The Benefits of Putting Objects into Boxes

Sophia Drossopoulou
Department of Computing,
Imperial College London
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

```java
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

```java
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.

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

with a possible heap:
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
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

```java
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

```java
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

```
1:E
2:L
3:N
4:N
5:T
6:T
7:E
8:L
9:N
11:T
```
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

```java
class List<o>{ ... }
```
and types mention owners, eg

```
List<this>
```

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

Therefore classes may have context parameters, eg

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

with a possible heap:
Boxes for Object Encapsulation – An Example

"Java + OT" code

```java
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:

```
1:E
2:L
3:N
4:N
5:T
6:T
7:E
8:L
9:N
11:T
```
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() { ... }
}

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

We want to be able to argue for “different” employees $e_1, e_2$:

$e_1 \not\equiv e_2 \vdash e_1.\text{delay()} \# e_2.\text{OK()}$

**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 $\Rightarrow$ independence
Boxes for Object/Property Encapsulation – An Example

**Approach:** we add effects to methods:

```java
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, $e_1.delay() : e_1.under$

Because $e_1 \neq e_2 \vdash e_1.under \# e_2.under$

we have $e_1 \neq e_2 \vdash e_1.delay() \# e_2.OK()$
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 individually.

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
<table>
<thead>
<tr>
<th></th>
<th>owner is ...</th>
<th>owner as dominator?</th>
<th>benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Confined types</strong></td>
<td>package - <em>static</em> number of owners</td>
<td>yes</td>
<td>object encapsulated in package</td>
</tr>
<tr>
<td><strong>Object Encapsulation</strong></td>
<td>an object</td>
<td>yes</td>
<td>object encapsulated in objects, scoped memory, visualization, independence</td>
</tr>
<tr>
<td><strong>Locking</strong></td>
<td>object or thread, holds “logic lock” to owned objects</td>
<td>no; no nesting</td>
<td>guarantee race-free, or atomicity</td>
</tr>
<tr>
<td><strong>Universes/Boogie</strong></td>
<td>an object; owner’s properties depend on owned objects’ state - owners may change</td>
<td>no; modifier owners may change</td>
<td>modular verification</td>
</tr>
</tbody>
</table>
However ...
The nano is mine

No, it is mine

OK, let us share it!
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:

- nicky: Daughter
- sophia: Parent
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.*

```java
class Daughter {...
    rep iPod nano;
    ...
}
class Parent {...
    readonly iPod nano;
    ...
}
```

then, **sophia** can listen to the **nano**.
However, then, sophia cannot switch the nano on or off!

```
nicky: Daughter

nano: iPod

readonly

sophia: Parent
```
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.
However, what if sophia wanted to
• share the nano with nicky, and also
• share the walkman with constantine?

nicky: Daughter
sophia: Parent
constantine: Parent
nano: iPod
walkman: Sony

people: Together
music:
Common Ownership – The Lu & Potter Way

... more at ECOOP’06
Common Ownership – so far

slightly relax an underlying, unique ownership hierarchy

Instead, today we explore

**Multiple Ownership**

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

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

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

class Task {
    ...
    void delay() { ... };
}

class Project {
    PList tasks;
    void delay() { ... } }

class PList {
    PList next;
    Task task;
    void delay() { ... } }
```
Multiple Ownership – An Example

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

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

class Task {
    ...
    void delay(){} ...
}

class Project {
    PList tasks;
    void delay(){} ...
}

class PList {
    PList next; Task 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.

\[ \text{task 5 is owned by Employee 1, and Project 11.} \]

We allow many owner parameters, as well as context parameters, i.e.:

```
class A<o_1, \ldots, o_n : p_1, \ldots, p_m> \{ ... \}
```

where \( o_1, \ldots, o_n \) owner parameters, and \( p_1, \ldots, p_m \) context parameters.

Thus, our earlier \( \text{class } A<o_1, p_2, \ldots, p_n> \) corresponds, now to \( \text{class } A<o_1:p_2, \ldots, p_n> \)

In a type, we say \textbf{any}, when actual owner/context unknown (cf \texttt{readonly}).
Multiple Ownership

class Employee<o:> {
    EList<this:> tasks;
    void delay() { ... } }

class EList<o:> {
    EList<o:> next;
    Task<o,any:> task;
    void delay() { ... } }

class Task<o1,o2:>{ ... 
    void delay() { ... } }

class Project<o:> {
    PList<this:> tasks;
    void delay() { ... } }

class PList<o:> {
    PList<o:> next;
    Task<any,o:> task;
    void delay() { ... } }
The meaning of \texttt{any}: the corresponding owner/context is unknown, but fixed.

```java
class EList<o:> { 
    ...
    Task<o,any:> task;
}
final Employee<p:> el;
final Project<q:> p1; final Project<r:> p2;
EList<el:> l1;
l1.task := new Task<el,p1>; : OK
l1.task := new Task<el,p2>; : OK
EList<any:> l2;
l2.task : Task<any,any>
l2.task := new Task<any,any>; : TYPE ERROR
```
We want to be able to argue:

\[ e_1 \parallel e_2 \vdash e_1.\text{delay()} \# e_2.\text{OK()} \]

We first define when an object is “inside” another object, i.e. \( \ll \ll' \) as the minimal reflexive, transitive relation, such that

if one of the owners of \( \ll \) is \( \ll' \) then \( \ll \ll' \)

Therefore

\[ 5 \ll 5 \]
\[ 5 \ll 1 \]
\[ 5 \ll 11 \]
Define run-time effects: \( \chi ::= l | c^{l_1,..,l_n} | \chi.\text{undr} | \ldots \)

meaning:

\[
[[l]] = \{l\} \\
[[c^{l_1,..,l_n}]] = \{l | l \text{ dyn. type } c^{l_1',..,l_n'} \text{ and } l_i' \ll l_i\} \\
[[\chi.\text{undr}]] = \{l | l \ll [[\chi]]\}
\]

\[
[[\text{Task}<1,11>]] = \{5\} \\
[[\text{Task}<1,\text{any}>]] = \{4,5\} \\
[[\text{Task}<11,\text{any}>]] = \emptyset
\]

\[
[[1]] = \{1\} \\
[[1.\text{under}]] = \{1,2,3,4,5\}
\]
Define static effects \( \phi ::= x \mid c^{x_1, \ldots, x_n} \mid \phi.\text{undr} \mid \ldots \)

Define also a static effects system, which gives

```java
class Employee<o:>
{
    ...
    void delay() this.undr[..]}
}
class EList<o:>
{
    ...
    void delay() o.undr[..]}
}
class Task<o1,o2:>
{
    ...
    void delay() this.undr[..]}
}
class Project<o:>
{
    ...
    void delay() this.undr[..]}
}
class PList<o:>
{
    void delay() o.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:**

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

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

**Theorem:**

$$\begin{align*}
\Gamma \vdash_{rd} e : \phi_1 & \quad \Gamma \vdash_{wr} e : \phi_2 \\
\Gamma \vdash s, h & \\
\vdash_{rd} e, s, h \sim v, h' & \\
\end{align*} \quad \Rightarrow \quad \begin{cases} 
\begin{align*}
\ h & = [[\phi_1]]_{s,h} \ast h_2 \\
[[\phi_1]]_{s,h} & = [[\phi_2]]_{s,h} \ast h_3 \\
\ h' & = h'' \ast h_3 \ast h_2 \\
\ e, s, [[\phi_2]]_{s,h} & \ast h_3 \sim v, h'\ast h_3 \\
\text{for some} \ h_2, h_3, h''
\end{align*}
\end{cases}$$
Thus,\[ e_1.\text{delay()} : e_1.\text{under} \]
\[ e_2.\text{OK()} : e_2.\text{under} \]

Because \[ e_1 \upharpoonright e_2 \vdash e_1.\text{under} \# e_2.\text{under} \]
we have \[ e_1 \upharpoonright e_2 \vdash e_1.\text{delay()} \# e_2.\text{OK()} \]

Similarly, \[ p_1 \upharpoonright p_2 \vdash p_1.\text{delay()} \# p_2.\text{OK()} \]

😊
Can I avoid multiple owners?

Single owners (usually) have the owners as dominators property. Can I replace multiple owners by single owners representing tuples of owners? i.e., instead of have pairs of owners objects
Can I avoid multiple owners? - not really

Thus scheme would require many more “ghost” objects, and much more “bookkeeping”.

Furthermore

- To whom does 1 belong?
- Accessing 1 would break the owners as dominators property.
- How do I delay an employee (say 1) atomically?
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”, $P_i$, 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”, $P_i$, 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 highlighted,

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 highlighted, // 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 highlighted

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 highlighted,
// 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;
   ... }

Diagram:

- 1:E
- 2:EL
- 3:EL
- 4:T
- 5:T
- 6:E
- 7:EL
- 8:T
- 9:PL
- 10:PL
- 11:P
- 12:P
- 13:PL
Preserving owners as dominators – partly

Aside: I have been tackling this problem (independence of actions and assertions in the presence of “overlapping topologies”) unsuccessfully by filtering out fields in and off for the last two years. Multiple owners was the missing link.

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 \( Pi \).

\[ f1 \oplus f2 = f0 * f3 * f4 \]

where \( f1 = f0 * f3 \) and \( f2 = f0 * f4 \)
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/](http://slurp.doc.ic.ac.uk/) 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, e.g., 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.
Thank you!