

Refining Object Relationships

Association and Aggregation

🎯 Objectives

- Familiarisation with modelling associations and aggregations

🎯 Contents

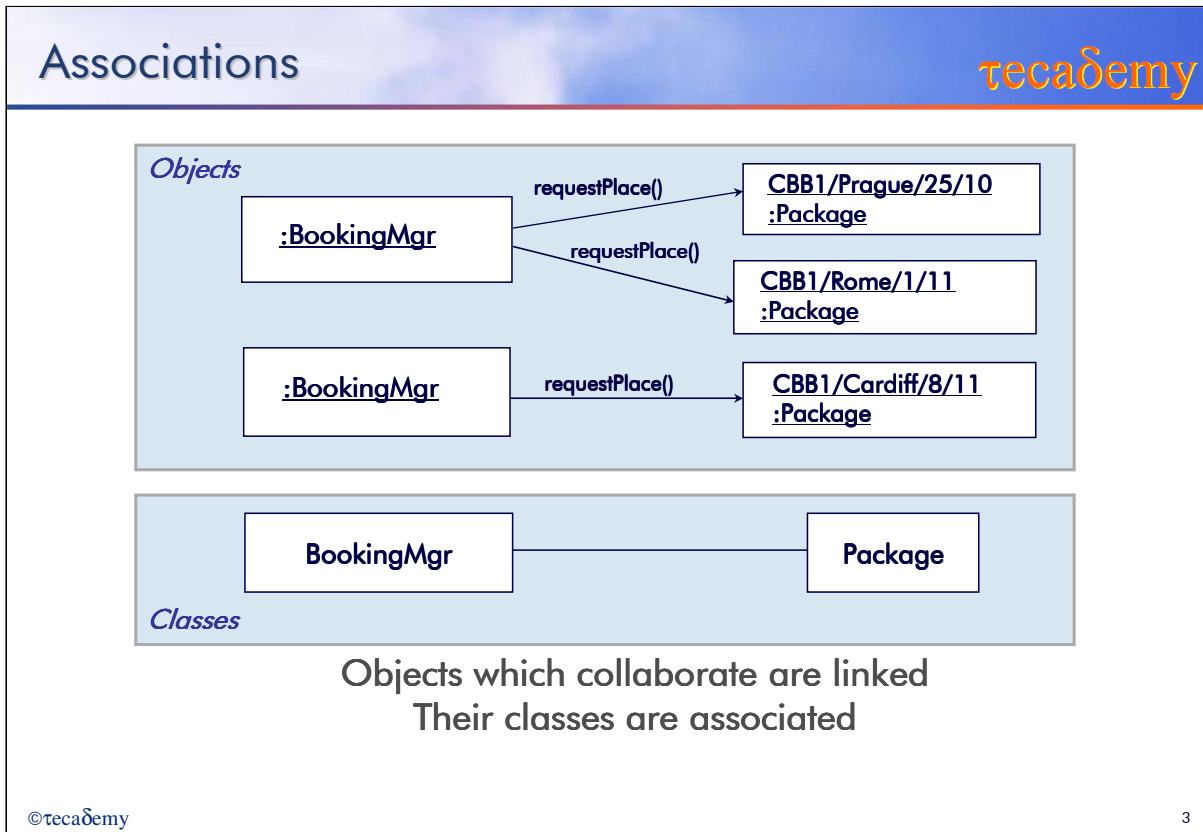
- **Associations**
 - Roles, Multiplicity, Constraints
 - Association Class
- **Aggregation**

🎯 Summary

🎯 Practicals

Before looking at the details of class diagrams, much preparation work is needed. Some use cases should have been done to identify the contexts within which the class diagram exists, as well as providing candidate classes. Scenarios and sequence diagrams will have built upon these candidate classes, adding attribute, operation and association detail.

In this chapter we will further refine the class diagram, looking at the topics shown above.



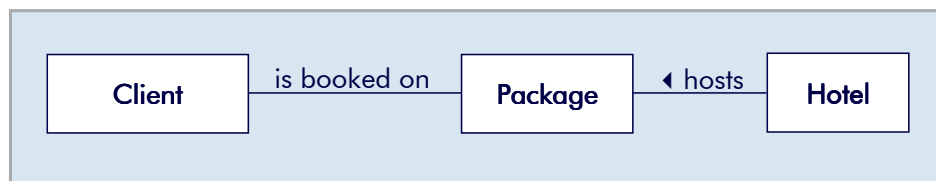
An association is a semantic relationship between two or more classes, in turn this will mean that the objects of that type can be linked. Most relationships connect two classes. Ternary relationships are also allowed.

Temporary links are established when an object is created (between the creator and created object) and when an object is passed as a parameter to another object.

Just as an object is an instance of a class, a link is an instance of an association.

Associations

- ▶ Association name describes why links exist
 - ▶ High-level abstract description, covering all operations
 - ▶ Written in one direction, readable in both (with adjustments)
- ▶ If reading direction is not obvious, add an arrowhead



Associations should be named. The activity of finding an appropriate name should help in understanding the reason for the association, as well as recording this for posterity. The underlying reason why particular objects are linked to each other should be sought, rather than simply documenting the name(s) of the operation(s) that are invoked across the link. Thus, ask why the Package knows about its Clients, (because it maintains or owns them), rather than simply recording that it adds and deletes them.

Roles and Reflexive Associations

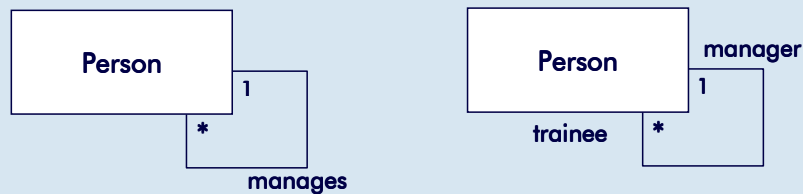
▶ A role is the face that class shows to another class

➤ Read in appropriate direction

Example Roles



Example Reflexive Association



Role names are read in both directions.

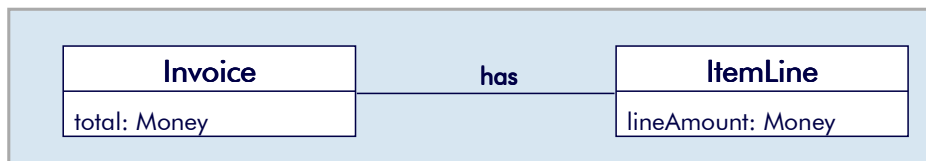
A Person plays the role of 'employee' in his/her relationship to a Company.

A Company plays the role of an employer to a Person.

Reflexive associations exist where two or more objects of the same class can be linked. Association names on reflexive associations can be ambiguous with regard to reading direction. In the slide above, it is not clear whether one employee manages many employees, or many employees manage one unfortunate individual. Role names remove this ambiguity.

Multiplicity

- ▶ Captures the business rules of the relationship
 - If I have one of these, how many of those must I have?
- ▶ Look at each relationship and decide how many objects are involved
 - From each end, decide how many objects at the other end



How many ItemLines does an Invoice have?
How many Invoices is an ItemLine connected to?

There are two multiplicity decisions to be made for a normal binary relationship as shown above.

Assuming the classes are A and B, the two decisions are:

One A object is associated with how many B objects?

One B object is associated with how many B objects?

These are dependent on the business rules. Some applicable ones may be:

An ItemLine must be on exactly one Invoice (An ItemLine with no Invoice is illegal)

An Invoice must always have at least one ItemLine (is there a maximum number?)

Multiplicity Notation

tecademy

```

classDiagram
    class Invoice {
        total: Money
    }
    class ItemLine {
        lineAmount: Money
    }
    Invoice "1" -- "1..*" ItemLine : has
            
```

An *ItemLine* is for one *Invoice*

An *Invoice* has one or more *ItemLines*

| | | | | | | | | | | | | | |
|---------------------------|--|--------|------------------|--------|-----------------------------------|-------|----------------------------|---|----------------------------|---|-----------------------|------------|-------------------|
| General expression | lower .. upper , lower .. upper , | | | | | | | | | | | | |
| Examples | <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px;">1 .. 5</td> <td style="padding: 2px 10px;">1 to 5 inclusive</td> </tr> <tr> <td style="padding: 2px 10px;">0 .. n</td> <td style="padding: 2px 10px;">Optional (n=any positive integer)</td> </tr> <tr> <td style="padding: 2px 10px;">0 , 1</td> <td style="padding: 2px 10px;">Zero or 1 (same as 0 .. 1)</td> </tr> <tr> <td style="padding: 2px 10px;">6</td> <td style="padding: 2px 10px;">Exactly 6 (same as 6 .. 6)</td> </tr> <tr> <td style="padding: 2px 10px;">*</td> <td style="padding: 2px 10px;">Many (same as 0 .. *)</td> </tr> <tr> <td style="padding: 2px 10px;">0 , 4 .. *</td> <td style="padding: 2px 10px;">Zero or 4 or more</td> </tr> </table> | 1 .. 5 | 1 to 5 inclusive | 0 .. n | Optional (n=any positive integer) | 0 , 1 | Zero or 1 (same as 0 .. 1) | 6 | Exactly 6 (same as 6 .. 6) | * | Many (same as 0 .. *) | 0 , 4 .. * | Zero or 4 or more |
| 1 .. 5 | 1 to 5 inclusive | | | | | | | | | | | | |
| 0 .. n | Optional (n=any positive integer) | | | | | | | | | | | | |
| 0 , 1 | Zero or 1 (same as 0 .. 1) | | | | | | | | | | | | |
| 6 | Exactly 6 (same as 6 .. 6) | | | | | | | | | | | | |
| * | Many (same as 0 .. *) | | | | | | | | | | | | |
| 0 , 4 .. * | Zero or 4 or more | | | | | | | | | | | | |

©tecademy
7

The multiplicity is read from a single object at one end, across the association to the multiplicity marker at the other end, which specifies how many of those objects there may be in association.

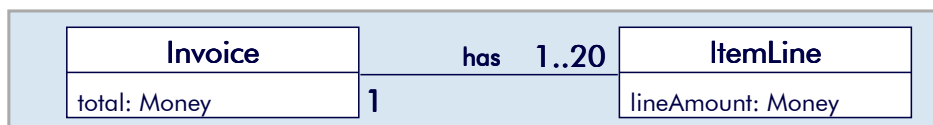
One Invoice has one or more ItemLine, it can never have no ItemLine.

One ItemLine is for exactly one Invoice.

The general expression for multiplicity is a set of comma-separated ranges. Each range may specify either a minimum and a maximum number, or a single number. An asterisk as the maximum means no upper limit, an asterisk on its own is read as 'many', and means from zero upwards. Note that a single number means just that number, not from zero to the number.

Multiplicity Implications

- ▶ The multiplicities define a *valid* system
 - (not necessarily a *useful* system)
- ▶ They are the Business Rules
- ▶ Must plan for attempts to break the constraints
 - Need to identify Use Cases/scenarios that make attempts



Must never have an ItemLine that is not on an Invoice

Must never have an Invoice that has no ItemLines

Must never have an Invoice that has more than 20 lines

Multiplicity constraints specify that the system should never be in a state where an invalid multiplicity could be observed. Obviously in the depths of programming, such a constraint could be momentarily broken, say for a one-to-one association. Where an object must be created, the multiplicity will be invalid until both objects exist.

If users or domain experts specify cases where business logic requires that the constraints are broken, the multiplicity (or the experts!) must be incorrect.

The class diagram does not, of course, specify the behaviour required when attempts are made to break the constraints. These must be handled in the dynamic models, Use Cases, scenarios, sequence diagrams, collaboration diagrams and state diagrams. E.g. the above model would imply the need for scenarios (and use case alternate flows) to handle situations where an attempt was made to delete the last ItemLine from an Invoice, or to add a twenty-first.

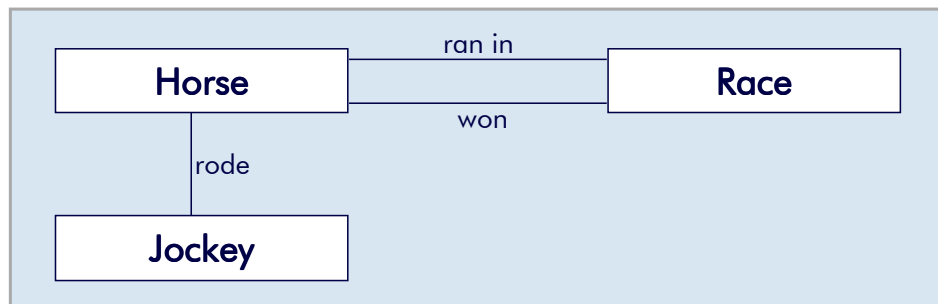
Determining Multiplicity

- ▶ Multiplicity captures the business rules
 - ▶ But consider impending/possible/reasonable changes
- ▶ Restrictive multiplicity will limit growth and extensibility
 - ▶ A system implemented with 'person works for one company', would have difficulty upgrading to many companies
- ▶ Too-expansive multiplicity will introduce complexity
 - ▶ Implementing many companies but only ever using one, has wasted development effort, and probably impacted user interaction
- ▶ Multiplicity must allow *only* legal situations
- ▶ Getting it right is best, but too many is the next best

Existing business rules are not always appropriate requirements for a new system. There may be scope for improvement.

Changing multiplicity after deployment is usually a tall order, and is more expensive than implementing it initially. For example, the cost of implementing a to-many multiplicity at initial design time may cost twice as much as implementing a to-one or optional multiplicity at that time, but changing *after* deployment from a to-one or optional multiplicity, to a to-many multiplicity may cost ten times as much.

Multiplicity In-Situ Exercise



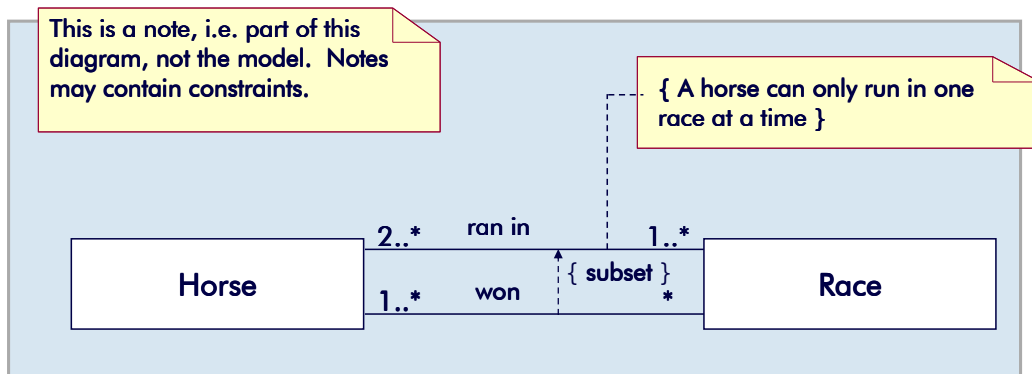
➤ **Multiplicity is dependent on the meaning of the association and classes**

- **Are we maintaining history?**
 - How would the system change if we only considered the current race?
- **Are the objects/associations complete?**
 - How would the multiplicity change if we allowed races that had not yet been run?

Without a definition of Race, almost any multiplicity could be argued, each using a different understanding of what a Race is. Alternatively, a Race could be 'The Cheltenham Gold Cup', allowing all years to be included, or future Races to be considered also. Each of these would have different effects on the multiplicity. Similarly, we must be careful with the naming and definition of the associations. Definitions of the classes and associations should appear in the data dictionary. Normally, the definitions and the multiplicity will be refined together.

Constraints and Notes

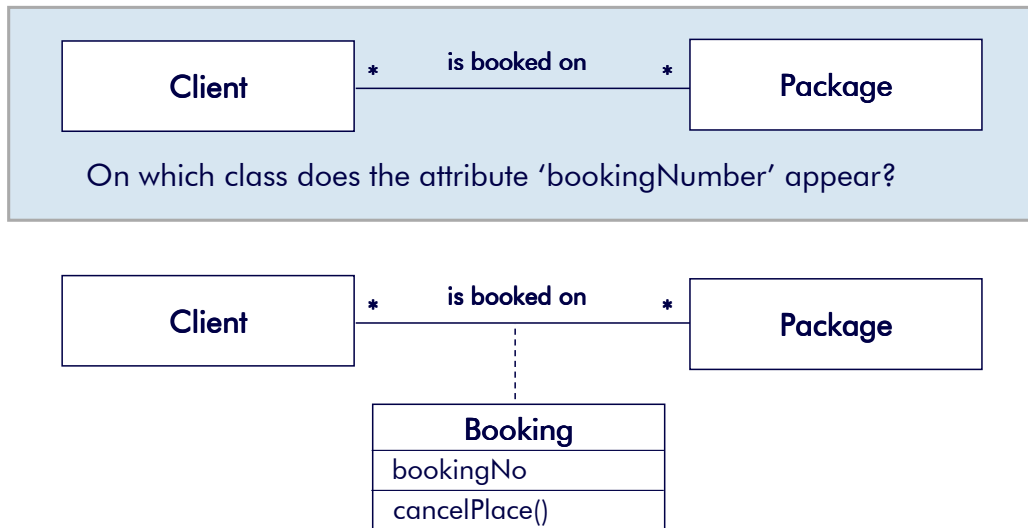
- ▶ Any modelling element(s) can be constrained
 - Constraints are modelling elements that may appear on diagrams
- ▶ Visible form is a dashed line
 - Constraint text must be within braces { }



Constraints should be used freely on all the diagrams of the UML, where the graphical format cannot easily be adapted to capture the constraint requirement. Constraints are first-class elements in the model, and should be captured in the data dictionary with references from all the constrained elements. Notes are not generally part of the model, and have no model semantics, except where a note contains a constraint.

Association Classes

- An association class defines properties of the association



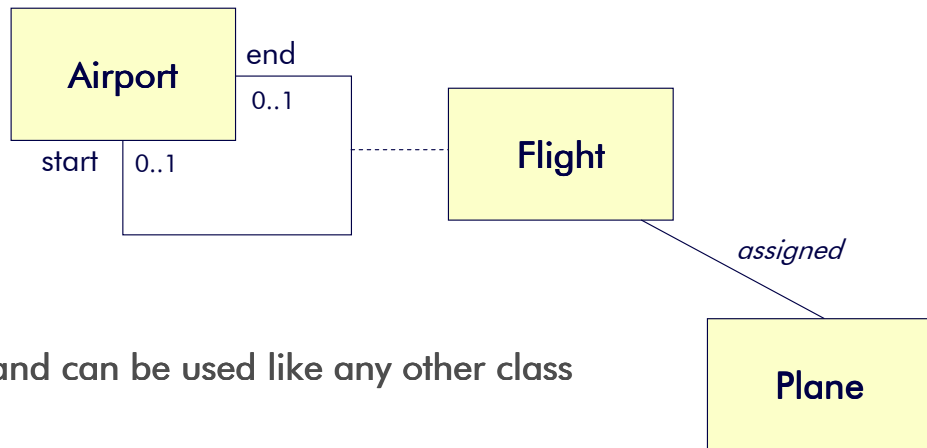
The attribute 'bookingNumber' could not be associated with the Client class only, as a Client may have many bookings, and the relationship between the booking and the Event would be lost. Similarly, the booking number could not be associated with the Event class only, as an Event will have many bookings, and the relationship between the booking and the Client would be lost. It would be possible for the booking number to be associated with both the Client class and the Event class, but this is essentially a duplication of the 'booking' relation, and may therefore confuse matters.

In this case, it is the concept of a 'Booking' that properly describes the relationship between a Client and an Event, and so 'bookingNumber' is actually a property of the association, not either of the participating classes. The association can therefore be modelled as a class, where an instance of the association class defines the link between instances of the two classes. These association classes can also have behaviours – a Booking can only be cancelled if a link exists between a Client and an Event.

To summarise, the Booking association class specifies that for each 'is booked on' link (instance of the association) between a Client and an Event, one instance of the Booking class will exist.

Association Classes

- Association Classes can exist regardless of multiplicity...



- ... and can be used like any other class

Association classes are not simply products of a many-to-many relationship between classes. They can exist regardless of the multiplicity of the association. They can exist also on reflexive relationships. The slide shows a reflexive optional relationship between two airports – it is an instance of the 'Flight' class that defines the relationship between a 'start' airport and an 'end' airport.

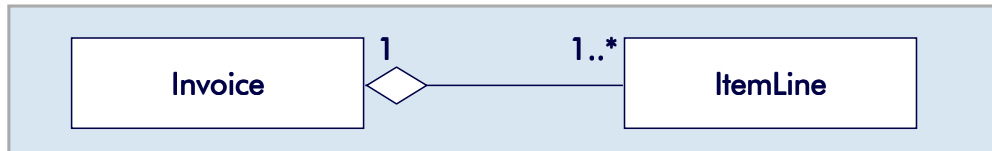
Association classes are classes in every respect, and can associate with, be aggregated into, or be sub/superclasses of other classes.

Aggregation

- A whole-part relationship
- Aspects of ownership
 - The whole owns the part(s)
 - The parts belong to one, and usually just one, whole
- Aspects of lifetime
 - The whole creates and/or deletes the part
 - If the whole dies, the parts die
 - The parts normally don't exist without a whole
- Aspects of accessibility
 - Associates of the parts access the parts via the whole
- Aspects of propagation
 - Some operations on the whole cause the same operations on the parts

There is much debate about the exact semantics of aggregation. In the absence of a definitive, unambiguous, cohesive definition of aggregation, we raise the issues involved. Each use of aggregation implies that 'some' of these issues are involved, but will need to be investigated and specified independently.

Aggregation Notation



An ItemLine is part of an Invoice

The Invoice owns its ItemLines

ItemLines don't exist without an owning Invoice

An Invoice is responsible for creating its ItemLines

If the Invoice is not needed, neither are its ItemLines

Access to the ItemLines should be through the Invoice

Archiving a Invoice should cause its ItemLines to be archived

The diamond is shown next to the class designated as the whole. The other object is the part. There is no need for an association name or role names; the aggregation defines that the whole 'is made up of', or 'contains', or 'has' the parts, and the parts 'are part of', 'are contained by' or 'belong to' the whole.

To take another example, say that of a Car to Chassis relationship, there is aggregation, because:

- There is a whole-part relationship; the Car is a whole, the chassis part of it.
- Chassis' don't normally exist independently (except in a manufacturing situation).
- If the Car is crushed in a junk yard, the Chassis would be crushed too.
- To check a chassis number, you would first identify the Car.
- If you drive the Car to Glasgow, the Chassis would go too.

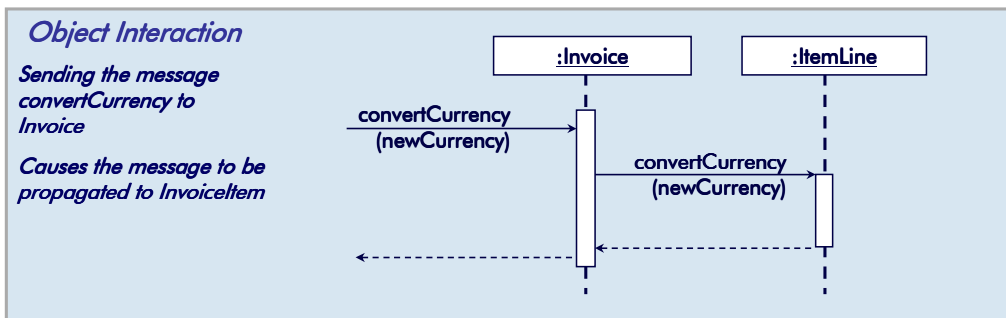
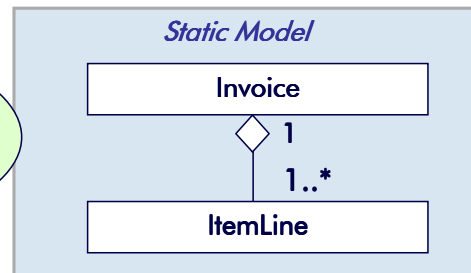
Whereas in the Car to Driver relationship, there is not aggregation, because:

- There is no whole-part relationship.
- Each exists without the other.
- Cars and Drivers are created and destroyed independently.
- Access to Cars is independent of access to drivers.

Although the Car may be driven to Glasgow, and the Driver would probably go with it, the Driver could also take the train to Edinburgh.

Propagation of Operations

"Propagation of operations to parts is often a good indicator of aggregation" Rumbaugh et al



If I delete an invoice, the item lines will be deleted with it, because they are part of the invoice. When an object is subjected to an operation that also gets 'passed down' or propagated to other objects, it can be taken as evidence to support the existence of an aggregation relationship between the objects.

Propagation of operations can take place across standard associations or aggregation relationships however it is commonly found in aggregation relationships. As another example, when sending the request to a document object to copy it may in turn send messages to the paragraph objects it owns and then in turn to character objects owned by the paragraph. This has the advantage of decoupling the original sender from the receiver. E.g. in the above example document has no direct knowledge of the character operations. When operations are propagated in this manner within a aggregation relationship it is common that one of the objects sending the message is acting as an agent. It simply receives messages and sends them on. Propagating operations can also add flexibility in assigning responsibilities to objects. As the responsibilities can be distributed among objects.

Aggregation: Why?

- ▶ **Makes a decision about abstraction explicit**
 - ▶ Identifies that the parts could be omitted (from a diagram) without losing the big picture: the parts are subservient to the whole
 - ▶ Identifies that the whole object should be responsible for managing the parts

- ▶ **May help with design decisions**
 - ▶ Parts should normally be in the same ownership unit
 - ▶ Parts should normally be in the same deployment unit
 - ▶ Parts only accessed through the whole (encapsulation)
 - ▶ May lead to physical object composition at implementation
 - ▶ Often identifies a reusable group of components

- ▶ **Because that's the way the Customer sees it!**

Although the aggregation does not mandate that any specific rule is true, it is useful, because it highlights the fact that each issue should be addressed. But the bottom line is – if the Customer says it's an aggregation, you are duty-bound to model it as such (you can always tinker with it during design!)

Aggregation: Why Not!

- ▶ **Aggregation signifies a stronger relationship than simple association**

- ▶ **But it does not require ALL of the criteria**
 - ▶ The whole may not own the part(s)
 - ▶ The parts may exist independently
 - ▶ The whole may not create or delete the part
 - ▶ The part may not die if the whole dies
 - ▶ Parts may be accessed directly, not via the whole
 - ▶ There may not be any operations propagated from the whole

If none of the criteria are met, it probably isn't aggregation

Aggregation really just specifies a closer than usual relationship between classes and objects. Each of the criteria or rules should be examined to determine if it is to be used.

If there's any argument about whether it's aggregation or association, default to association – it's the more flexible option.

Aggregation In-Situ Exercise

Objective

- To practice differentiating aggregation from association

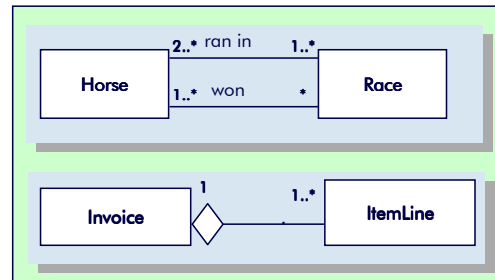
Task:

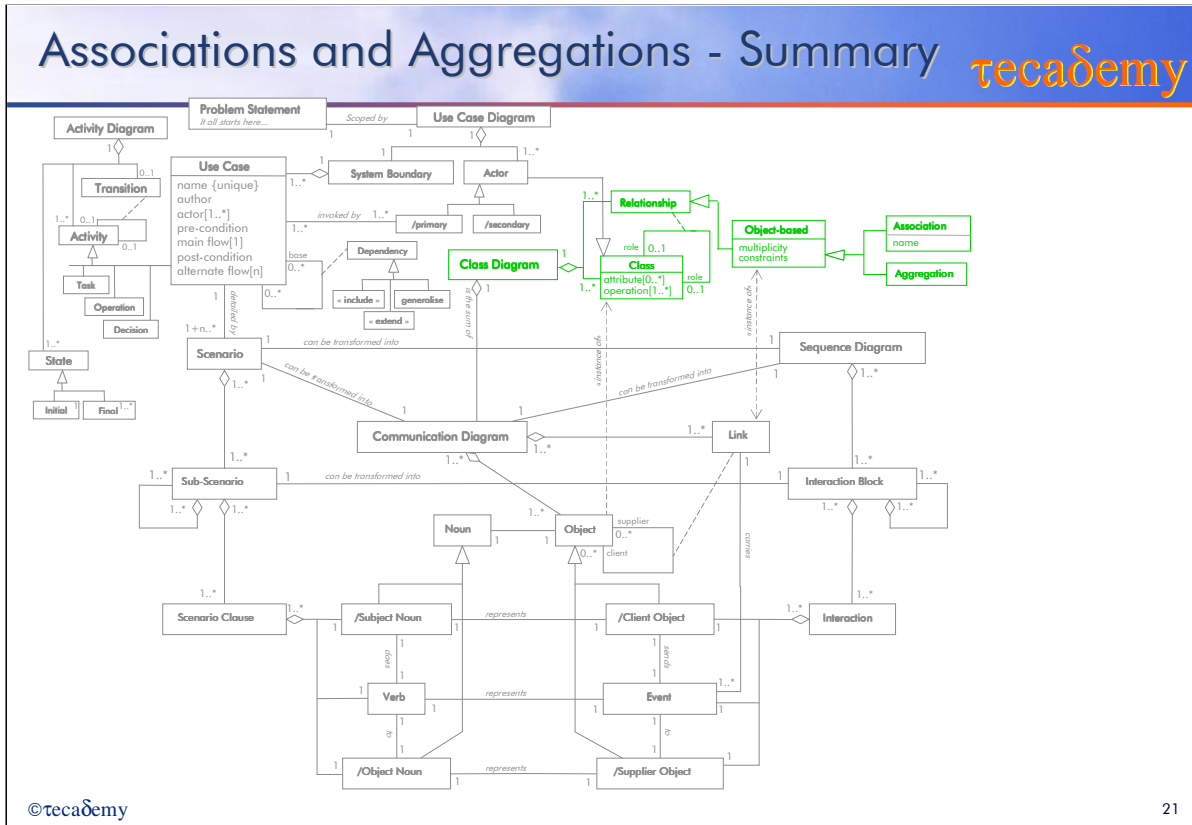
- As a group , decide for each of the following whether each aggregation or association would be most appropriate

- A track on a CD
- A mobile phone has a battery and sim
- An author of a document
- A car parked in a car park
- An organisation has many departments
- A plan has many tasks and milestones
- A person driving a car
- A guitar has strings

Chapter Summary

- ▶ Analyse each association to determine multiplicity in both directions
- ▶ Analyse each association to determine whether it is association or aggregation
- ▶ Do not be surprised if your class diagram requires revision
- ▶ Class diagrams often require multiple iterations to clarify names, add details, repair errors, reduce redundancy and simplify





The class diagram consists of classes (showing their attributes and operations) and the object-based relationships between them (association or aggregation, showing the multiplicity).

Case Study - Exercise 11

Objectives

- To practice refining the object-based relationships in a class diagram
- To consolidate understanding of multiplicity, role names, association and aggregation

Turn to and complete exercise 11 in the exercise booklet