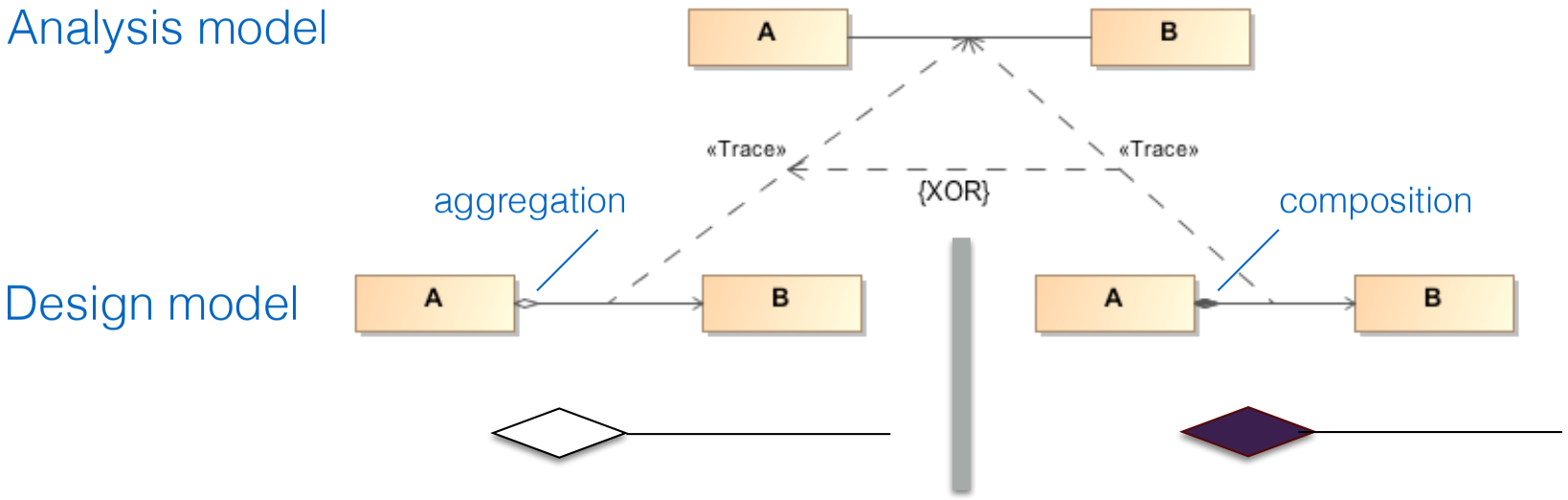
Design - refining analysis relationships

Design relationships

- Refining analysis associations to design associations involves several procedures:
  - Refining associations to aggregation or composition relationships where appropriate
  - Specifying navigability and multiplicity on *both* association ends
  - Implementing one-to-many, many-to-one and many-to-many associations
  - Implementing bidirectional associations
  - Implementing association classes

Aggregation & composition



- In analysis, we often use unrefined associations. In design, these can become aggregation or composition relationships where appropriate

- We must also add navigability, multiplicity and (optionally) role names or association names

Aggregation & composition

Aggregation

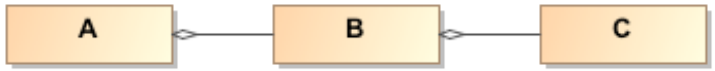
Some objects are weakly related like a computer and its peripherals

Composition

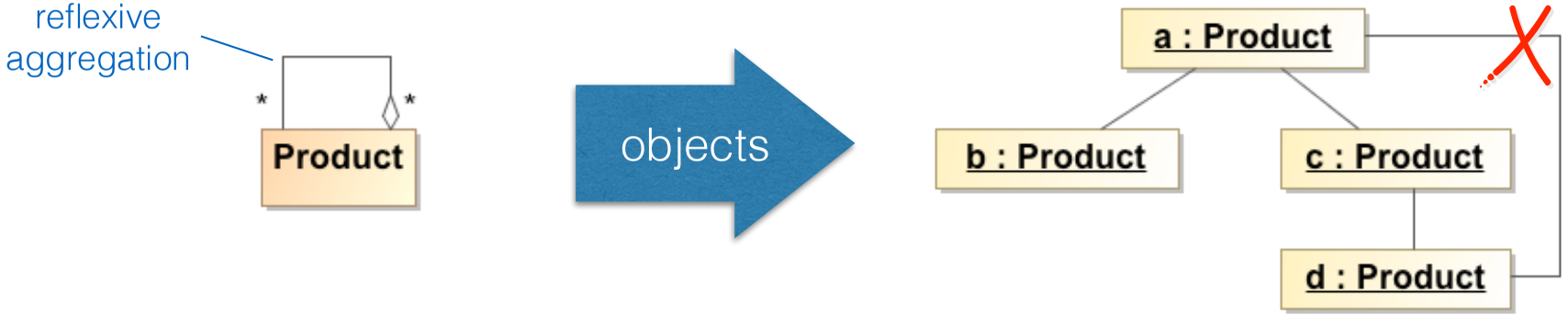
Some objects are strongly related like a tree and its leaves

- UML has two whole-part relationships - aggregation and composition
- They are *transitive* and *asymmetric*

Transitivity and asymmetry



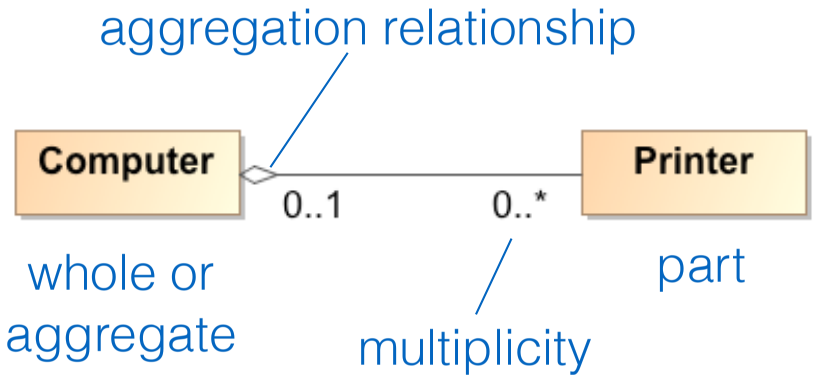
Transitivity: if C is part of B and B is part of A then C is part of A



Asymmetry: an object can never be part of itself

- Aggregation and composition are transitive and asymmetric
- Association is transitive and symmetric

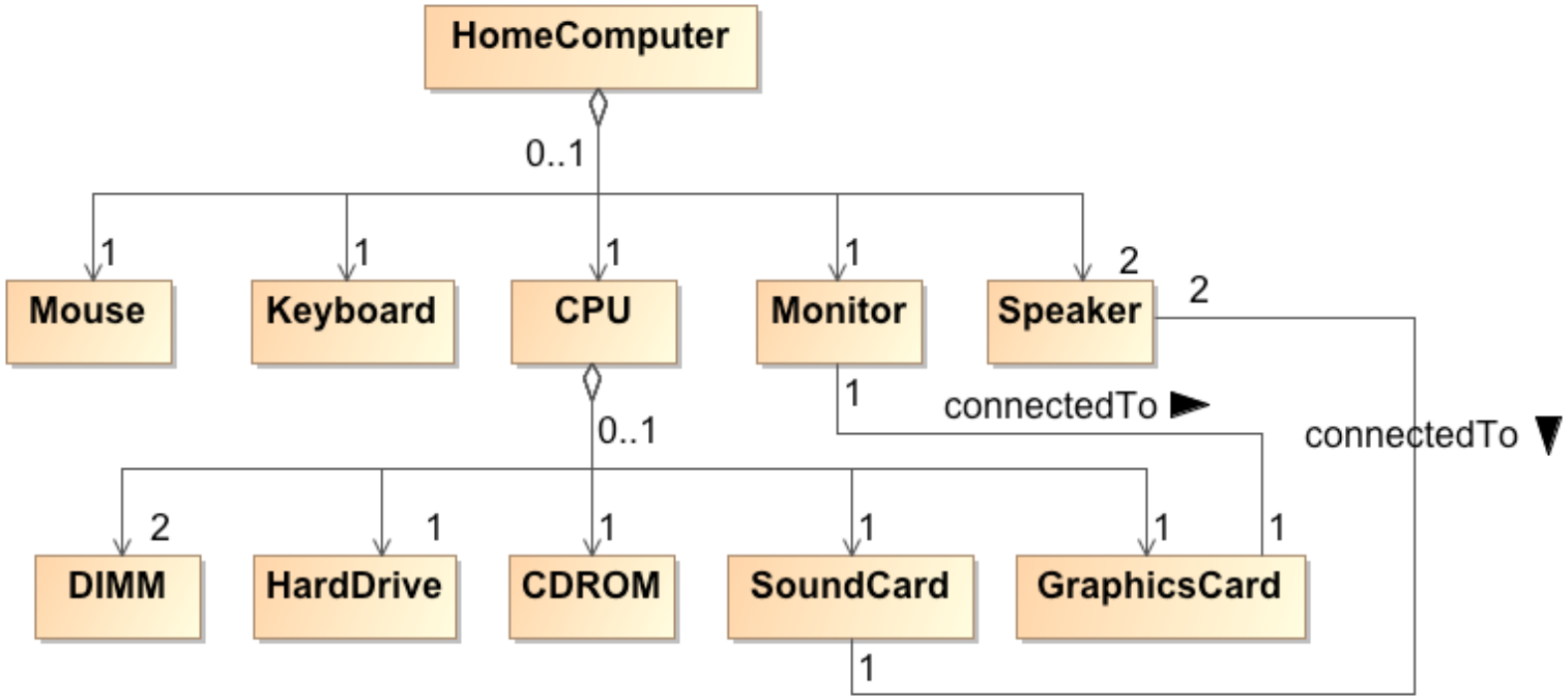
Aggregation semantics



- A Computer can be attached to 0 or more Printers
- At any point in time a Printer is connected to 0 or 1 Computers
- Over time, many Computers may use a given Printer
- The Printer exists even if there are no Computers
- The Printer is independent of the Computer

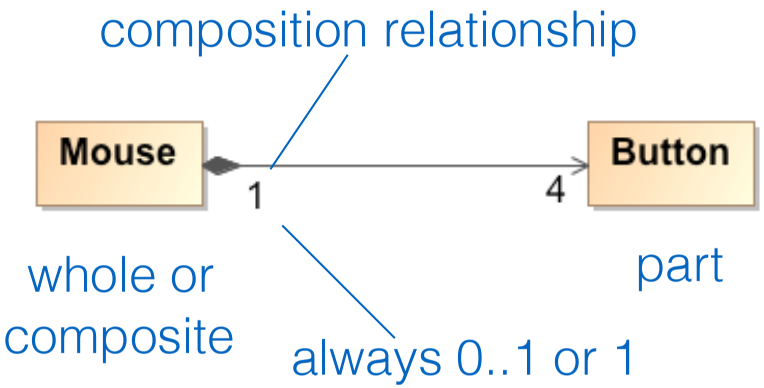
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some sense incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates

An aggregation hierarchy



- Notice how we have used multiplicities to constrain the configuration of this class of computer

Composition semantics



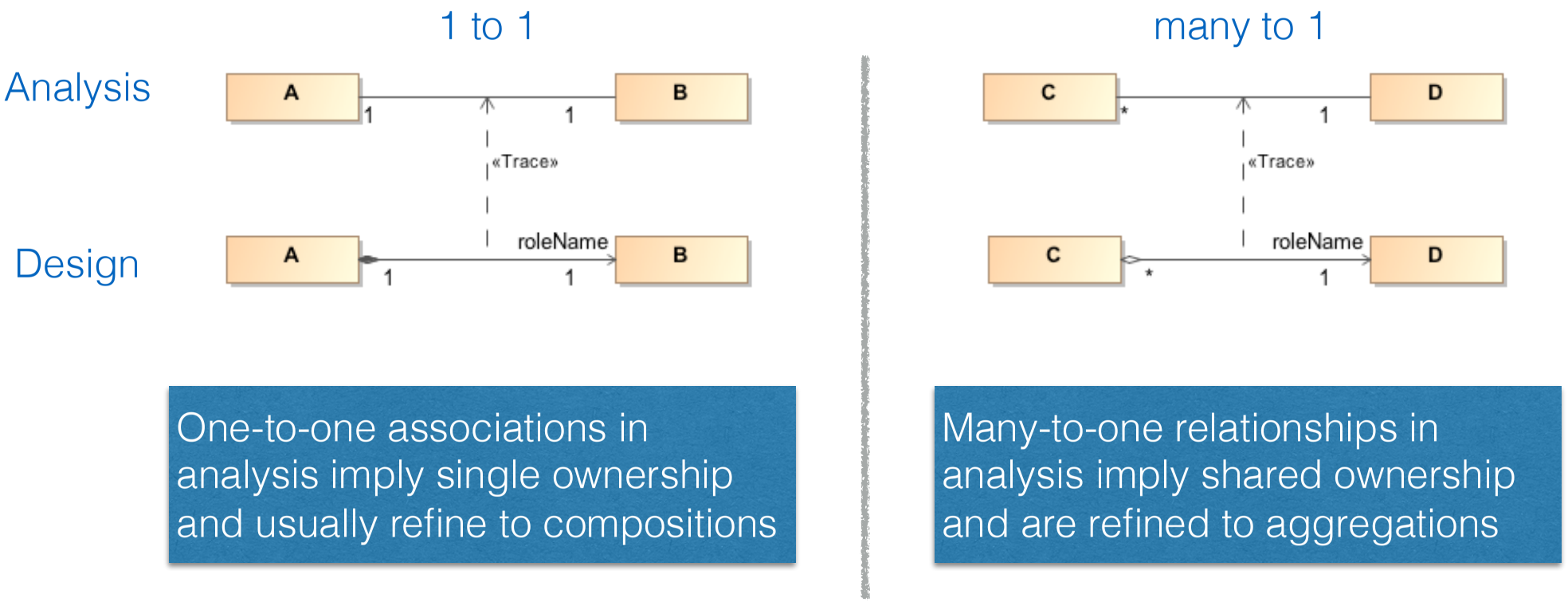
- The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse
- Each button can belong to exactly 1 mouse

- The parts belong to exactly 0 or 1 whole at a time
- The composite has *sole* responsibility for the disposition of all its parts i.e. responsibility for their creation and destruction
- The composite may release parts *provided* responsibility for them is assumed by another object
- If the composite is destroyed, it must either destroy all its parts, OR give responsibility for them over to some other object

Composition and attributes

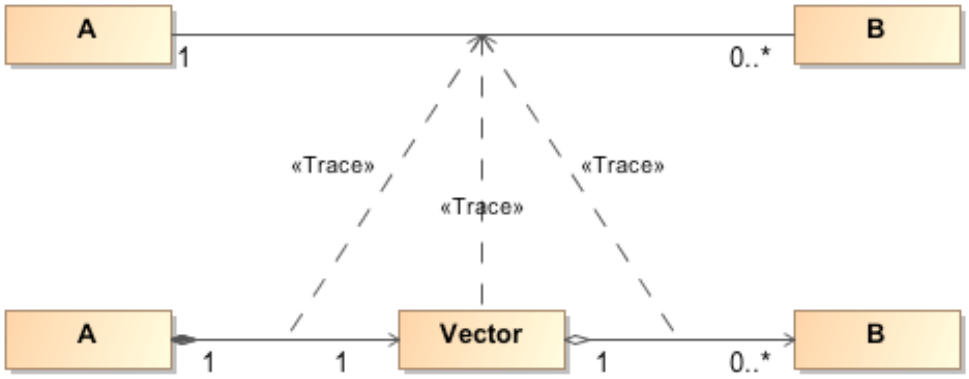
- Attributes are equivalent to composition relationships between a class and the classes of its attributes
- Attributes should be reserved for primitive data types and library classes (int, String, Date etc.)
- Compositions should be reserved for important classes

1 to 1 and many to 1 associations



- Single ownership generally implies composition, although aggregation may be used if the single ownership is weak
- Shared ownership generally implies aggregation and precludes any possibility of composition

1 to many associations



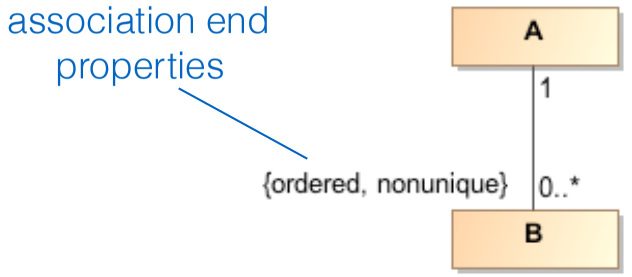
A single analysis artifact may often trace to one or more design artifacts!

- To refine 1-to-many associations we introduce a *collection class*. This is a class whose instances store a collection of references to objects of the target class
- A collection class *must* have methods for adding an object, removing an object, retrieving an object and traversing the collection of objects
- Collection classes are typically supplied in libraries that come as part of the implementation language, e.g. java.util library. Sadly, UML has no standard collections…

Collection semantics

Property	Semantics
{ordered}	Elements in the collection are maintained in a strict order
{unordered}	There is no ordering of the elements in the collection
{unique}	Elements in the collection are all unique i.e. an object appears in the collection once
{nonunique}	Duplicate elements are allowed in the collection

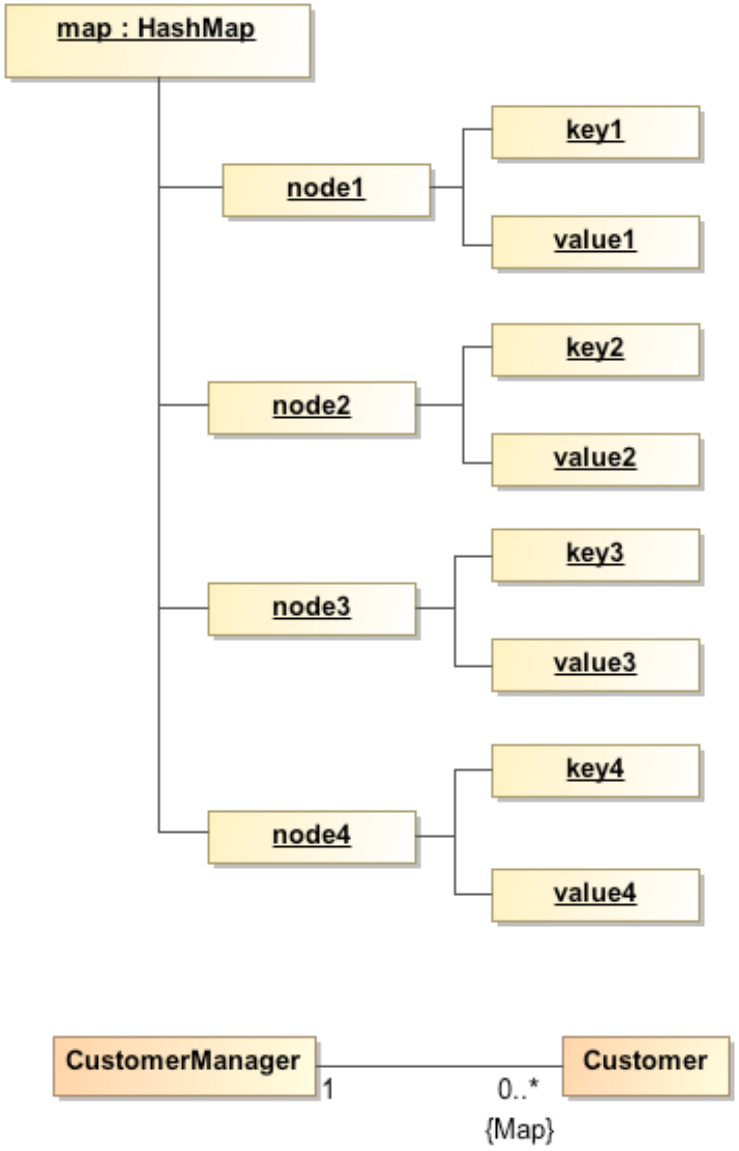
Property pair	OCL collection
{unordered, unique}	Set (the default)
{unordered, nonunique}	Bag
{ordered, unique}	OrderedSet
{ordered,nonunique}	Sequence



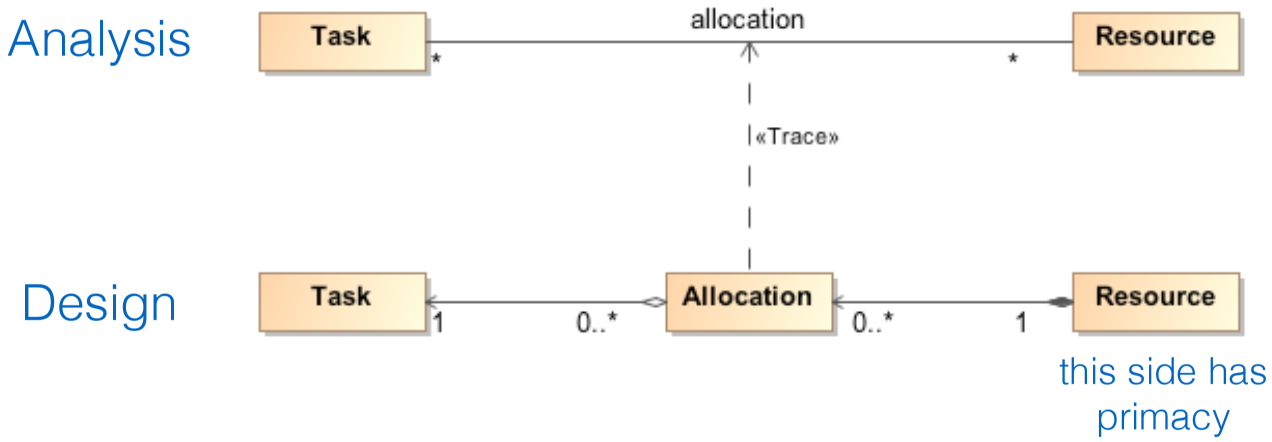
- UML can specify collection semantics using association end properties
- Note: OCL is the Object Constraint Language - it is a language embedded in UML for specifying constraints

The Map

- Maps (also known as dictionaries) have no equivalent in OCL
- Maps usually work by maintaining a set of nodes where each node points to two objects – the "key" and the “value”. They are optimized to find a value efficiently given a specific key
- A Map is like a database table with only two columns, one of which is the primary key - they are incredibly useful for storing any objects that must be accessed quickly using a key, for example customer details or products
- You can indicate a Map is required using a constraint on the association end



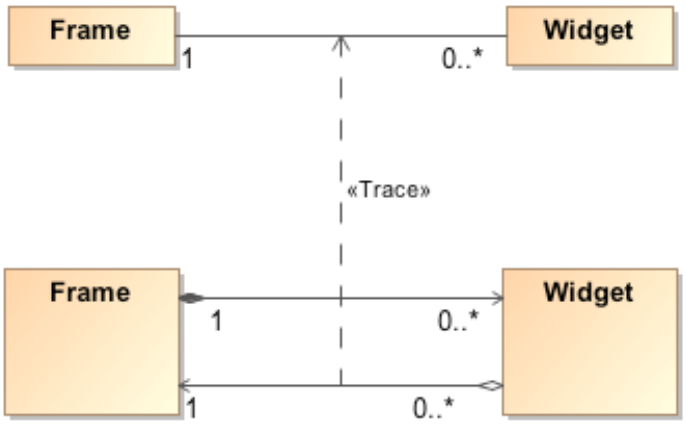
Many to many associations



- There is no commonly used OO language that directly supports many-to-many associations so we must implement these associations with design classes (reification)
- Decide which side of the association should have primacy and use composition, aggregation and navigability accordingly

Bi-directional associations

Analysis



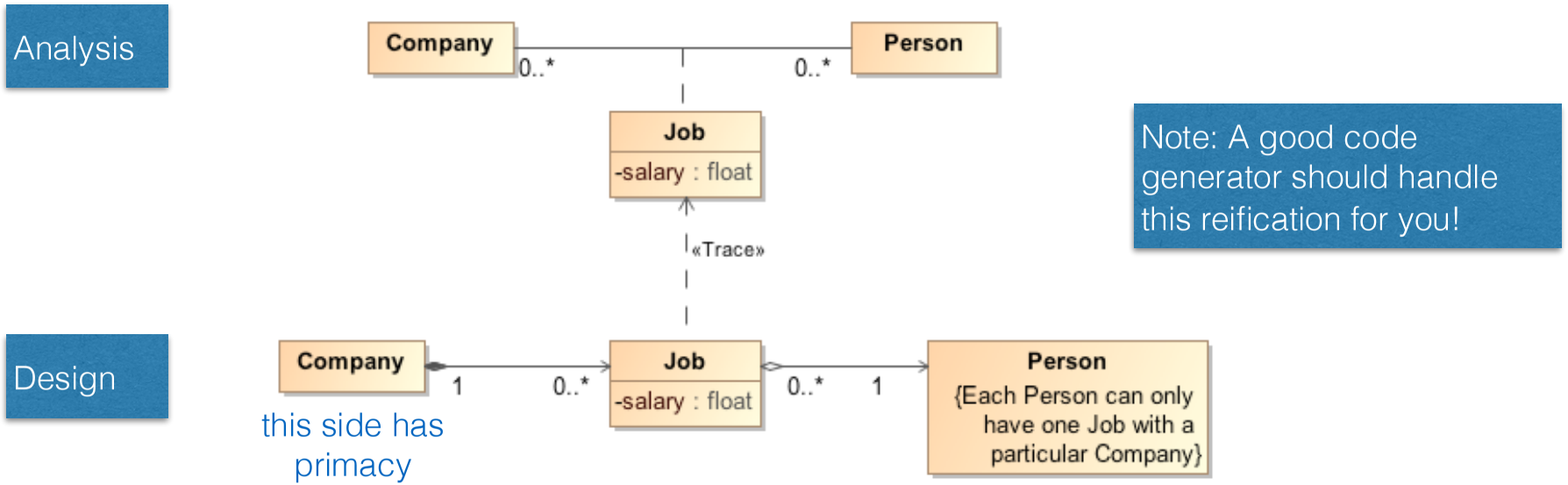
Design

this side has primacy

Note: this would be an unusually detailed design model!

- There is no commonly used OO language that directly supports bi-directional associations. We must resolve each bi-directional association into two unidirectional associations
- We must decide which side of the association should have primacy and use composition, aggregation and navigability accordingly
- Typically, you can leave this level of refinement up to the programmer or code generator!

Association classes



- There is no commonly used OO language that directly supports association classes. Refine each association class into a design class plus a constraint
- Decide which side of the association has primacy and use composition, aggregation and navigability accordingly

Summary

- In this section we have seen how we take the incompletely specified associations in an analysis model and refine them to aggregations or compositions
- Aggregation is a whole-part relationship:
  - Parts are independent of the whole
  - Parts may be shared between wholes
  - The whole is incomplete in some way without the parts
- Composition is a strong form of aggregation:
  - Parts are entirely dependent on the whole
  - Parts may not be shared
  - The whole is incomplete without the parts
- One-to-many, many-to-many, bi-directional associations and association classes may be refined in design