



Using Inheritance

Generalisation/Specialisation

🌐 Objectives

- Introduce generalisation/specialisation modelling techniques and notation

🌐 Contents

- Notation
- Abstract classes
- Multi-level hierarchies
- Modelling Problems
- Heuristics

🌐 Summary

🌐 Practical

Inheritance

- A feature of generalisation/specialisation relationship between classes
- Inheritance provides a simplification of captured knowledge
 - Can reason about the superclass, ignoring the existence of, and differences between, subclasses
- Inheritance provides a foundation on which to build subclasses
 - Analysis of subclass has a starting point
- Inheritance enables seamless addition of new classes
 - All existing classes continue to work exactly as before

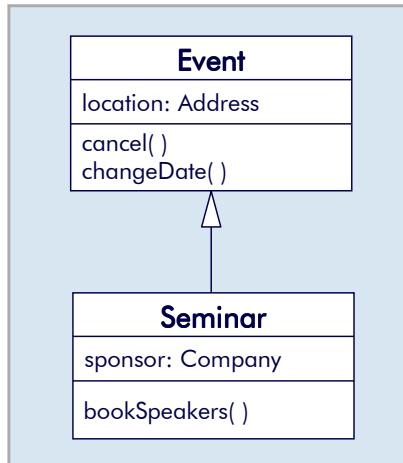
While aggregation is ultimately a relationship between objects, e.g. a particular 'Event' object *has-a* 'Location' object, *inheritance* occurs between two classes if one can be regarded as a specialisation of the other. E.g. a 'Seminar' class can be defined *as-a-type-of* 'Event' class. Because Seminar objects are types of Events, they inherit *all* of the features of Events – attributes, operations, relationships and everything else too. The specialisation is known as a *subclass* of the more generalised *superclass*.

Typically, if two or more classes have features in common and can be thought of as being 'part of the same family', then there may be advantages to factoring the commonality into a more generalised superclass. To do this is to create an opportunity for re-use in the future, as we can then create new subclasses as circumstances dictate.

It would be possible to build a model without using generalisation or specialisation, and it should only be used with justification. It is true, however, that the use of this relationship and the resulting inheritance can help our understanding of the model. As humans, we reason quite naturally at different levels of abstraction, generalising when convenient and being particular when it is necessary. Charles Darwin managed to make a career of it!

Generalisation/specialisation Notation

- A line with a closed, unfilled arrowhead at the superclass end



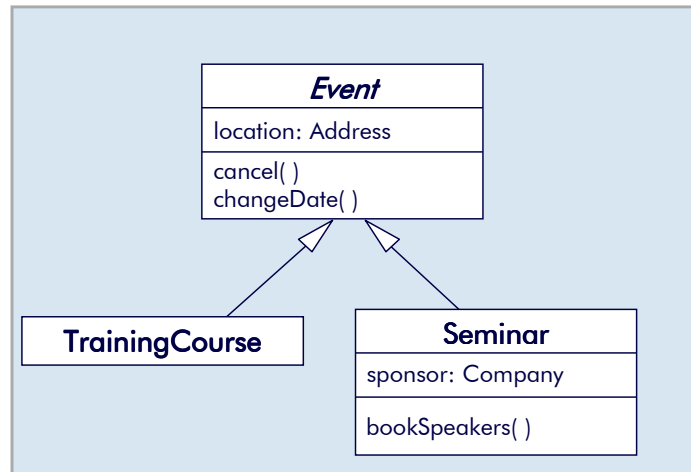
- A seminar is a type of Event
- A seminar has a location and a sponsor, and has operations 'cancel', 'changeDate' and 'bookSpeakers'

The definition of the Seminar class *includes* the definition of the Event class by inheritance. There is no need to duplicate attributes, operations or associations in the Seminar subclass. The subclass can be thought of as a Event with additions.

Abstract Classes

- An abstract class is one that will never have direct instances
 - Rendered using italics OR an {abstract} constraint under class name

In this case we have separated Training Course into a separate class, even though there are no differences from Event. This would allow us to change TrainingCourse without affecting other subclasses of Event e.g. Seminar



Have you ever seen a Mammal? That is, something that has only mammalian features? A Cat won't do, because it is more than just a mammal – it is a mammal with catty bits added, which distinguish it from a dog, which is also a mammal, albeit one with doggy bits added. No, mammals *per se* don't actually exist in the real world, but that doesn't stop 'Mammal' being a useful classification when we want to talk about a certain type of warm-blooded animal at a general level. Mammal, in this case, is an *Abstract* class, as opposed to a *Concrete* class which can have direct instances.

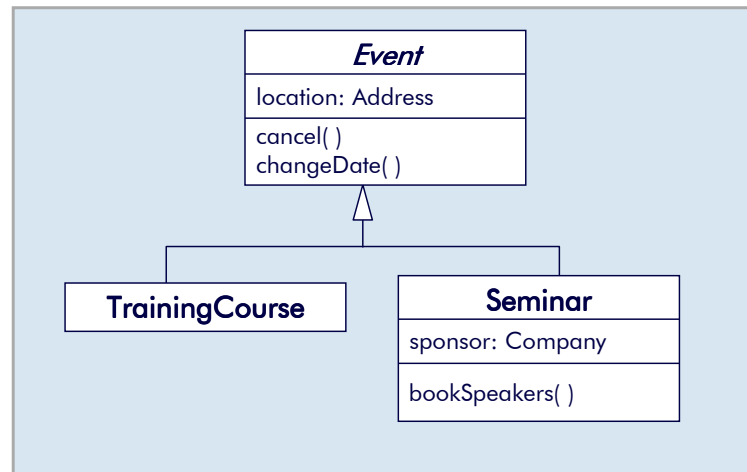
In the picture above, 'Event' is abstract, because the only types of event under consideration are training courses or seminars. There is no such *thing* as something that is *only* an event.

Abstract classes are used to contain commonality, which may be common concepts, common attributes, common operations or common relationships. This is, of course, also true for concrete (or non-abstract) classes. Thus, common superclasses are useful, whether they are abstract or not.

Some methodologists believe that only the leaf classes (those at the very bottom of the hierarchy) should be concrete, and all others should be abstract. The benefit of this is that an explicit subclass can be changed without affecting all other subclasses. For example, if Event above was concrete and TrainingCourse didn't exist, changes to TrainingCourse would have to be done through Event, thus affecting Seminar.

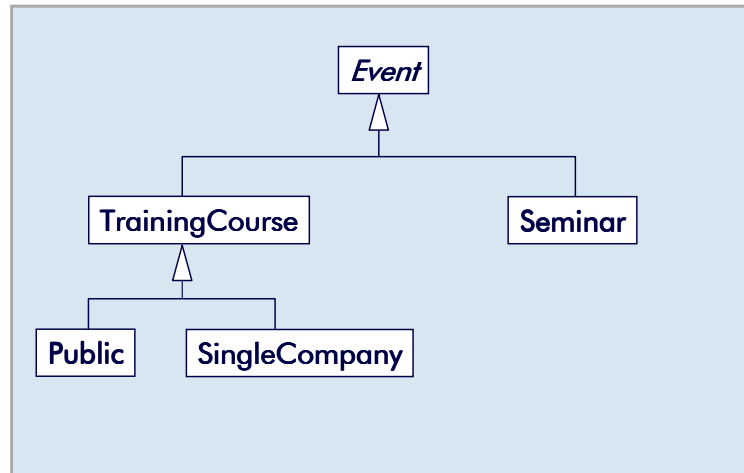
Alternative Representation

🎯 Means exactly the same!



All subclasses can be combined into a single arrow, or left separate. Both representations mean that the subclasses are mutually exclusive alternatives.

Multi-Level Hierarchy



Any number of levels of generalisation is possible, although in practice, we find that about three is more than adequate for most purposes, and is a threshold beyond which understanding is impaired. That is, if you have a deeper hierarchy, in order to understand the lowest class, you need to understand each of the classes above it, and it becomes difficult to conceptualise them all at the same time.

There is no limit to the width of the hierarchies, as each class is considered independently from its peers.

"is-a-kind-of" and Substitutability

- Generalisation/specialisation should only be used when the "is-a-kind-of" rule applies

A subclass "is-a-kind-of" superclass

A public training course "is-a-kind-of" training course

A training course "is-a-kind-of" event

Therefore a public training course "is-a-kind-of" event

- Generalisation/specialisation should only be used when the substitution rule applies

Wherever a superclass object can be used, a subclass object could be used, without any difference to the user

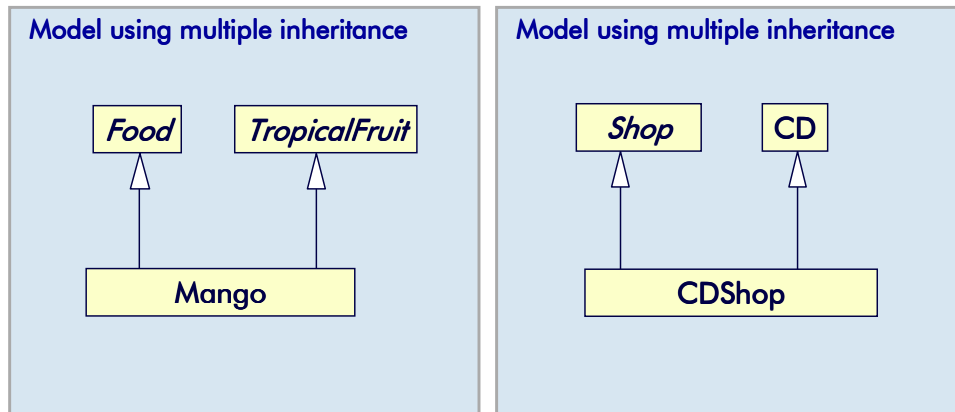
A delegate can attend an event. A delegate can attend a public training course

Unfortunately, programming languages cannot enforce these semantic rules, and it is easy to find examples in commercial software that break these rules. Sticking rigidly to these rules will make it easier for your classes to be understood.

An additional rule is that a subclass should never omit any of its superclasses' operations or attributes. If even one operation or attribute does not make sense on a subclass, the inheritance hierarchy is wrong. Not providing a method for an operation, or overriding a superclass method to do nothing indicates that the subclass behaves differently to the superclass.

Modelling Problem 1

What's wrong with these models ?

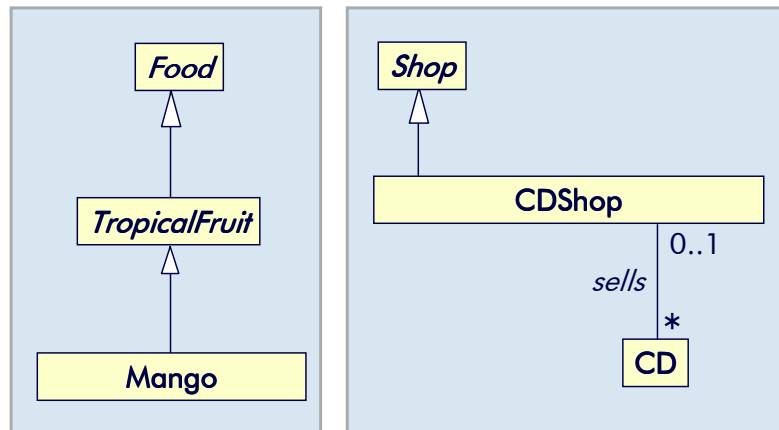


Multiple inheritance (MI) occurs when a class has two or more superclasses. While our brains are quite capable of perceiving examples of MI in the real world, models that include it can be quite complex and often lead to lattice-type structures rather than the simpler trees of single inheritance. It is also easy to misuse, as the examples above show.

Modelling Solution 1

Use multiple inheritance with caution

Question whether you have made a mistake



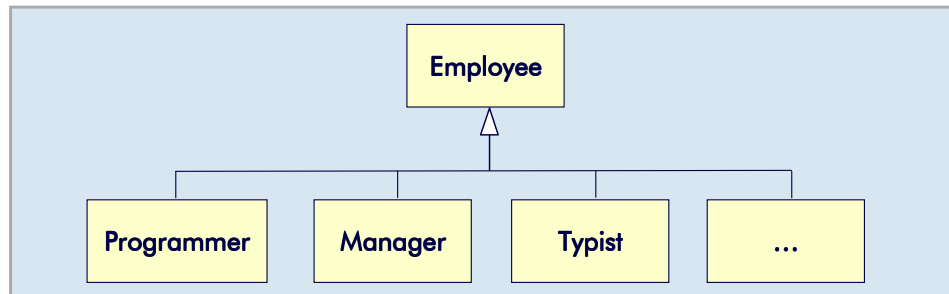
For this problem all tropical fruit are in fact kinds of food. This first model can be straightened into a single generalisation/specialisation hierarchy.

A CDSShop is a kind of shop but it is not a kind of CD. There may be some aspects in common between a CD and a CD shop. Both can provide music and information. Both can be bought and sold.

They are also fundamentally different in nature. A shop is a place of employment and may sell items. The CDSShop sells the CD. It has many CDs. The relationship between the CD and the CDSShop certainly not generalisation. It is that of association.

Modelling Problem 2

What's wrong with this model?



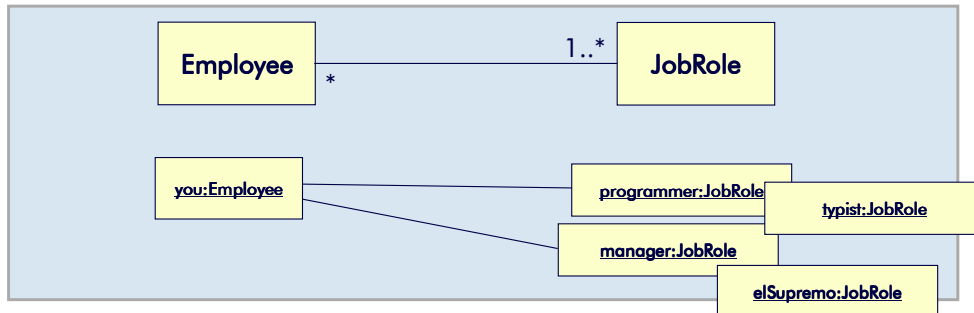
Consider:

- What happens if a typist gets promoted to manager?
- What happens if we want programmer/managers?
- What are the problems with adding new job descriptions?

Modelling Solution 2

- All changes require adjustment to model

- Better to use dynamic relationships than static



- New job roles can be instantiated dynamically

- Links can be made and broken dynamically

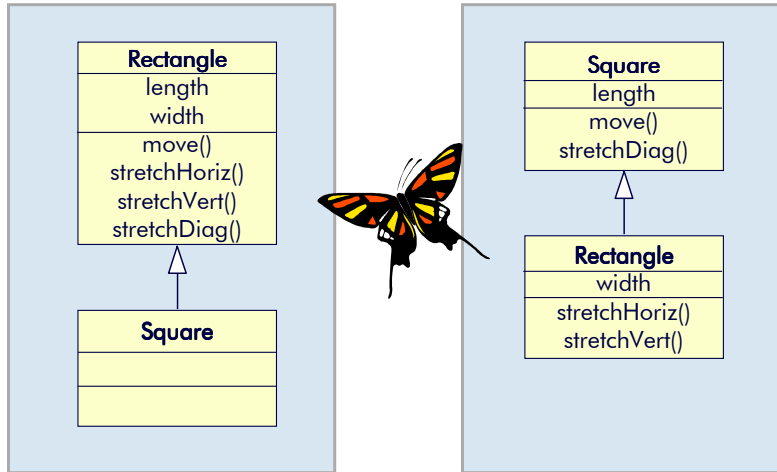
The big problem with the model on the previous page is that it looks OK! Many people would readily agree that 'A Manager is_a_type_of Employee'.

But the real problem lies in its inflexibility. Generalisation/specialisation hierarchies are static – that is, they remain the same throughout the life of the system and they cannot be changed dynamically. This means that any change is a compile-time issue.

It can be more effective when modelling volatile or dynamically-changing structures to use associations and/or aggregations which can be instantiated or destroyed as links at run-time.

Modelling Problem 3

🎯 Compare, contrast, discuss: which model is wrong?

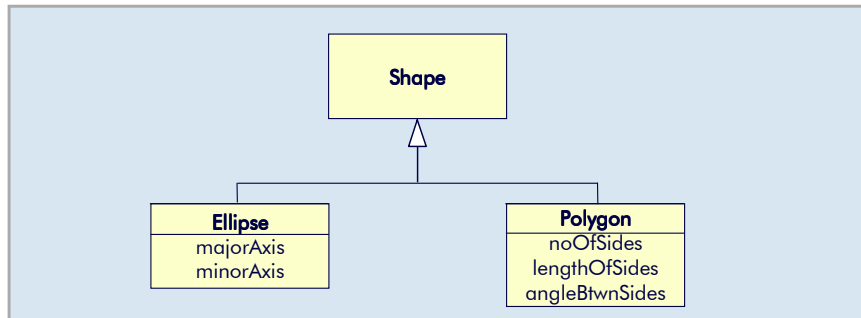


Modelling Solution 3

- They're both sub-optimal!
 - An example of the 'Butterfly Problem'



- Squares and rectangles are both states of a quadrilateral
 - and a quadrilateral is just a state of a polygon!



This is an example of the 'Butterfly Problem'. Consider: an egg, a caterpillar, a pupa and a butterfly. How many objects? The answer: just one, but in one of four different states in its lifecycle! This demonstrates how easy it is to be fooled by the simple 'noun = class' method of identifying classes. State names (or role names) can also be nouns.

Generalisation/specialisation Heuristics 1

- Generalisation/specialisation should only be used when the "is a kind of" rule or substitutability rule apply
- Identify classes that have similar lists of operations and/or associations
- Attributes are not the main instigator
- Remember a subclass inherits *everything* from its superclasses
 - Do you really want everything?

Generalisation/specialisation Heuristics 2

- Do not create deeply nested hierarchies
- Do not use generalisation/specialisation for volatile hierarchies
 - Remember – Inheritance is for life, not just for Christmas
 - Consider using more dynamic relationships
- Use generalisation/specialisation with care
 - Use multiple care with multiple inheritance
- Use generalisation/specialisation strategically, not tactically
 - Keep in mind future re-use

Class Modelling Exercise

Objective

- To consolidate class modelling techniques

Task:

- Create separate class diagrams that you think best represent the situations below:
- Problem 1: Trains are either passenger trains or freight trains. The same kind of locomotives are used in each, but the cars in a passenger train are all passenger cars, and in a freight train, all freight cars.
- Problem 2: A widget is composed of three parts. Parts are either metal or plastic. The parts suppliers are set up to provide either exclusively metal or plastic parts; not both.