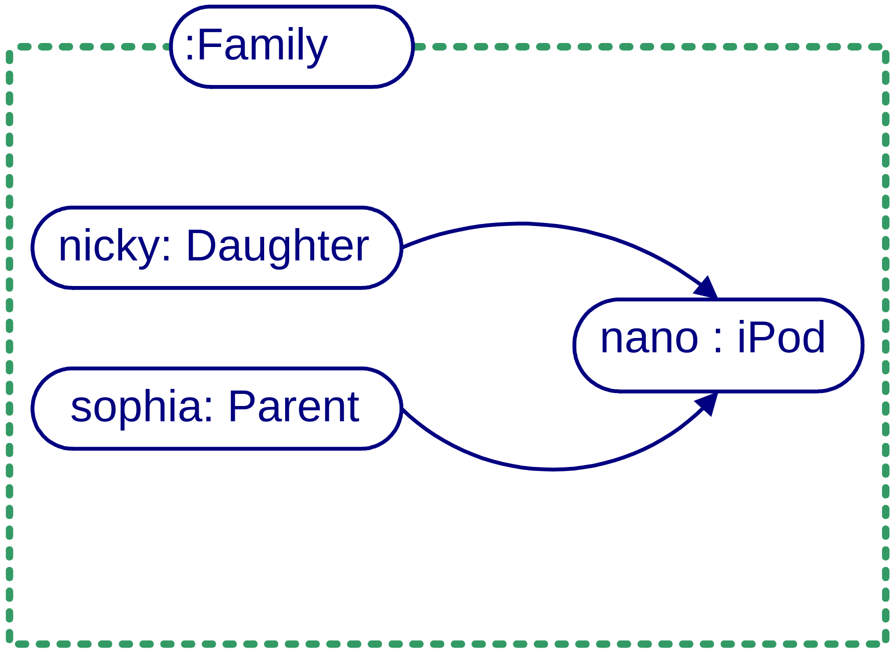

The nano is mine
No, it is mine
OK, let us share it!
Now Playing
9 of 12
The Journey
Fatboy Slim
Palookaville
3:37
-0:59
MENU

Common Ownership - The Classic Way

Put the nano in the most enclosing inner box.

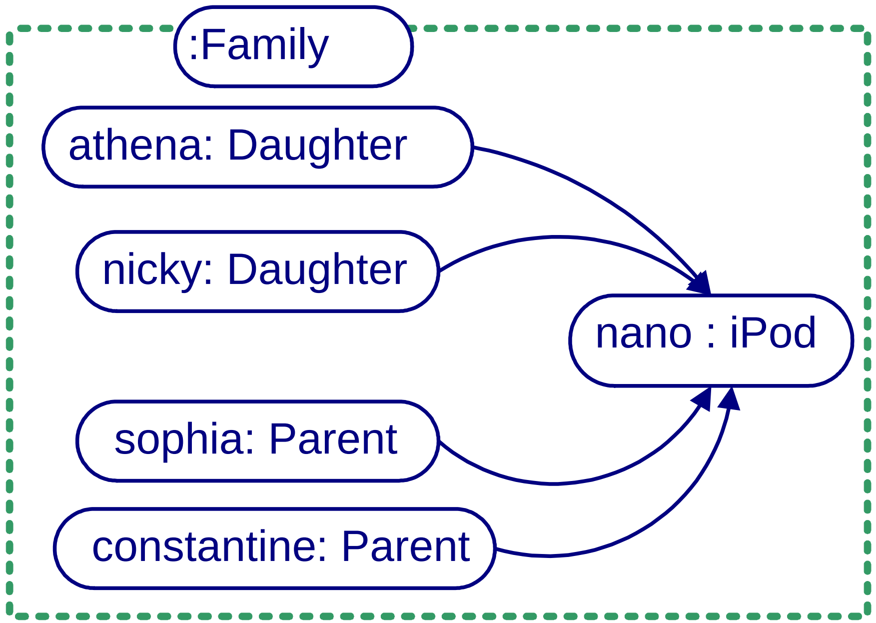
```
class Family<o> {...
    iPod<this> nano;
    Daughter<this> nicky;
    Parent<this> sophia;
    ...
 }
```

then:



Common Ownership - The Classic Way - Limitations

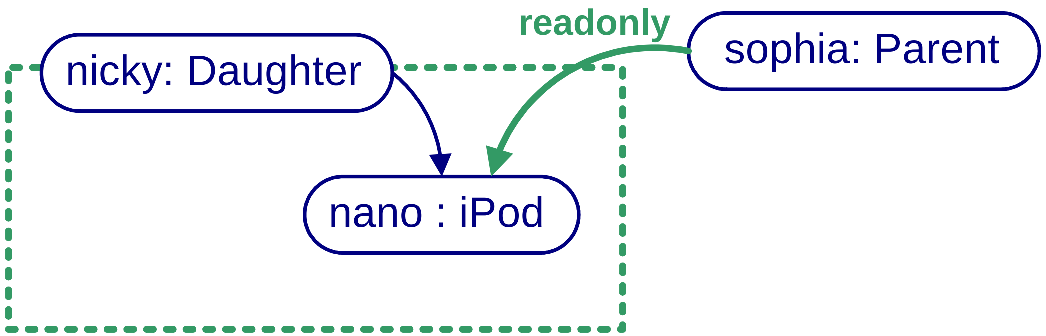
However, the family also includes athena and constantine. Therefore, they too will get their hands on the nano....



Common Ownership - The Universes Way

Give sophia a readonly reference to the nano.

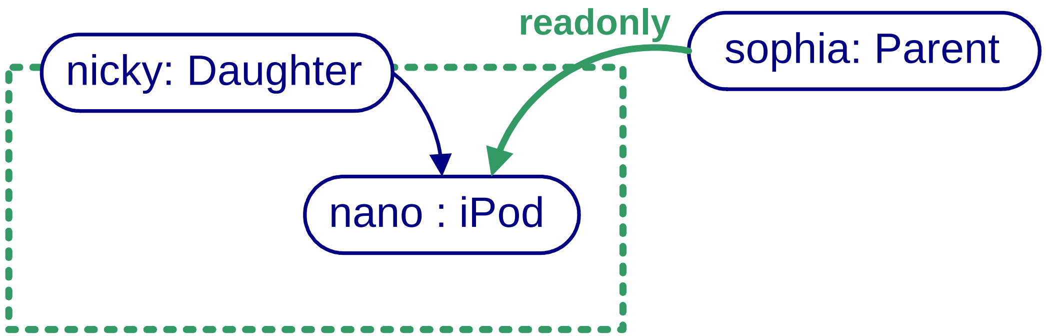
```
class Daughter {...
    rep iPod nano;
    ...
 }

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!

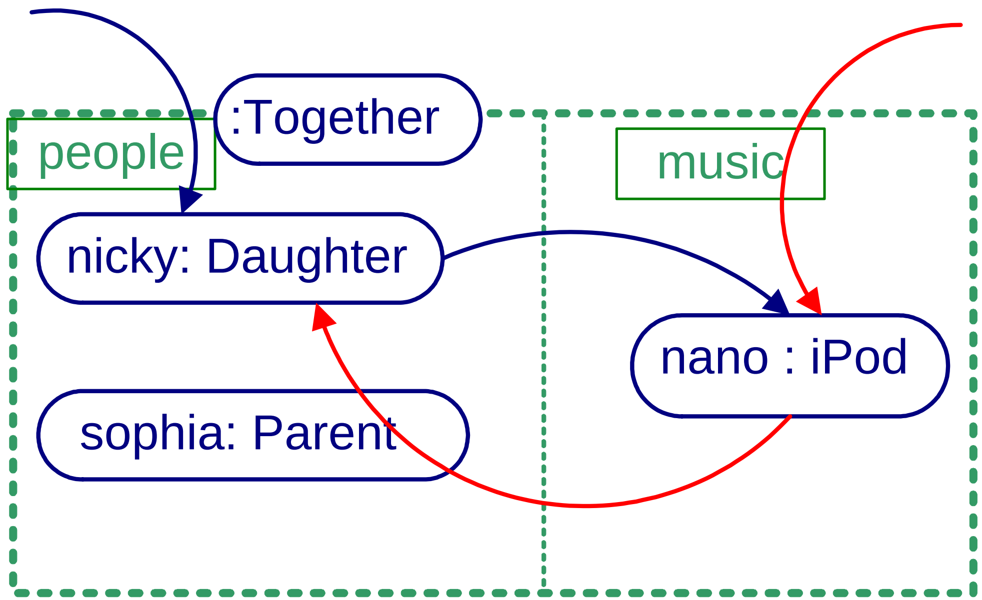


Common Ownership – Ownership Domains Way

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;  }
```

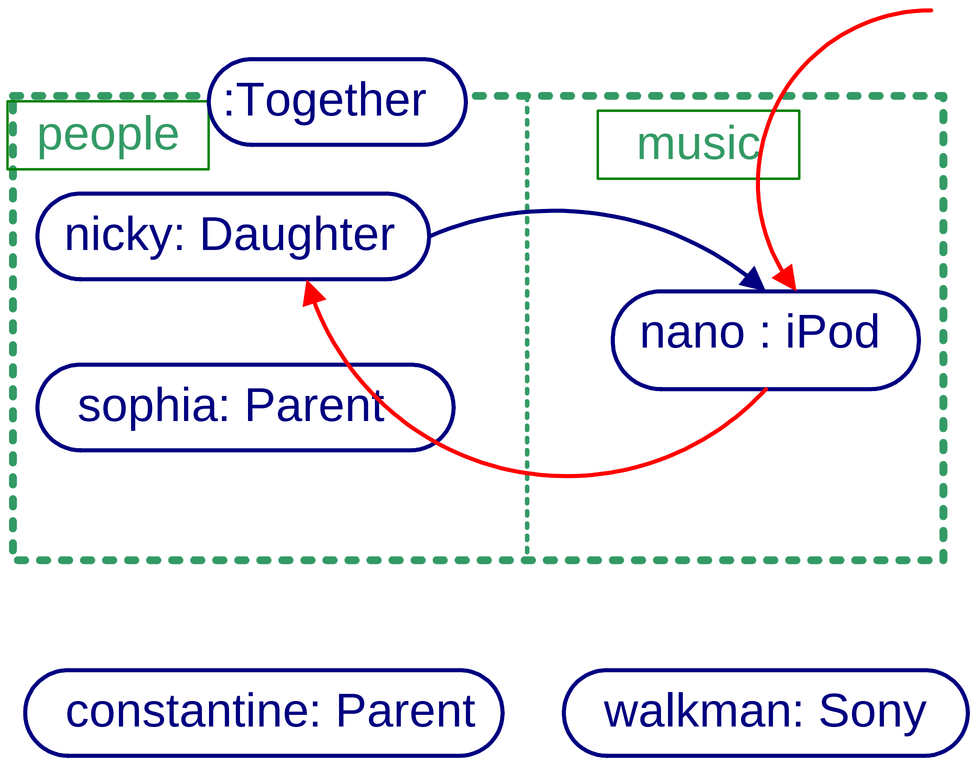
then, only sophia and nicky can manipulate nano.



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 explore

Multiple Ownership

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

Cameron, Drossopoulou, Smith, Noble
OOPSLA 2007

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.

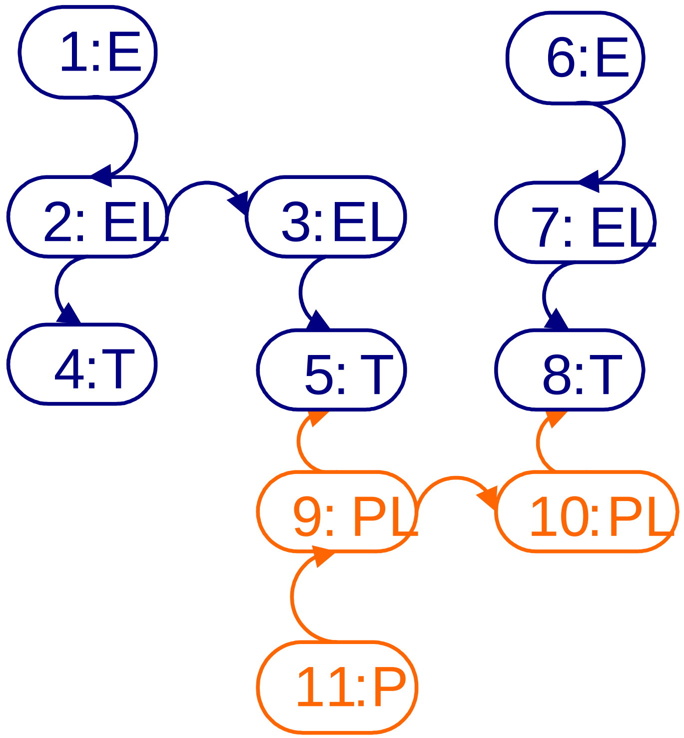
In the code we omit `Node` class.

"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(){ ... }; }
```

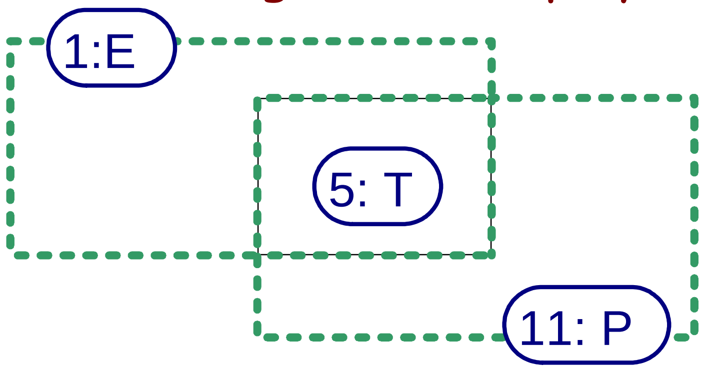


We want:

e1 ∦ e2 ⊢ e1.delay() # e2.OK()

p1 ∦ p2 ⊢ 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**`>`

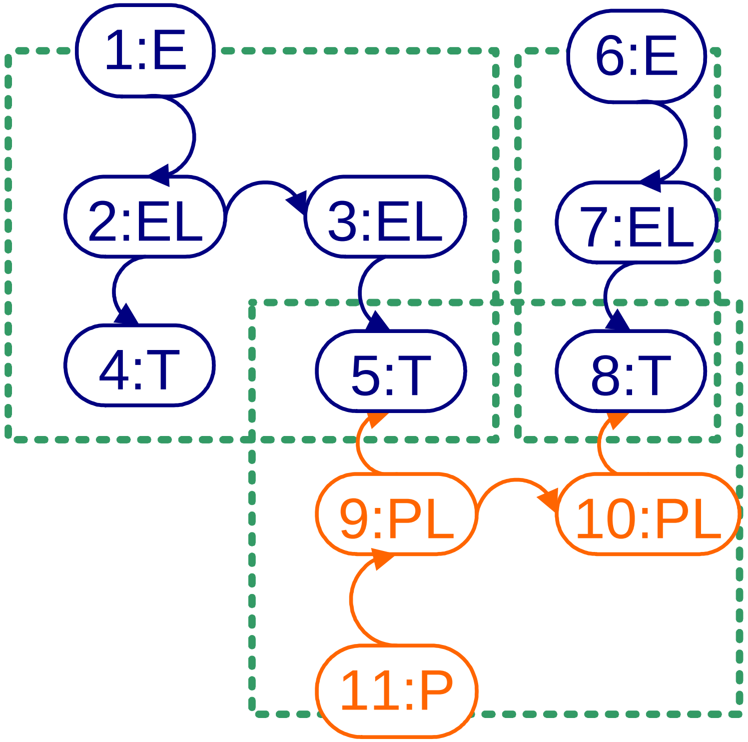
or

`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&any> tasks;
    void delay( ){ ... } }

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

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

class Task<o1>{  ...
    void delay(){ ... }; }
```



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

The meaning of any: the corresponding owner is unknown, but fixed.

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

     …
List<o4,o5&any> l1;
```

```
l1 = new List<o4,o5&o6>;          : OK
l1 = new List<o4,o5&o7>;          : OK
l1 = new List<o4,o5&o7&o8>;       : OK

l1.next;                          : is a List<o4,o5&any>
l1.next.next.next;                : is a List<o4,o5&any>

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

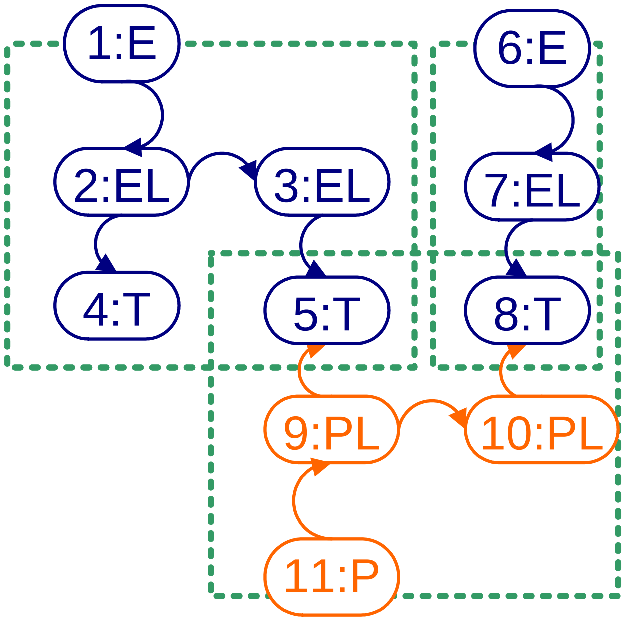
We want to be able to argue:

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

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 « 5

5 « 1

5 « 11

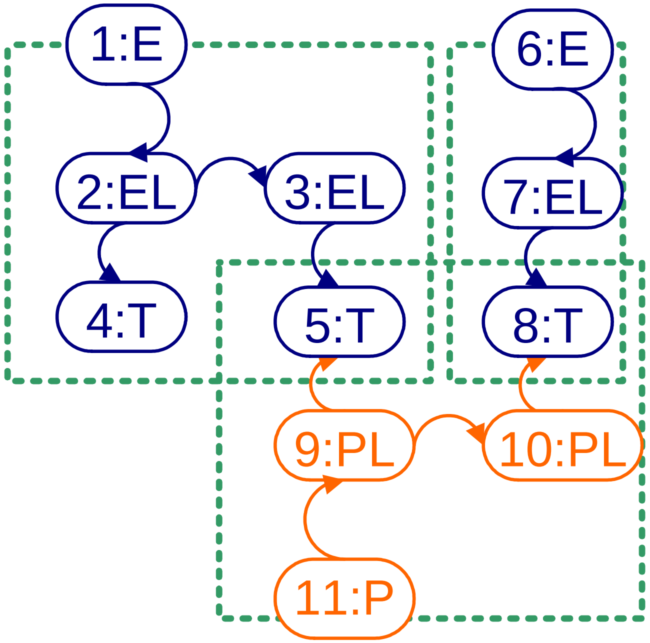
Define *run-time* effects: χ ::= ι | χ`.undr` | χ & χ

meaning:

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

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

$[[\chi \,\&\, \chi']] = [[\chi]] \cap [[\chi']]$



$[[\,\mathtt{1}\,]] = \{ 1 \}$

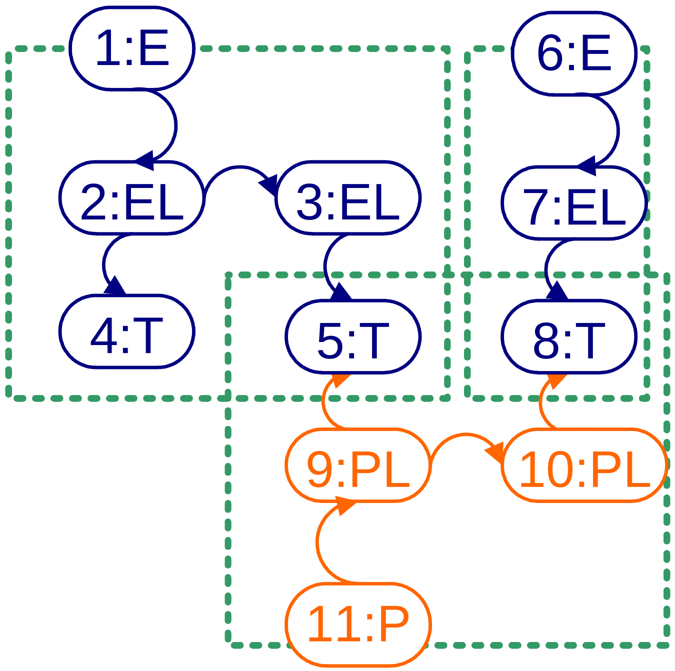
$[[\,\mathtt{1.under}\,]] = \{ 1, 2, 3, 4, 5 \}$

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

$[[\,\mathtt{1.under} \,\&\, \mathtt{6.under}]] = \varnothing$

Define also an effects annotation system, which gives

```
class Employee<o> {
   List<this,this&any> tasks;
   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:

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

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

Theorem:

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

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

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

Because e1 ∦ e2 ⊢ **(e1&any).under** # **(e2&any).under**

we have e1 ∦ e2 ⊢ `e1.delay()` # `e2.OK()`

Similarly, p1 ∦ p2 ⊢ `p1.delay()` # `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**,

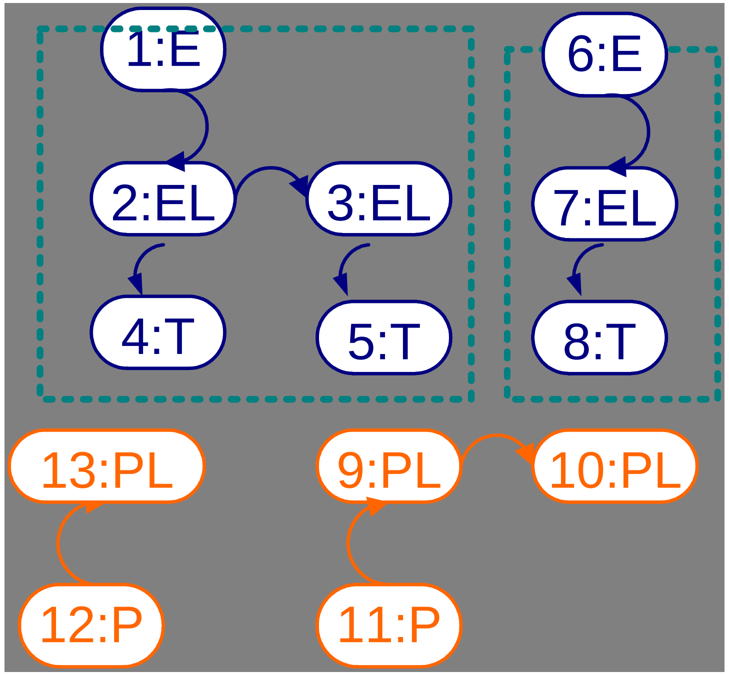
```
class Task<o1,o2:>{  ...   }
class Employee<o:> {
    EList<this:> tasks;
.. }
class EList<o:> {
    EList<o:> next;
   Task<o,any:> task;
... }
class Project<o:> {
    PList<this:> tasks; ...   }
class PList<o:> {
    PList<o:> next;
    Task<any,o:> task;
  ...   }
```

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

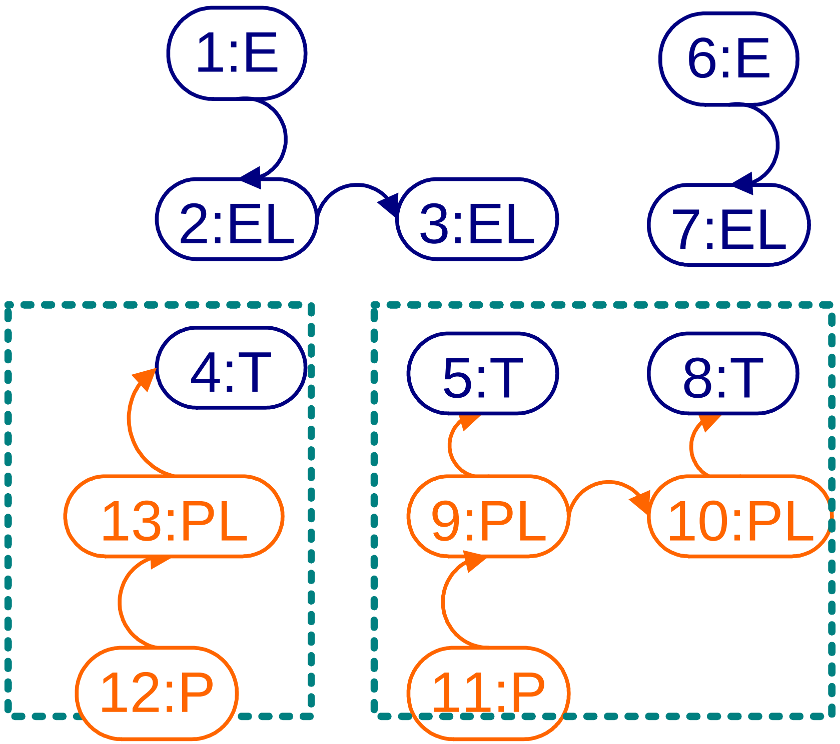
Selected owner **higlighted**

```
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

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

the heap is

$$h = h1 \oplus .... \oplus hn$$

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.

Watch http://slurp.doc.ic.ac.uk/ and OOPSLA 2007 for the paper

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.

Things that have worked for me

- enjoying the work,
- choise what to work on,
- naiveté,
- work on several things in parallel,
- opportunity to revisit a question,
- collaborations,
- teaching,

and also

- family,
- two "blue eyed" decisions

Thank you!