

TriReme

**Building models from
Generic Frameworks**

Alan Cameron Wills

Building Models from Generic Frameworks

Alan Cameron Wills

Trireme International Ltd

<http://www.trireme.com>

alan@trireme.com

0161 225 3240

Reuse is why we like Objects

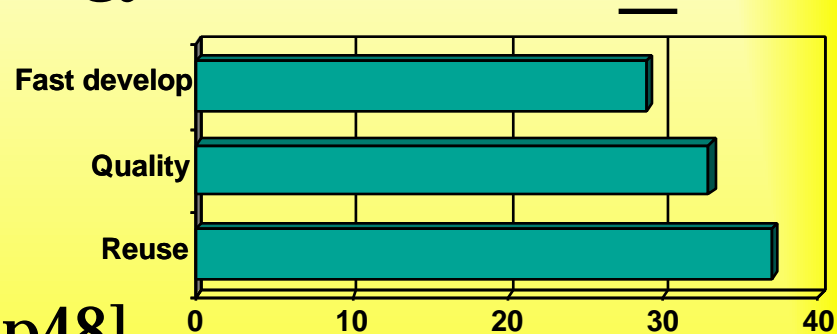
■ “What is your principal motivation for taking up object technology?”

● Fast development?

● Better quality?

● Reuse?

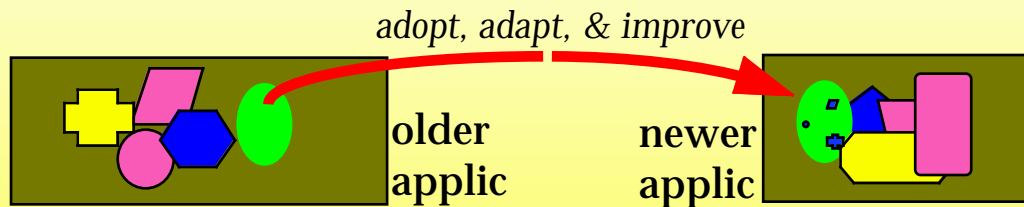
[Unix News May 95 p48]



■ So are you getting it ...?

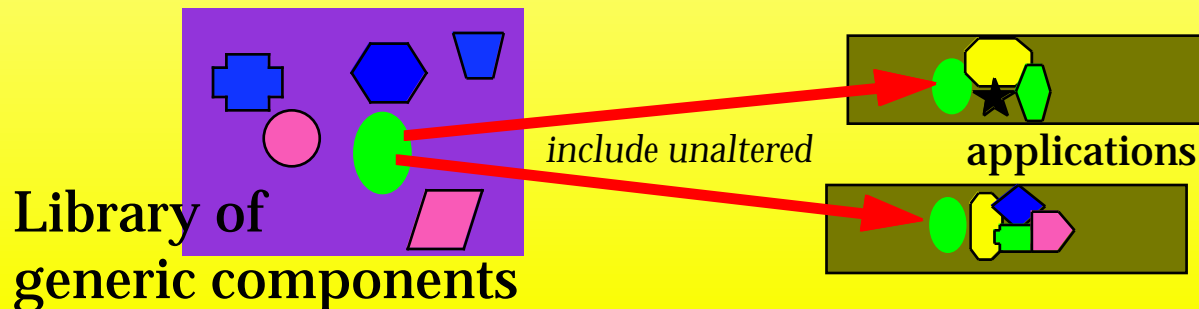
Reuse is ...

■ Cutting and pasting (“cloning”)?



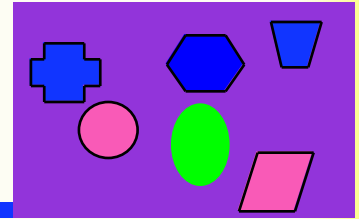
OK at first,
but no help
with change

■ Imported components



Better at
dealing with
change —
though components
in Library must
be generalised

So what's in the Library?



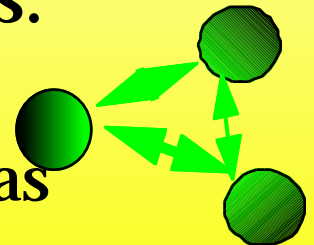
■ Generic components:

- Classes; Class templates
- Collaboration frameworks
- Specifications
- Patterns
- Application frameworks
- Operating Systems, Databases, GUIs
- Standards
- Plans
- General advice

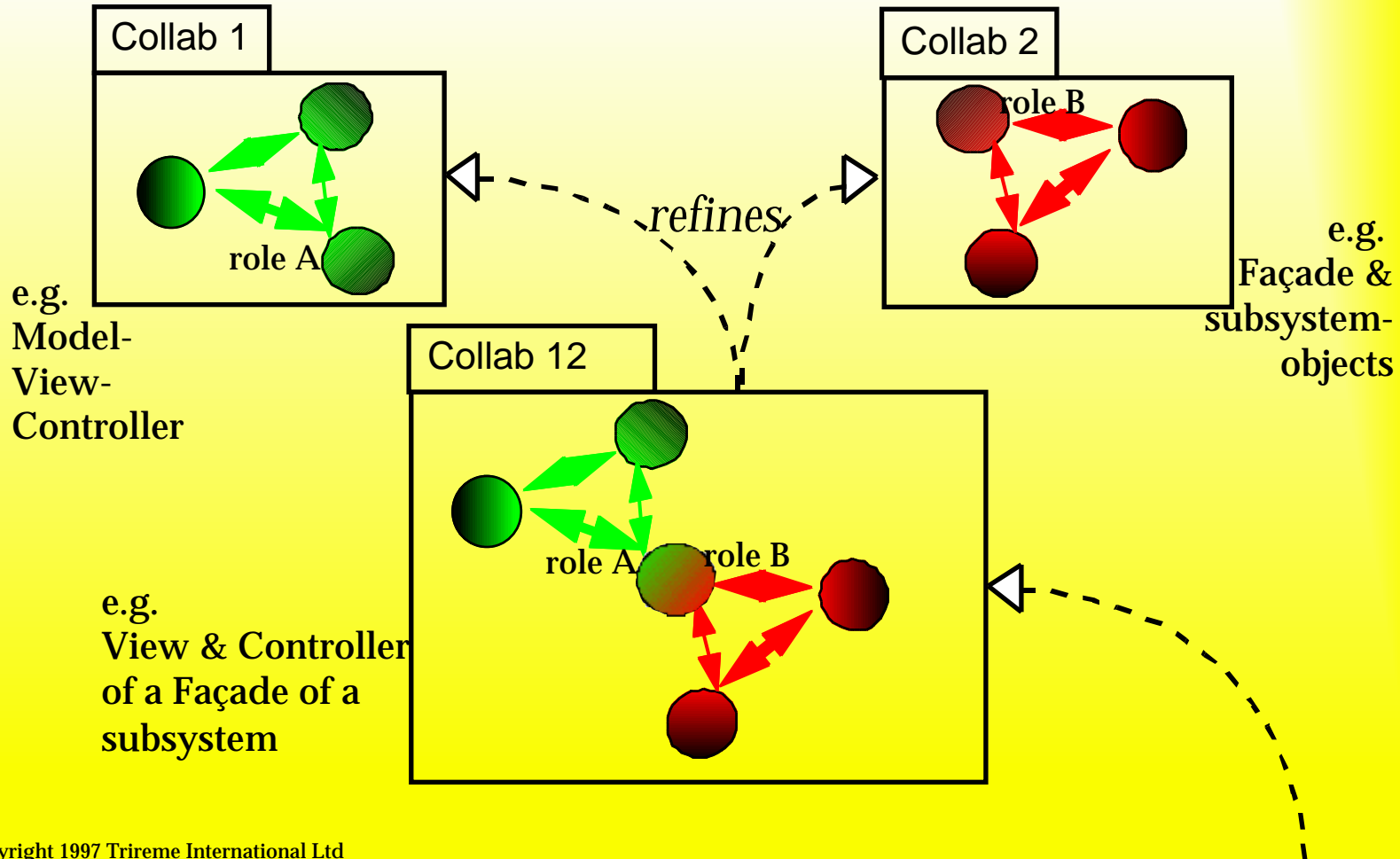
any chunk
of work

Library asset = any generic chunk of work

- Any chunk of work that can be generalised and combined with other such chunks
- In OOD, the focus of design work is on collaborations and responsibilities:
 - As library assets, collaborations are at least as useful as classes

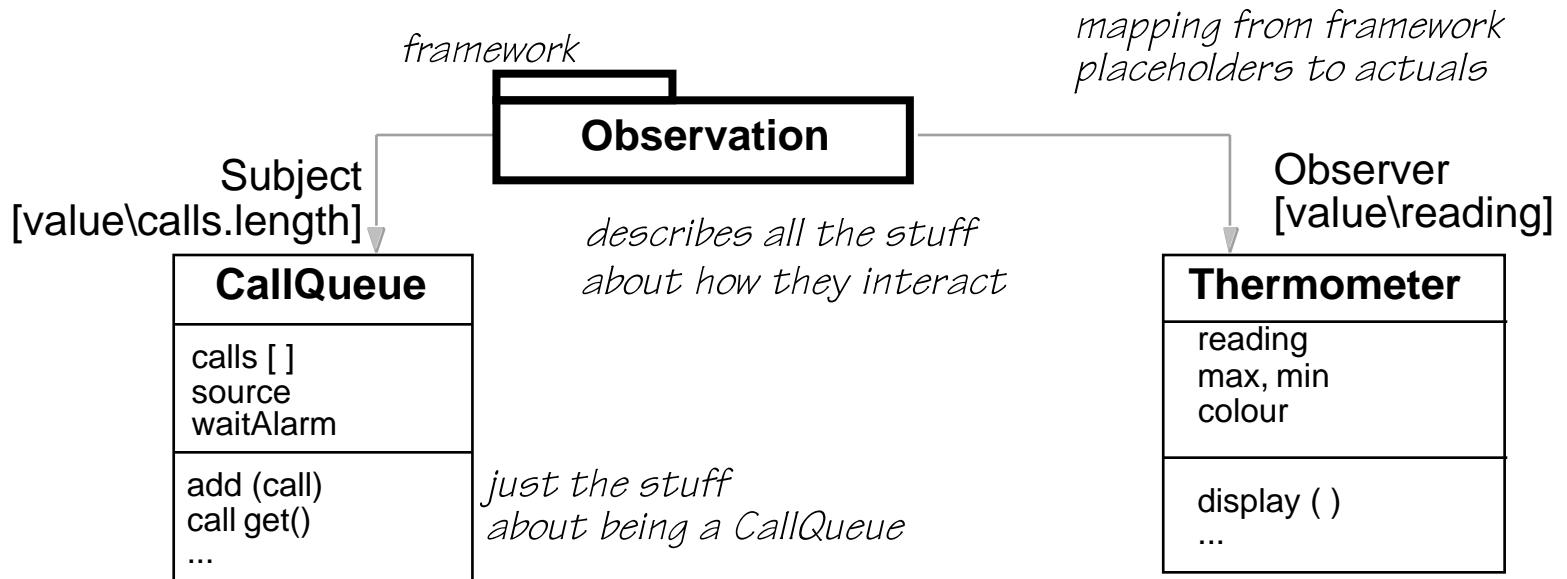


Collaborations as reusable chunks



Framework application

You draw an application of a framework:



Tool works out complete model by applying framework(s) to classes

Agenda

- **Drawing & composing generic partial models**
 - Static models — no interactions
 - Relation to subtypes
- **Specifying collaborations**
 - Interactions happening
- **Composing collaborations**
- **Conclusions, if any**
- **Thanks to John Cameron / OT96**

Generalisation

■ ?

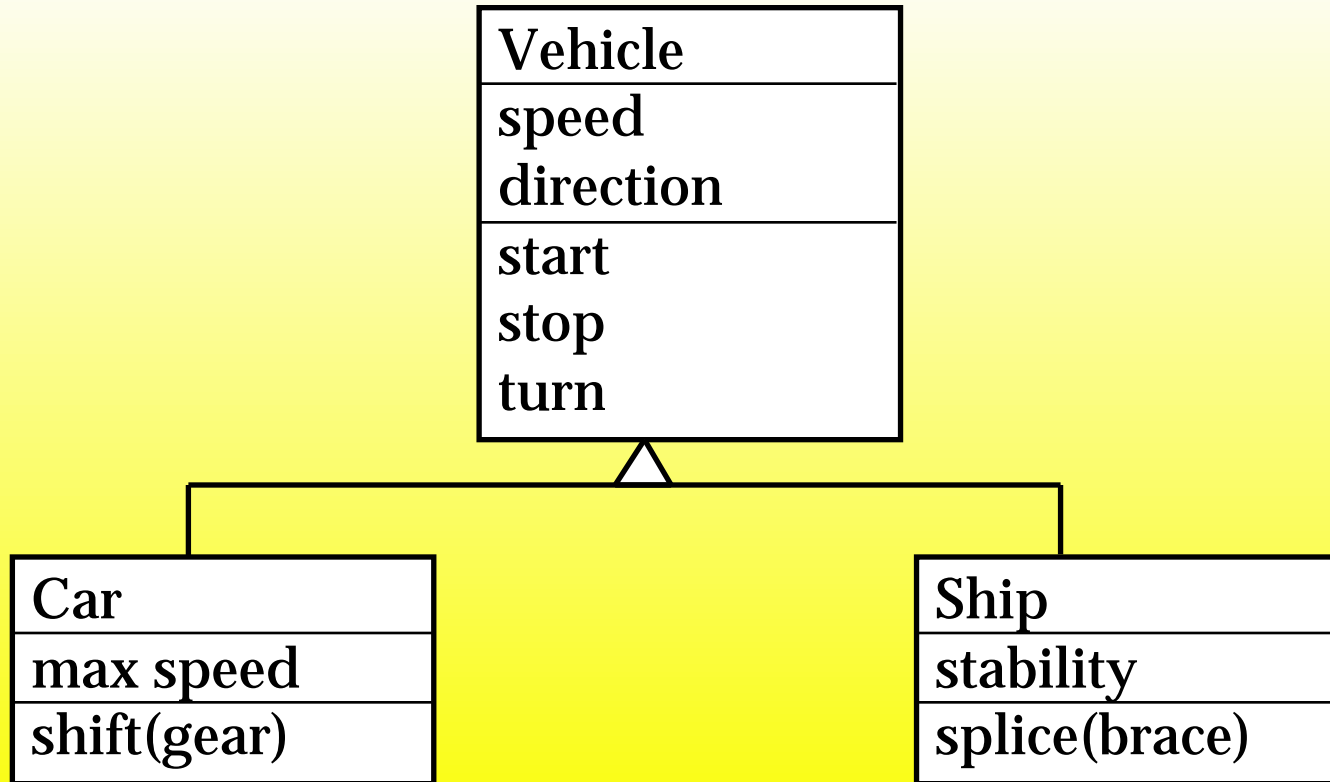
Car
speed direction max speed
start stop turn shift(gear)

attributes

operations

Ship
speed direction stability
start stop turn splice(brace)

Generalisation



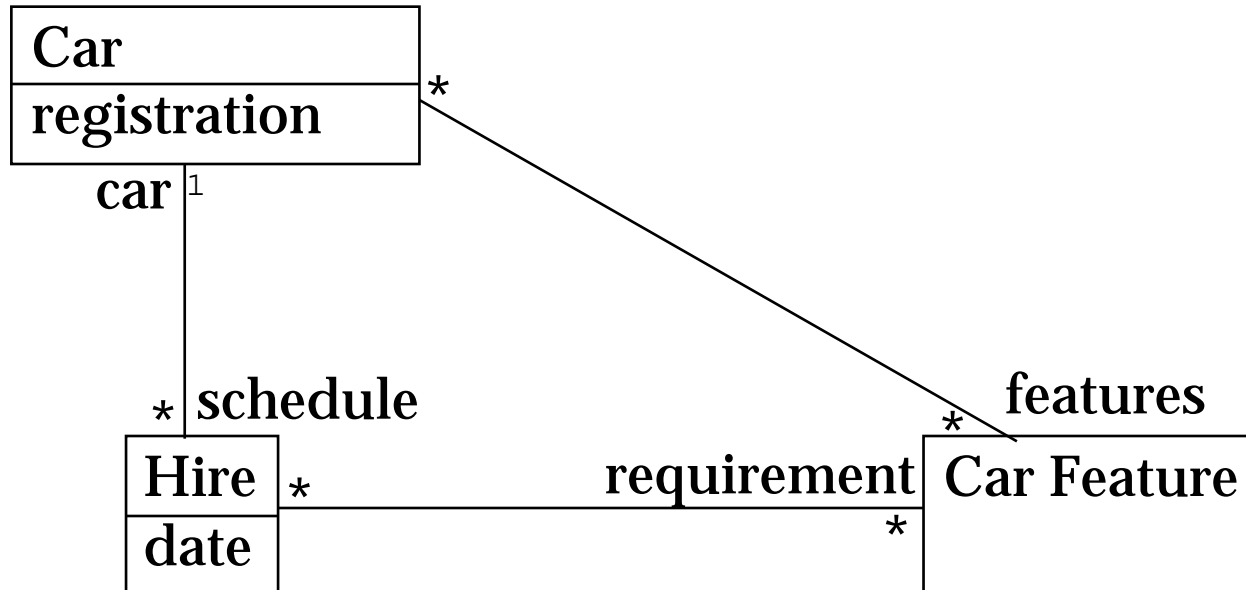
Into the problem:

1. A small model.

- A car hire company rents cars for individual days. Cars must not be double-booked, and the car allocated to a specific hire must meet the requirements of the customer. Requirements are chosen from a set including things like '4 door', 'big boot', 'drinks cabinet', 'auto-shift', etc.
- Never mind the operations for now; just draw a type diagram and invariants.

My solution

Day Hire Cars



-- No Car is double booked --

Car :: forall h1, h2 in schedule, h1 != h2 => h1.date != h2.date

-- requirements of every Hire are met --

Hire :: requirement subsetOf car.features

• Precise notation complements and disambiguates

2. Another model

- The *Levenshulme Whistle* has a fixed layout. Reporters are required every week to fill each slot with a story. Each slot has a fixed set of characteristics to which the story in it must conform:

Levenshulme Whistle

photo
hospital;
scandal
national;
politician;
scandal

page 1

Rescued
cat
local;
mugg-foreign;
ing disaster

page 2

female	clergy;
	sex
Local; sex;	
politician	

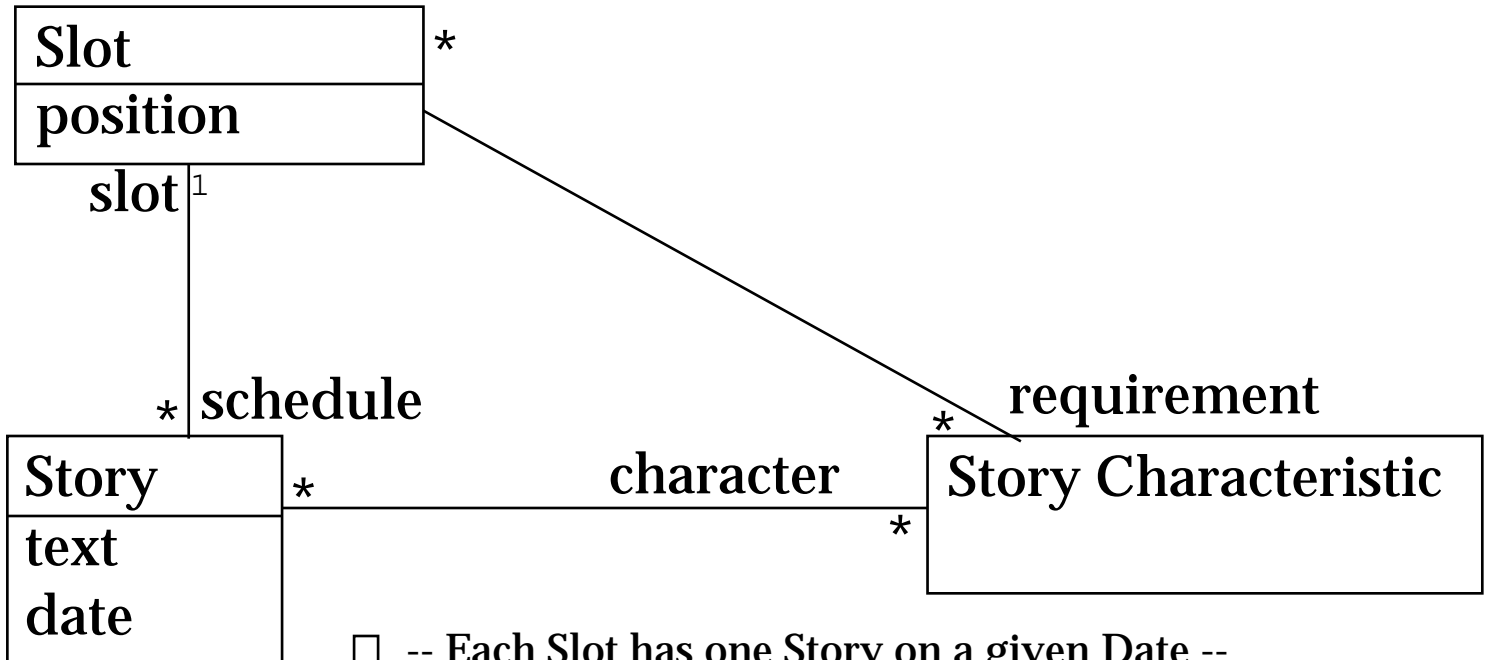
page 3

celeb ;	
charity;	youth;
local	charity
local;	
car	local;
crash	conviction

page 4

etc.

Levenshulme Whistle



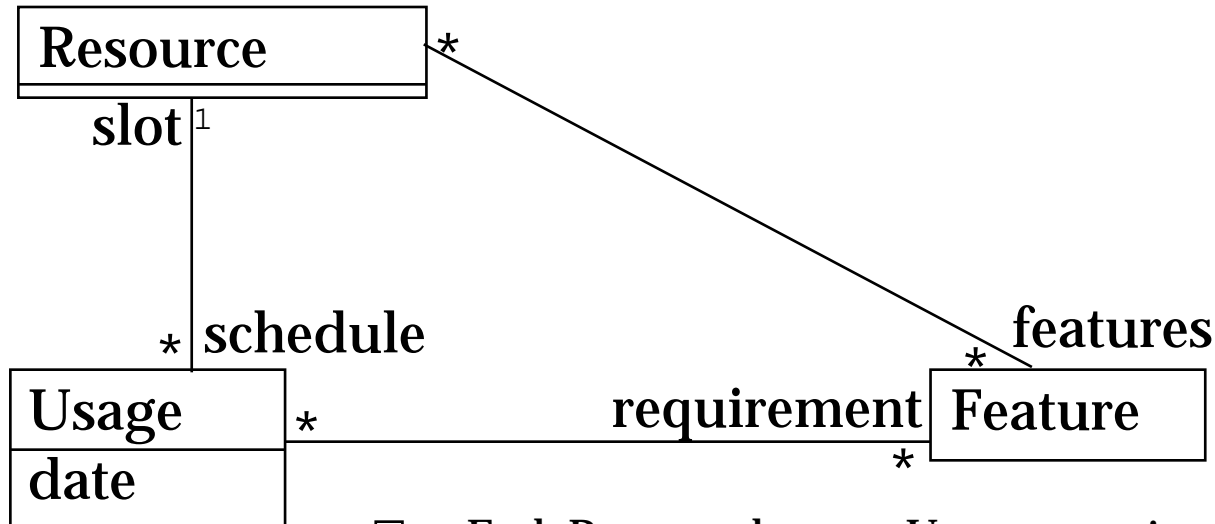
- -- Each Slot has one Story on a given Date --
Slot :: **forall** story1, story2 in schedule,
story1 != story2 => story1.date != story2.date
- -- requirements of every Slot are met --
Slot :: requirement **subsetOf** slot.character

• Generalise ; apply to each case ...

?

A Generalisation (?)

Resource Allocation

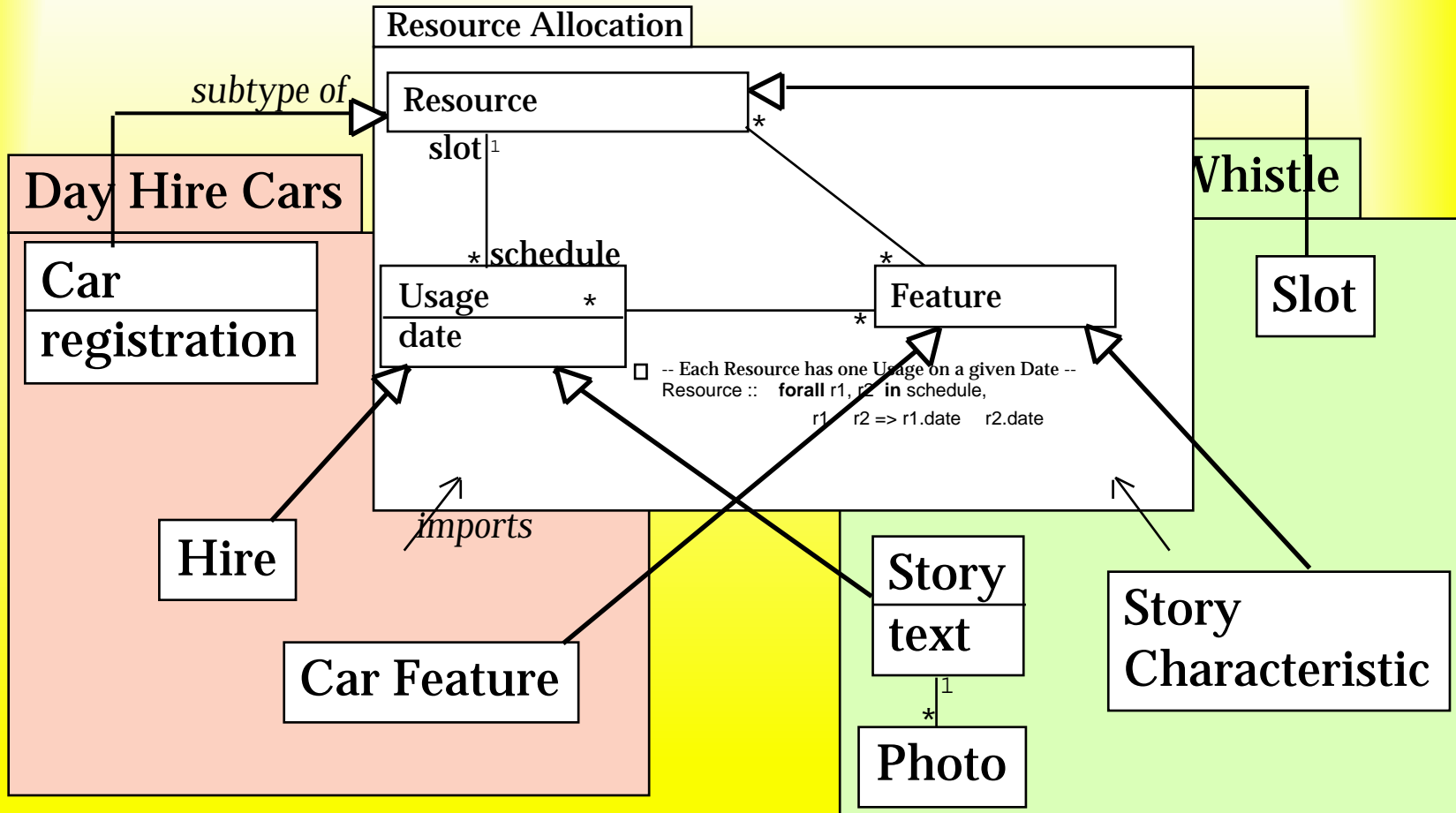


- Each Resource has one Usage on a given Date --
Resource :: forall r1, r2 in schedule,
r1 != r2 => r1.date != r2.date

• Apply to each case... ?

The Wrong Specialisation ?

- 2 problems ?



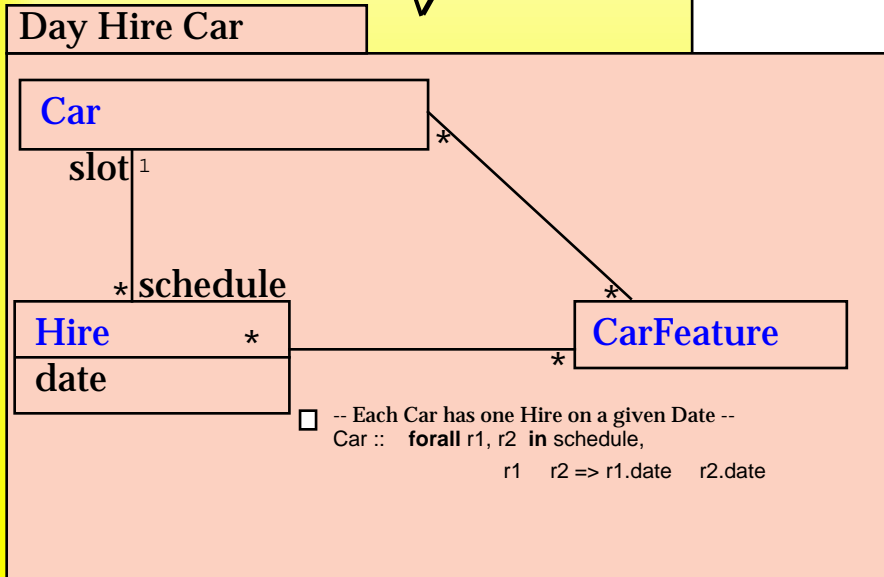
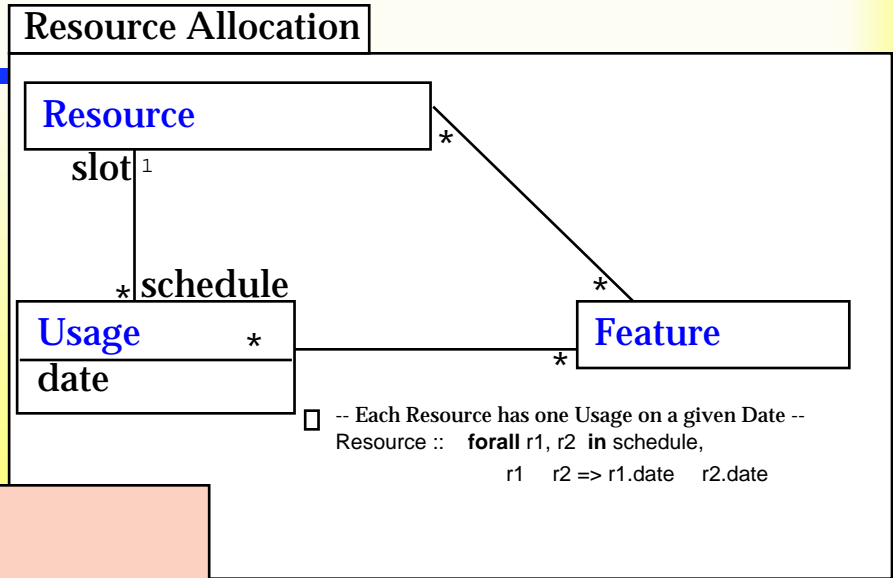
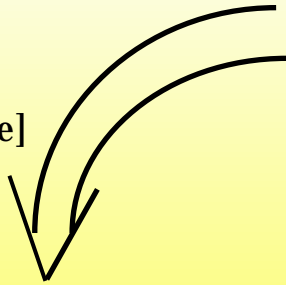
Subtyping inappropriate

- Can you Hire a Slot?
Can you assign a Car to a Story?
Can a Car have Sex, or a Story have luggage space?
- We talk about the features of a Car, but the character of a Story.
The relationships are homomorphic, but differently-named.

Substitution the solution

macro import with renaming

[Resource \ Car,
Usage \ Hire,
Feature \ CarFeature]



DayHireCar =
ResourceAllocation
[Resource \ Car,
Usage \ Hire,
Feature \ CarFeature]

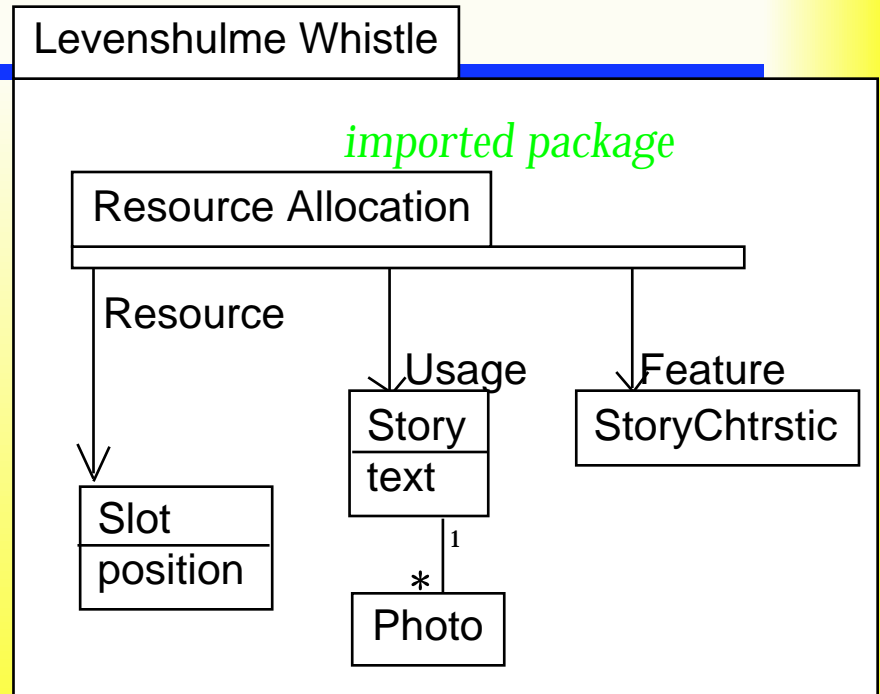
Levenshulme Whistle =
ResourceAllocation
[Resource \ Slot,
Usage \ Story,
Feature \ StoryCharacteristi

Macro import

text version

```
DayHireCar =  
( ResourceAllocation  
  [ Resource \ Car,  
    Usage \ Hire,  
    Feature \ CarFeature ]  
  && Hire:: car.features  
  subsetOf features  
)  
Levenshulme Whistle =  
( ResourceAllocation  
  [ Resource \ Slot,  
    Usage \ Story,  
    Feature \ StoryChrtrstic]  
  && Story:: features subsetOf  
  slot.features  
)
```

pictorial version



- Most useful if there is a tool for it
- You supply this info; tool unfolds to complete defns

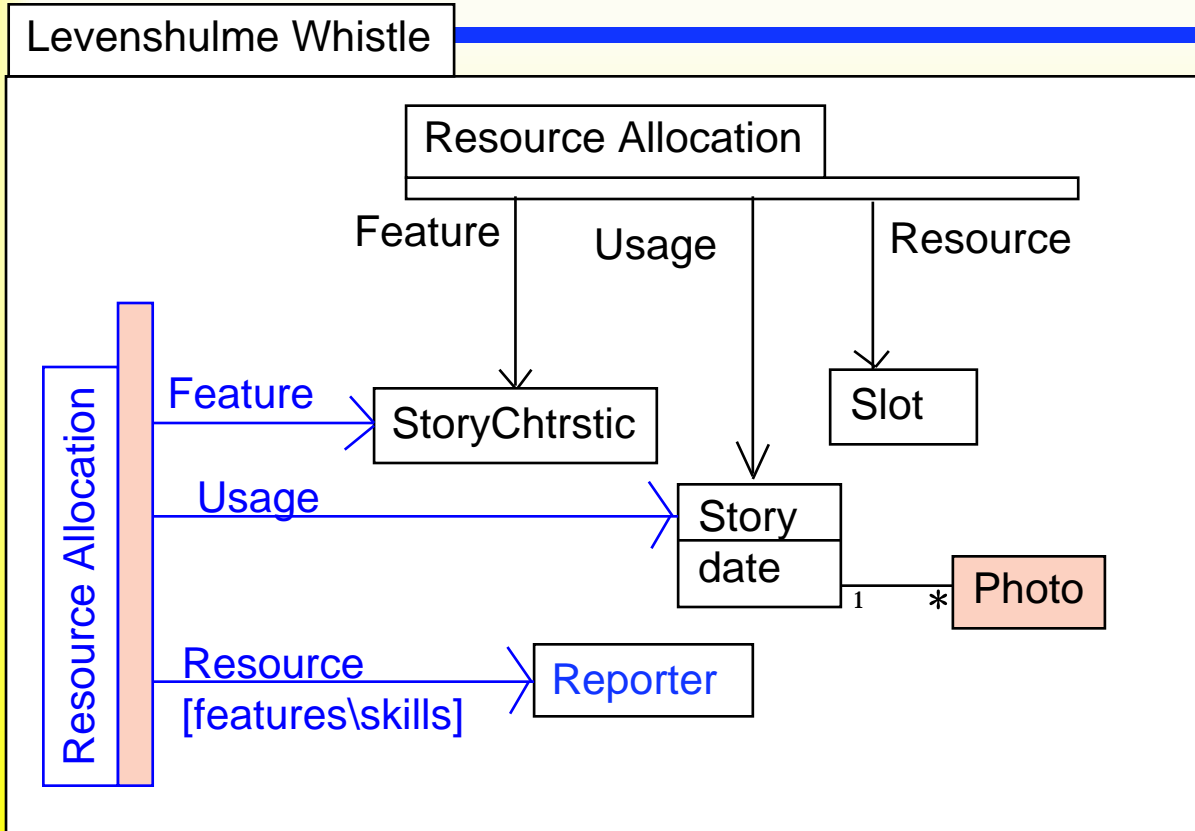
Nearest tools to date

- Some tools (such as Rose) have a facility for applying a script to a model
 - This is not an import
 - subsequent alterations to generic part have no opportunity to propagate
 - There is no way of making a generic model
 - you have to write a script
- They can't compose the operation frameworks that we'll look at shortly

More

- Ensure the *Whistle's* staff aren't overworked: each Reporter should only handle one story in each issue.
- Each Reporter is able to handle Stories with certain Characteristics. We need to ensure the Reporter assigned to every Story has the skills for the job.
- What's the easy way to add these to our model?

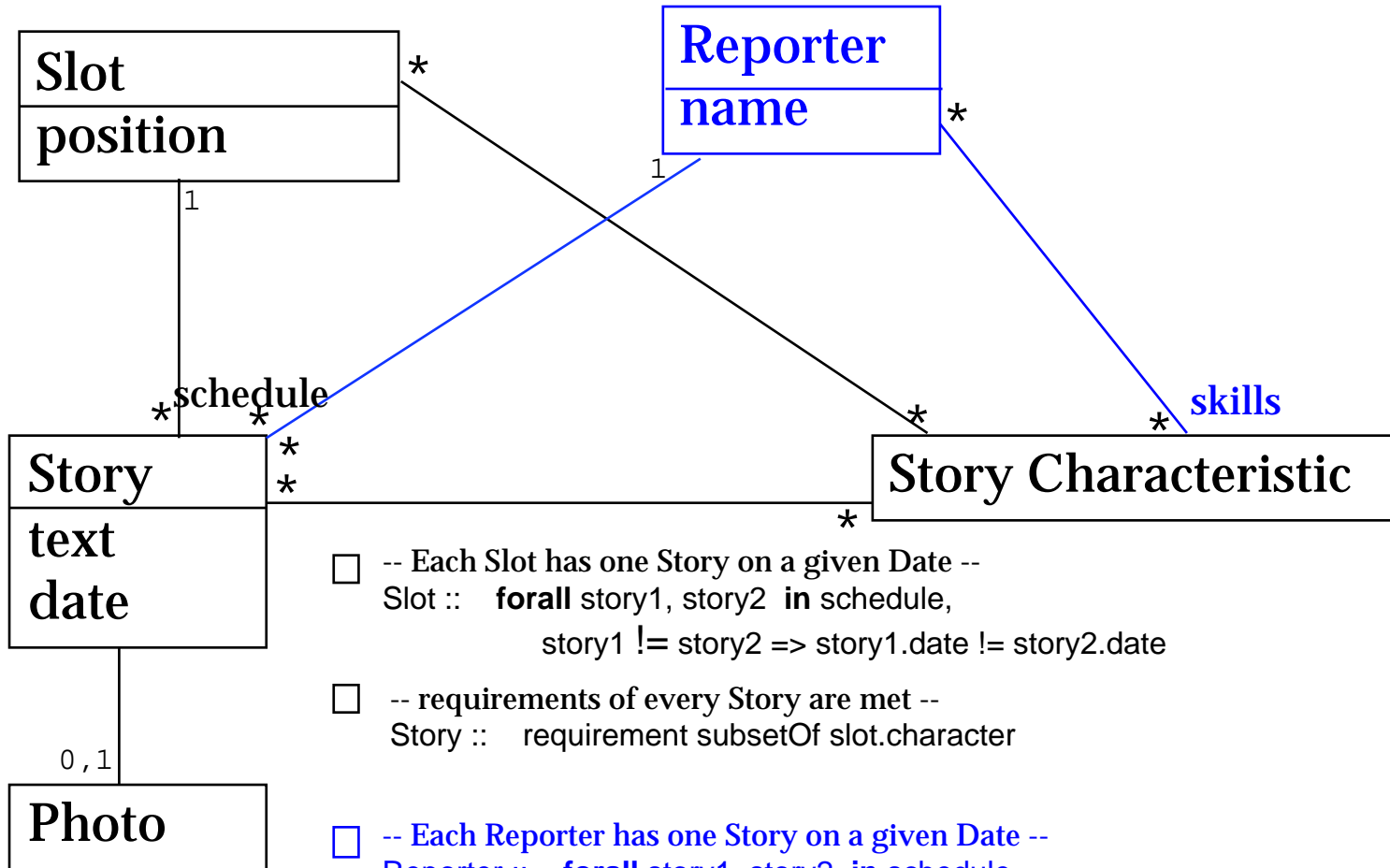
Multiple macro application



- Is this useful?
- Does it make anything easier?
- Are there tools that do this?

Result ->

Levenshulme Whistle



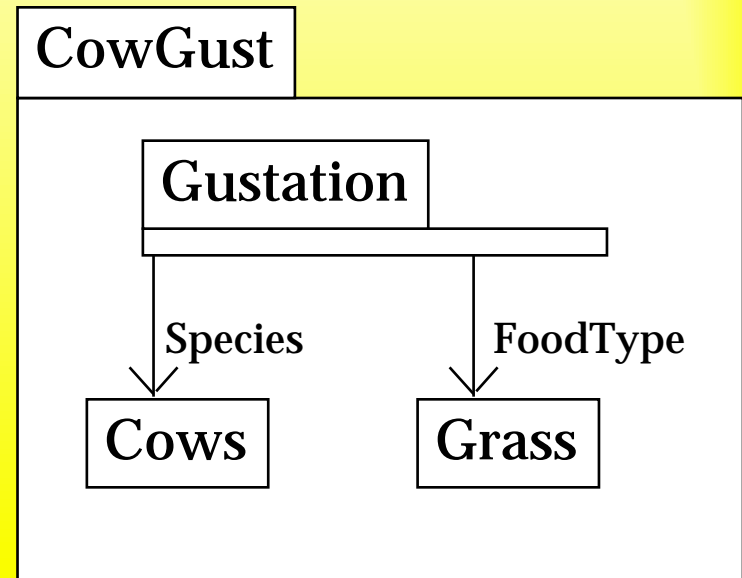
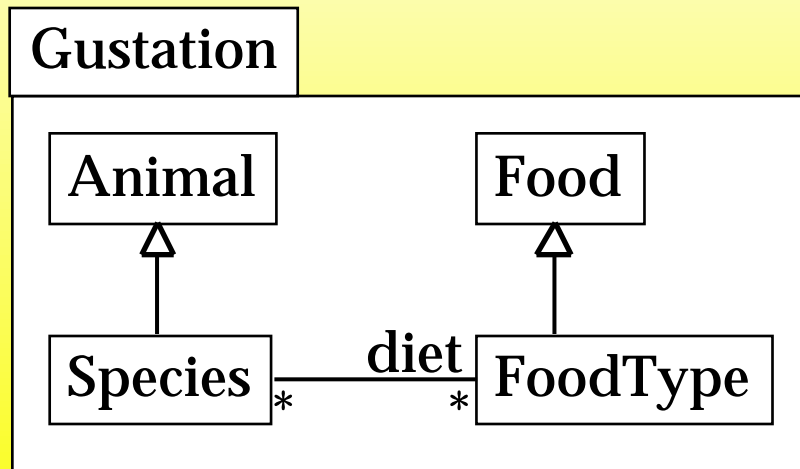
- Each Slot has one Story on a given Date --
Slot :: forall story1, story2 in schedule,
story1 != story2 => story1.date != story2.date
- requirements of every Story are met --
Story :: requirement subsetOf slot.character
- Each Reporter has one Story on a given Date --
Reporter :: forall story1, story2 in schedule,
story1 != story2 => story1.date != story2.date
- requirements of every Story are met --
Story :: requirement **subsetOf** reporter.skills

Modelling conundrum

- Animals eat Food
- Cows is Animals
- Burgers is Food
- => Cows eat Burgers
- ?
- Sort out the mistake.
- Model this correctly

Species — Food relation

- Every Species of Animal has a Food it eats; Cows is a Species; Grass is a Food



Operations

- Pre/postconditions
 - Postconditions specify what's required, but can leave some aspects open to implementor or further specification (in specialisations)

Circle
float radius
Point centre
alignWith(Circle c2)
post centre==c2.centre
...

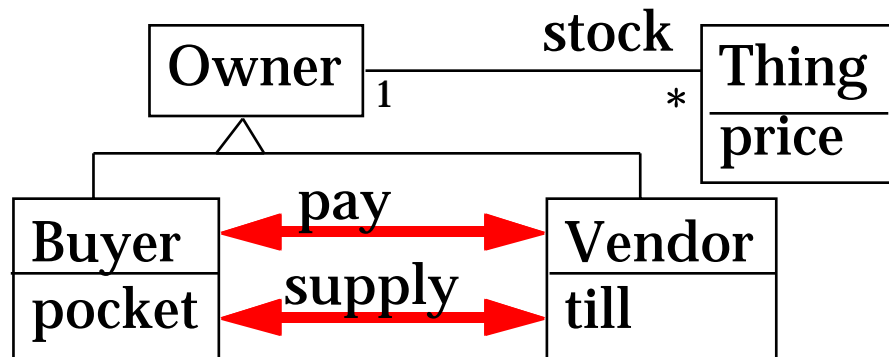
Collaborations & responsibilities

- Key OO design decision:
who does what
 - distribution of responsibilities between collaborators
- A good design notation should allow documentation of the state of the design both before and after each such decision is made
- “Collaboration”: interactive relationship

Collaborations

Sales

Before taking decisions about who initiates actions:



action pay (buyer, vendor, amount)

pre buyer.pocket > amount

post buyer.pocket -= amount

&& vendor.till += amount

action supply (vendor, buyer, thing)

pre vendor.stock **includes** thing

post vendor.stock -= thing

&& buyer.stock += thing

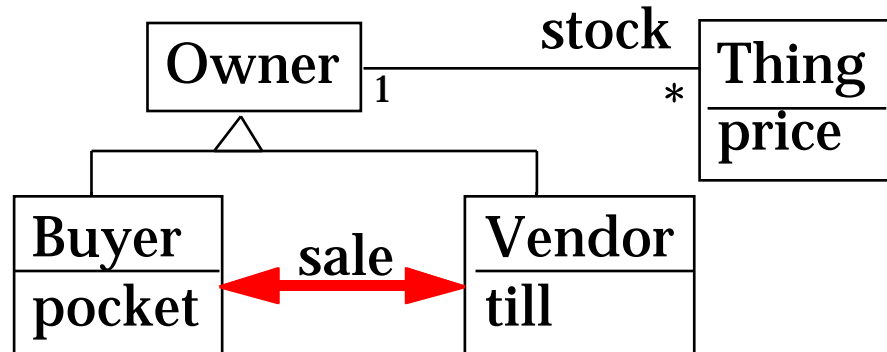
Refining and abstracting collaborations

- Pay and supply are abstractions.
 - What could replace them in a more refined design, one step closer to the code?
 - What could they be refinements of? What would its pre/post spec look like?
 - How are they related to it?
- How would this model apply to:
 - A person and a drinks vending machine?
 - A retailer and a distributor? — AND:
 - A retailer and a customer?

Abstract action

Sales Spec

pay(buyer,
vendor, thing)
&&
supply (vendor,
buyer, thing)
=>
sale (thing,
buyer, vendor)

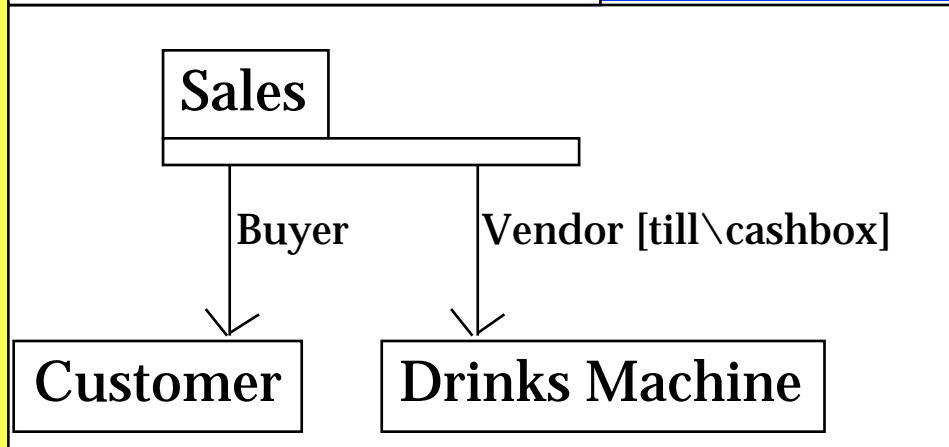


action sale (thing, buyer, vendor)
pre buyer.pocket > thing.price
post buyer.pocket -= thing.price
&& vendor.till += thing.price
&& buyer.stock += thing
&& vendor.stock -= thing

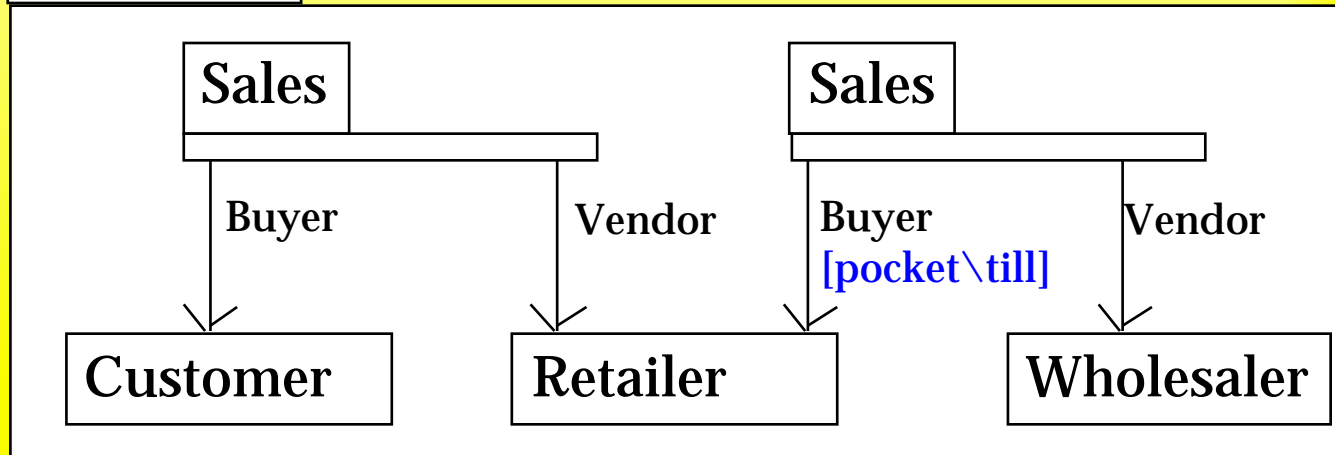
Sales

Generic Model Instantiation

DrinksVendingModel



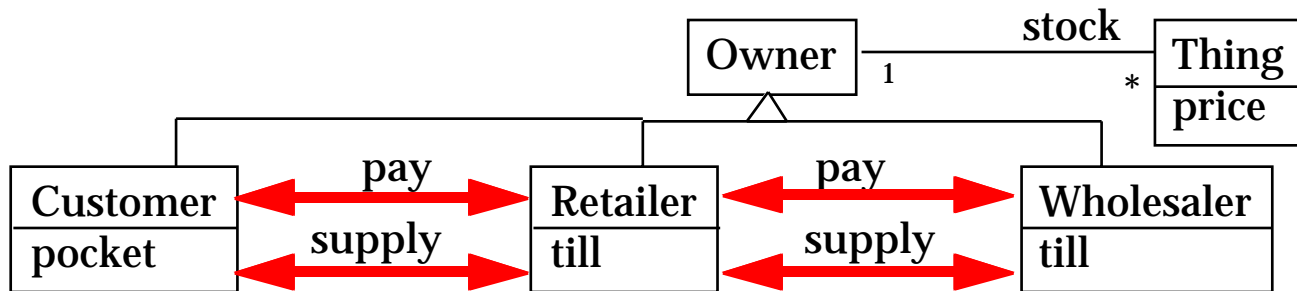
SaleChain



Unfolded composite

- Tool could show this on request

SalesChain



action pay (customer, retailer, amount)
pre customer.pocket > amount
post customer.pocket -= amount
&& retailer.till += amount

action supply (retailer, customer, thing)
pre retailer.stock **includes** thing
post retailer.stock -= thing
&& customer.stock += thing

action pay (retailer, wholesaler, amount)
pre retailer.till > amount
post retailer.till -= amount
&& wholesaler.till += amount

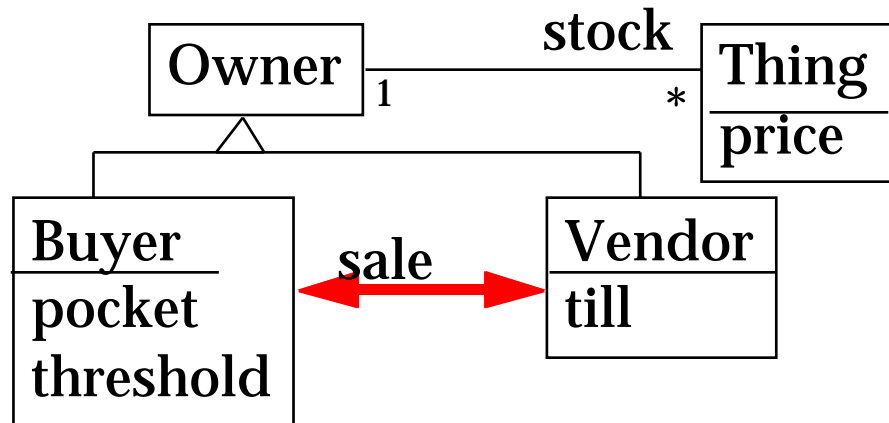
action supply (wholesaler, retailer, thing)
pre wholesaler.stock **includes** thing
post wholesaler.stock -= thing
&& retailer.stock += thing

Decoupling generic models

- What prompts buying in:
 - a Retailer or Distributor?
 - a Restaurant or Factory?
- What's common between these?
- Separation of concerns:
in what generic model should we record
what prompts buying?
 - Model describing buying new stock?
 - Model describing consumption of stock?

'Abstract hook' actions

Sales

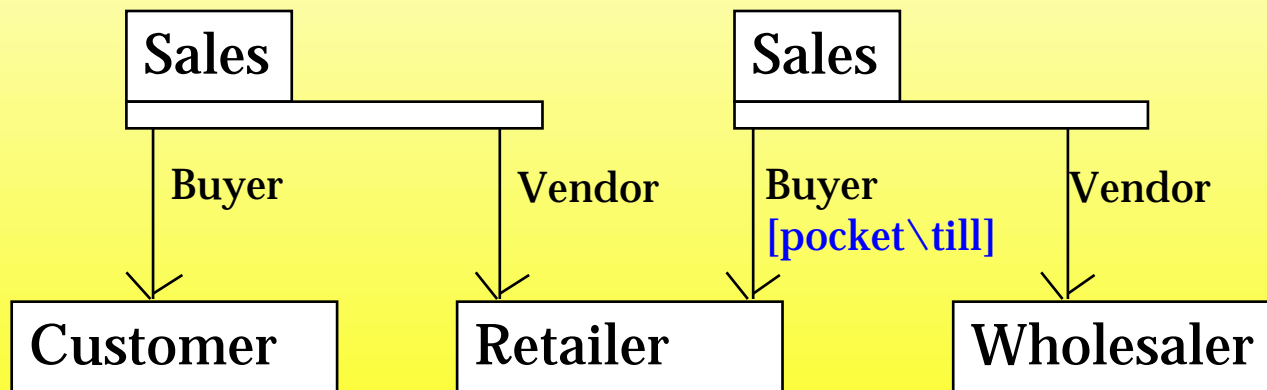


Any other action that causes the buyer's stock to drop below a threshold results in a sale

post buyer.stock < buyer.threshold ==>
action sale (buyer, vendor, amount)
pre ...// can only happen if this true
post ...

Model framework composition

- What causes the Retailer to buy new stock from the Wholesaler?



- Depletion of stock

Using generic models

- Model frameworks give *specifications* for classes that play multiple roles
- With clever architecture, you can either
 - build corresponding implementations that plug together in many combinations;
 - or use the generic models to do the design, and complete it by coding the classes from the resulting specs.

Roles composition & subtyping

Conclusions

- As reusable assets, generic model frameworks are as good as classes
- Roles are not substitutable subtypes
- Model frameworks are the formalisable part of design patterns