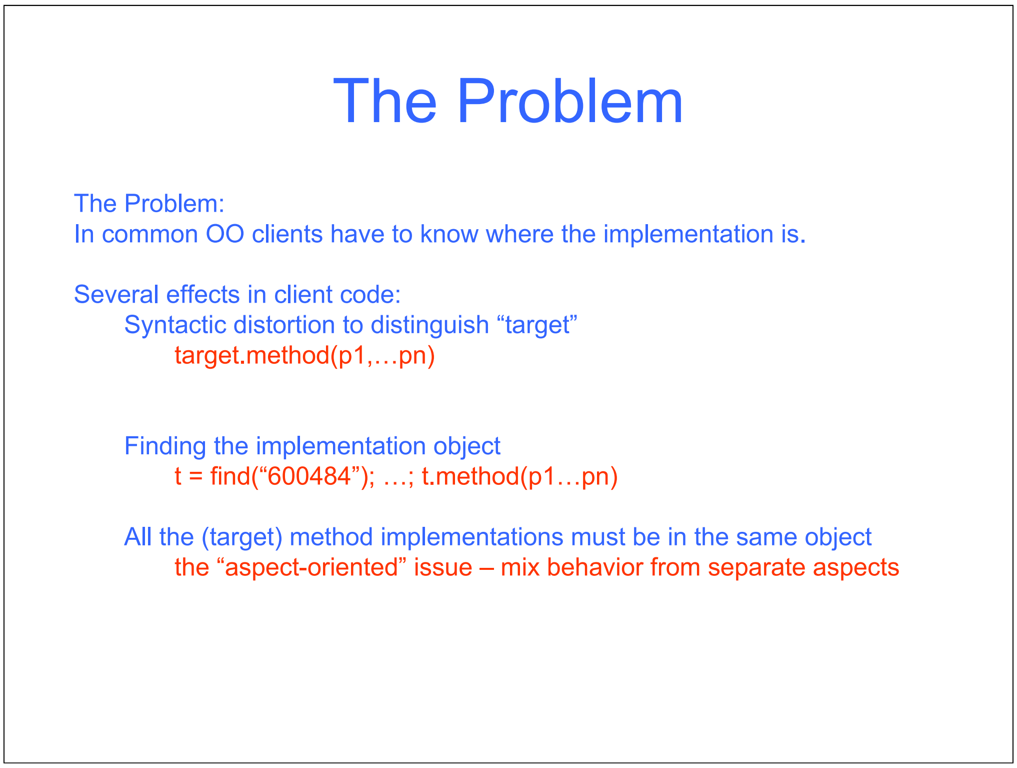
Assurances vs. Capabilities as a Basis for Dispatch

William Harrison
Trinity College Dublin



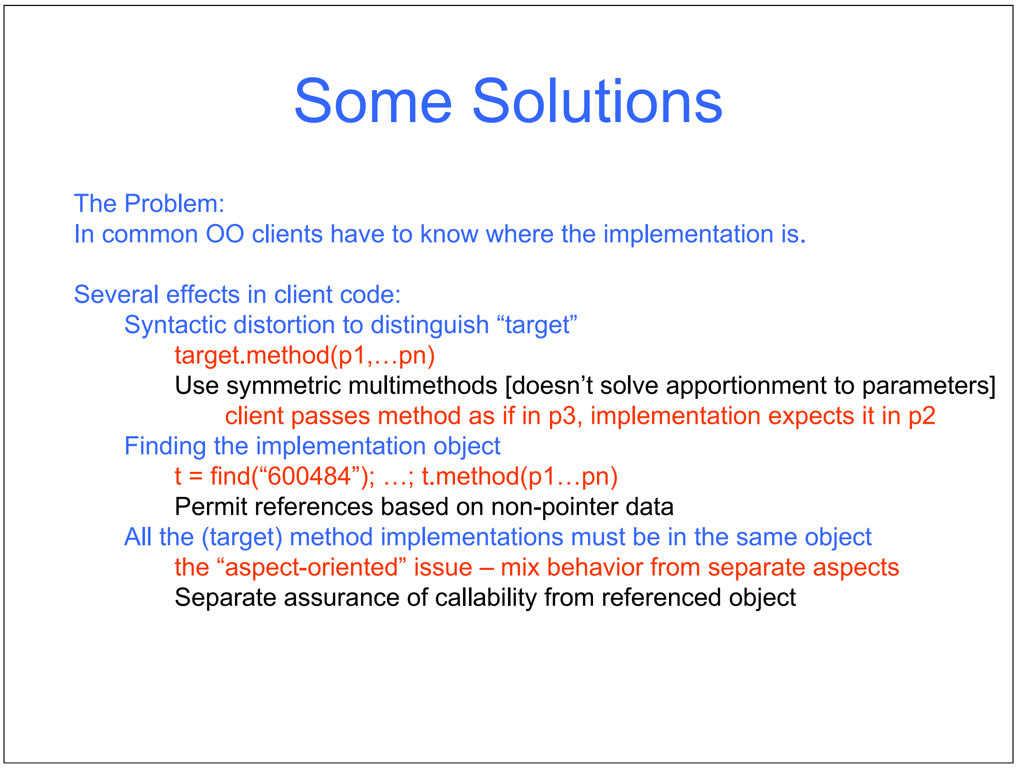
The Problem

The Problem:
In common OO clients have to know where the implementation is.

Several effects in client code:
- Syntactic distortion to distinguish “target”
  - target.method(p1,…pn)
- Finding the implementation object
  - t = find(“600484”); …; t.method(p1…pn)
- All the (target) method implementations must be in the same object
  - the “aspect-oriented” issue – mix behavior from separate aspects

It is desirable for a client to be unaware of the structure of the services implementing its calls because this allows the cliend to be used with more different service implementation.

Common object-oriented languages employ a structure for method-call in which the client finds a “target” object and sends a method call to that object. object to have the service performed. The syntax reflects the knowledge of which object implemented the method. In the first instance the target is given a distinct syntactic position in the call. In the second instance the client must contain code to find the target object. The code for doing this can be located far away from the call. In fact, it may have occurred long before with the result stored in a state variable for the interim. In the third instance, the client assumes that all of the methods described by the interface used for the target are implemented in one object. This rules out implementations in which they reside in several.



Some Solutions

The Problem:
In common OO clients have to know where the implementation is.

Several effects in client code:

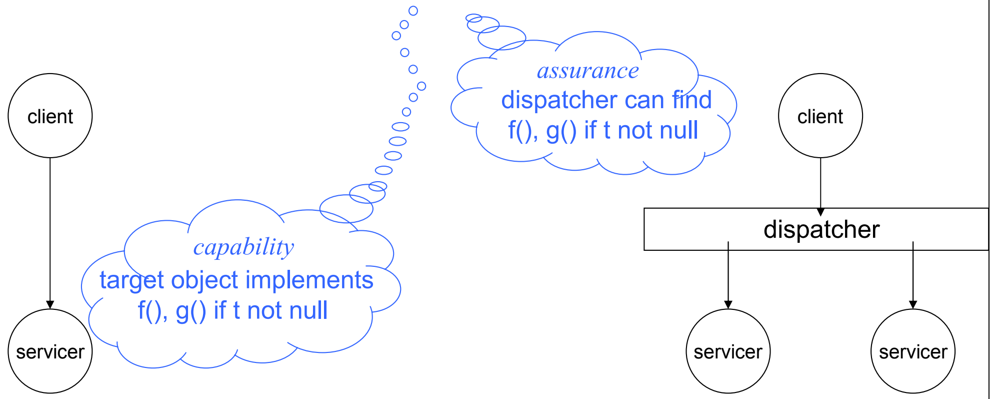
- Syntactic distortion to distinguish "target"
  - target.method(p1,…pn)
  - Use symmetric multimethods [doesn't solve apportionment to parameters]
    - client passes method as if in p3, implementation expects it in p2
- Finding the implementation object
  - t = find("600484"); …; t.method(p1…pn)
  - Permit references based on non-pointer data
- All the (target) method implementations must be in the same object
  - the "aspect-oriented" issue – mix behavior from separate aspects
  - Separate assurance of callability from referenced object

There are some solutions that can be applied to these problems. The local syntactic distortion can be alleviated by using symmetric multimethods. But this still leaves what might be called the "apportionment problem". The apportionment problem arises when the client assumes that some capability is passed by one parameter, say p3, while the implementation expects the assurance to be passed via p2. We can avoid having the client find the object by allowing it to use the non-pointer data directly instead of requiring it to use the data to get a pointer first. And we can remove the presumption of which object's pointer carries the implementation by separating the assurance from the object referenced.

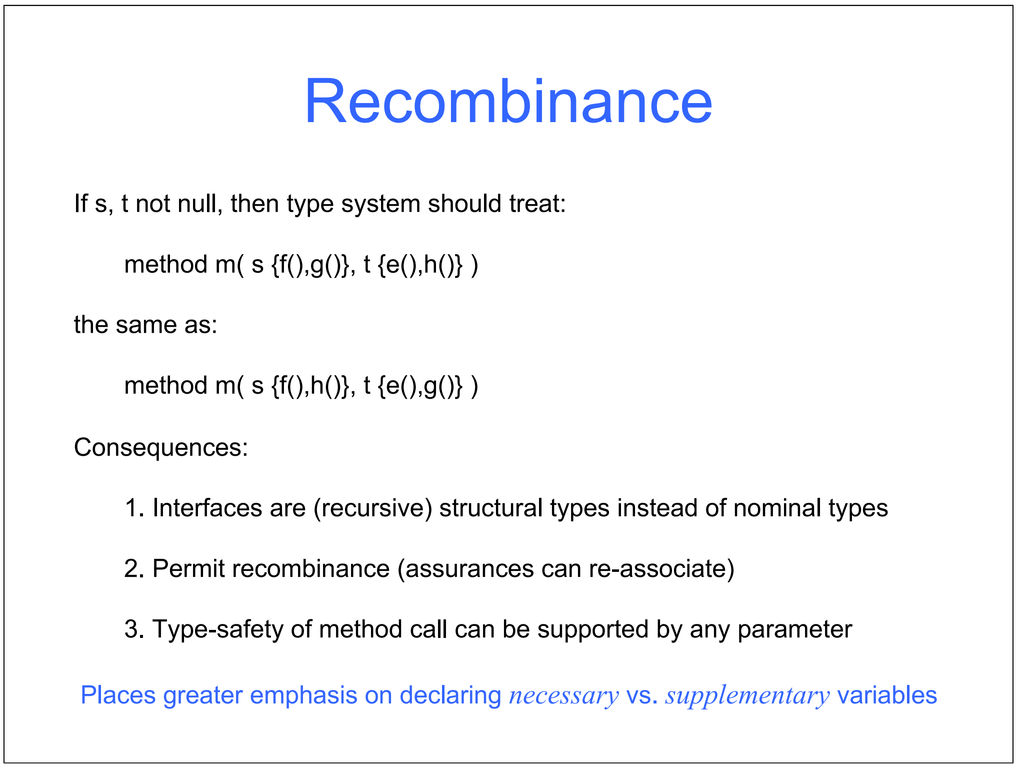
Some Solutions

Separate assurance of callability from referenced object

Change meaning of declaration: {f(); g();} t



We remove the presumption that the reference tells about the target object by slightly changing the meaning of a declaration. When we declare t as pointing, if it’s not null, to an object that implements f() and g(), we say that, if not null, it points some objects and guarantees that the dispatcher can find the methods f() and g().



Recombinance

If s, t not null, then type system should treat:

> method m(s {f(),g()}, t {e(),h()})

the same as:

> method m(s {f(),h()}, t {e(),g()})

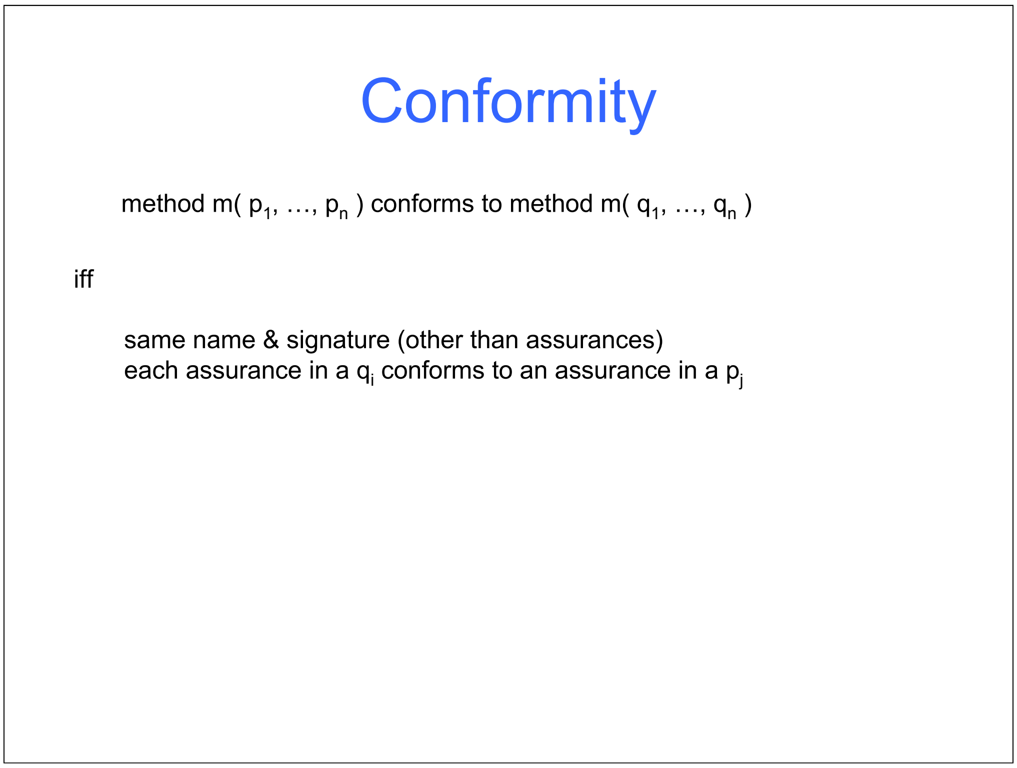
Consequences:

1. Interfaces are (recursive) structural types instead of nominal types
2. Permit recombinance (assurances can re-associate)
3. Type-safety of method call can be supported by any parameter

Places greater emphasis on declaring *necessary* vs. *supplementary* variables

An important difference that this change in interpretation yields can be highlighted when we have two references, BOTH of which are not null. In this case we know that all the indicated methods can be safely called. The fact that they are all true means we can recombine those assurances, assign them differently among other variables. For example knowing that h() and g() can be called means that they are assured with respect to ANY variable available for assignment at the time that fact is known.

This effect, called recombinance, points us in the direction of using recursive structural comparisons for interfaces, instead of simply comparing them by name, and allows the assertion that a method call is safe if it is assured by any of the parameters, and not just by a distinguished target parameter. But exploitation of recombinance places greater emphasis on knowing declaratively that parameters may not be null, to enable their recombination.



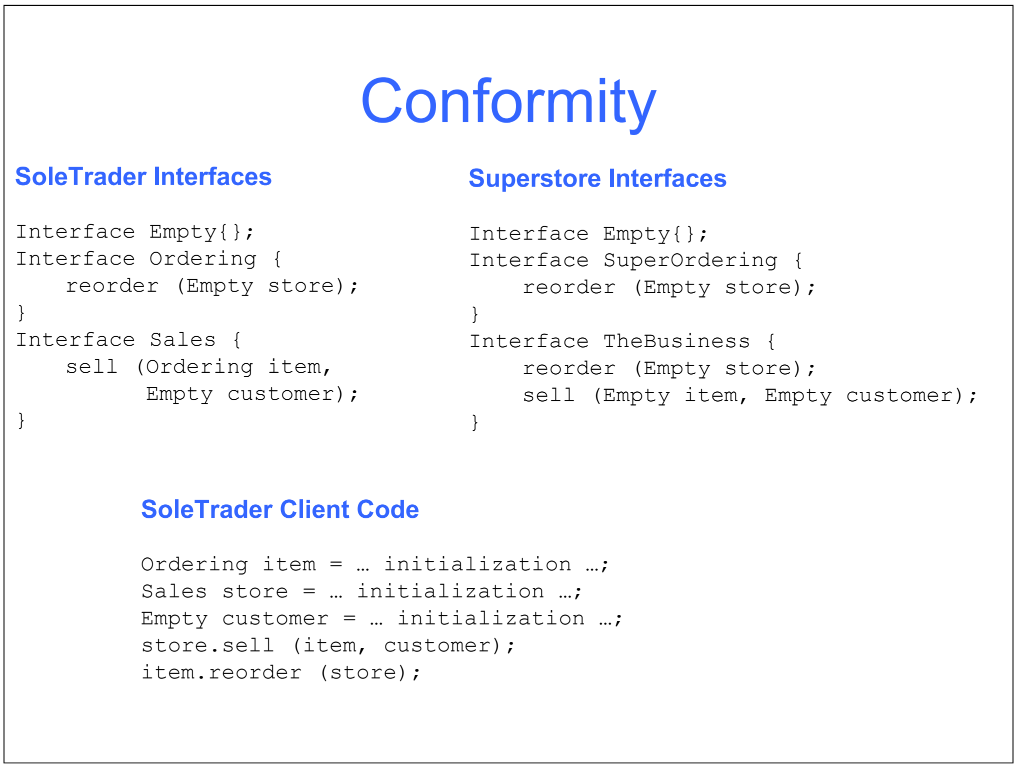
Conformity

method m(p_1, …, p_n) conforms to method m(q_1, …, q_n)

iff

same name & signature (other than assurances)
each assurance in a q_i conforms to an assurance in a p_j

This gives us a new conformance rule, which we can illustrate with a small example.



Conformity

SoleTrader Interfaces

```
Interface Empty{};
Interface Ordering {
    reorder (Empty store);
}
Interface Sales {
    sell (Ordering item,
          Empty customer);
}
```

Superstore Interfaces

```
Interface Empty{};
Interface SuperOrdering {
    reorder (Empty store);
}
Interface TheBusiness {
    reorder (Empty store);
    sell (Empty item, Empty customer);
}
```

SoleTrader Client Code

```
Ordering item = … initialization …;
Sales store = … initialization …;
Empty customer = … initialization …;
store.sell (item, customer);
item.reorder (store);
```

Consider the case of a small store, a sole trader, which implements its software in the way implied by the sole-trader interfaces at the upper-left of this diagram, and that some client written for it as shown in the lower part of this diagram. If the sole trader is bought-out by a superstore which has used instead the interfaces at the upper right, its client code still matches those interfaces. This is not a simple effect arising from the fact that the superstore put all its functions in the same object which implements both interfaces. It arises because the declarations of the “sell” methods are, in fact, different, as illustrated by the highlighted first parameter to “Sell” but arises because of the changed conformance rules.