Object-Oriented Design

Lecturer: Raman Ramsin

Lecture 17:

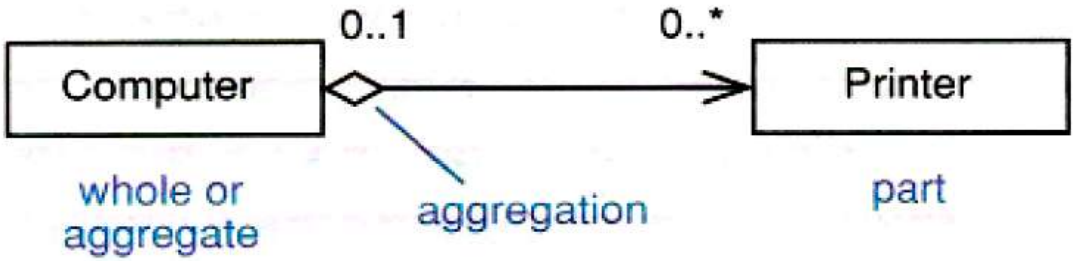
Refining Analysis Relationships

Refining Analysis Relationships

- Relationships in analysis are converted to implementable design relationships.
- Refining analysis relationships to design relationships involves:
  - adding navigability;
  - adding multiplicity to both ends of the association;
  - adding a role name at both ends of the association, or at at least on the target end of the association;
  - implementing one-to-one, one-to-many, many-to-one, and many-to-many associations;
  - implementing bidirectional associations and association classes;
  - using structured classifiers for modeling composition.

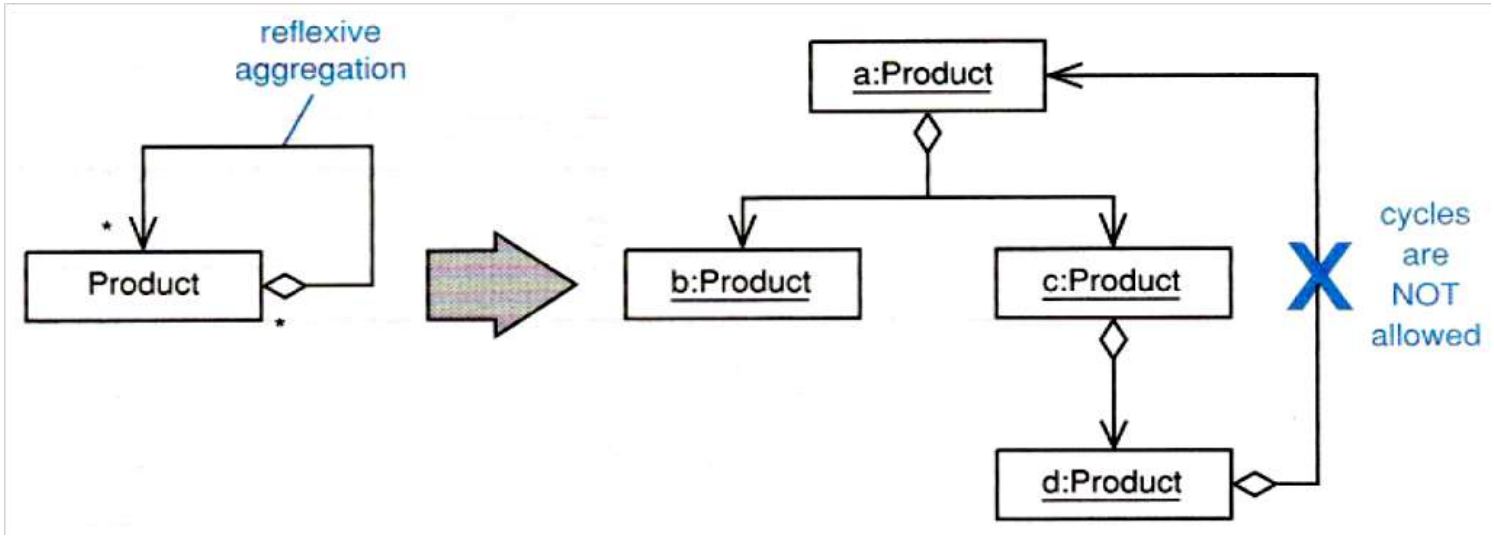
Aggregation Relationship

- Whole-part relationship where objects of one class act as the whole or *aggregate*, and objects of the other class act as the parts;
  - general semantics: Assembly, Containment, or Membership; can be used in analysis as well as design;
  - the whole uses the services of the parts; the parts service the requests of the whole;
  - the whole is the dominant, controlling side of the relationship; the part tends to be more passive;
  - aggregation is transitive: If C is part of B and B is part of A, then C is part of A.



Aggregation Relationship: Asymmetry

- Aggregation relationship is asymmetric:
  - a whole can never directly or indirectly be a part of itself;
  - there must never be a cycle in the aggregation graph.



Aggregation and Composition

- There are two types of aggregation relationship:
  - **Aggregation**;
  - **Composition Aggregation** - usually referred to simply as **Composition**.

Aggregation

Some objects are weakly related, like a computer and its peripherals

Composition

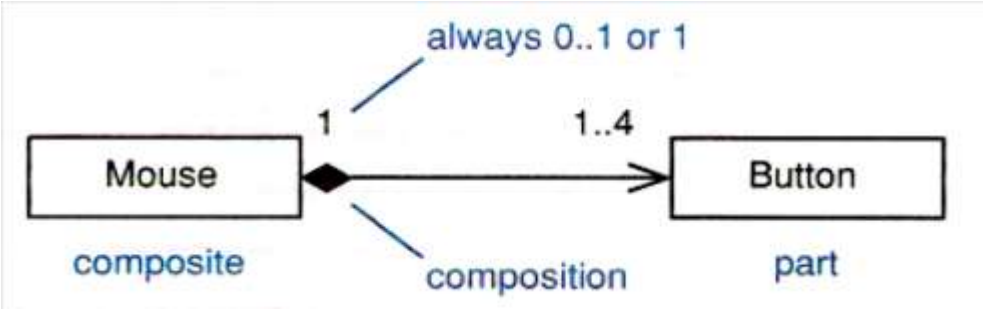
Some objects are strongly related, like a tree and its leaves

Aggregation: Semantics

- "Aggregation" is a weak Whole-Part relationship (like a computer system and its peripherals);
- The aggregate can sometimes exist independently of the parts, sometimes not;
- The parts may exist independently of the aggregate;
- It is possible to have shared ownership of the parts by several aggregates;
- Aggregation hierarchies and aggregation networks are possible;
- The whole always knows about the parts, but if the relationship is one-way from the whole to the part (which is typically the case), the parts don't know about the whole.

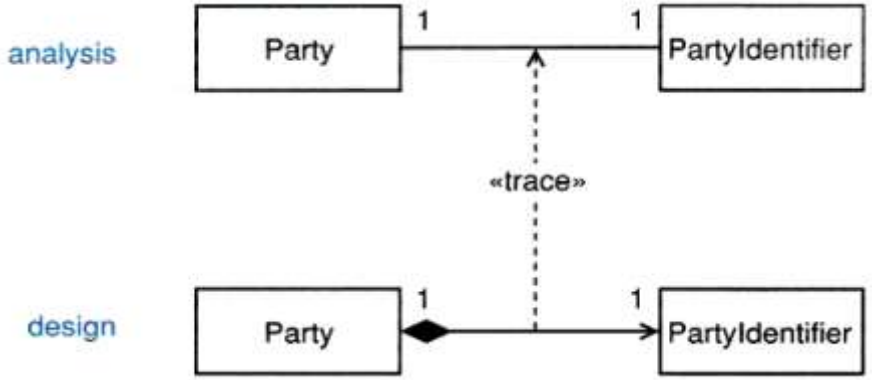
Composition

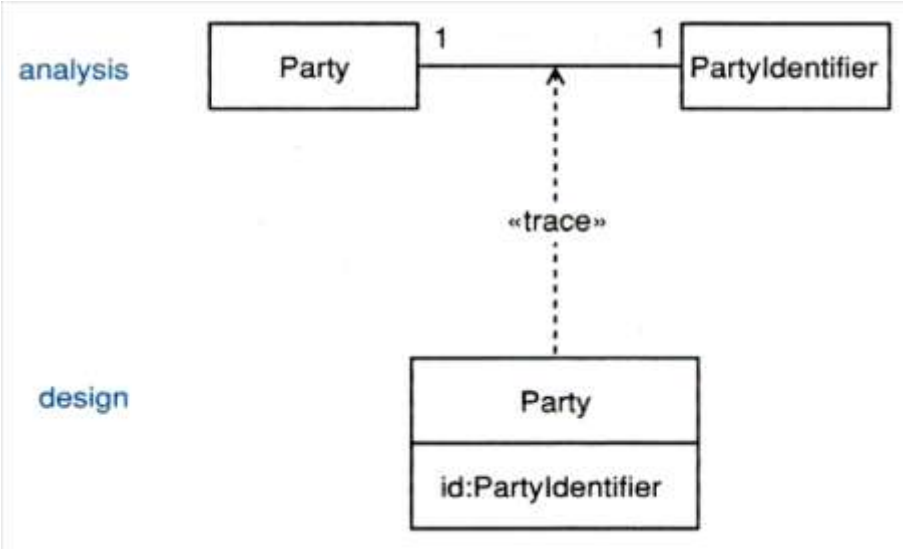
- A strong form of aggregation (like a tree and its leaves):
  - the parts belong to exactly one *composite* at a time;
  - the composite has sole responsibility for the disposition of all its parts - this means responsibility for their creation and destruction;
  - the composite may also release parts, provided responsibility for them is assumed by another object;
  - if the composite is destroyed, it must destroy all its parts or give responsibility for them over to some other object;
  - each part belongs to exactly one composite so you can only have composition hierarchies - composition networks are impossible.



Refining Analysis Relationships: One-to-One Association

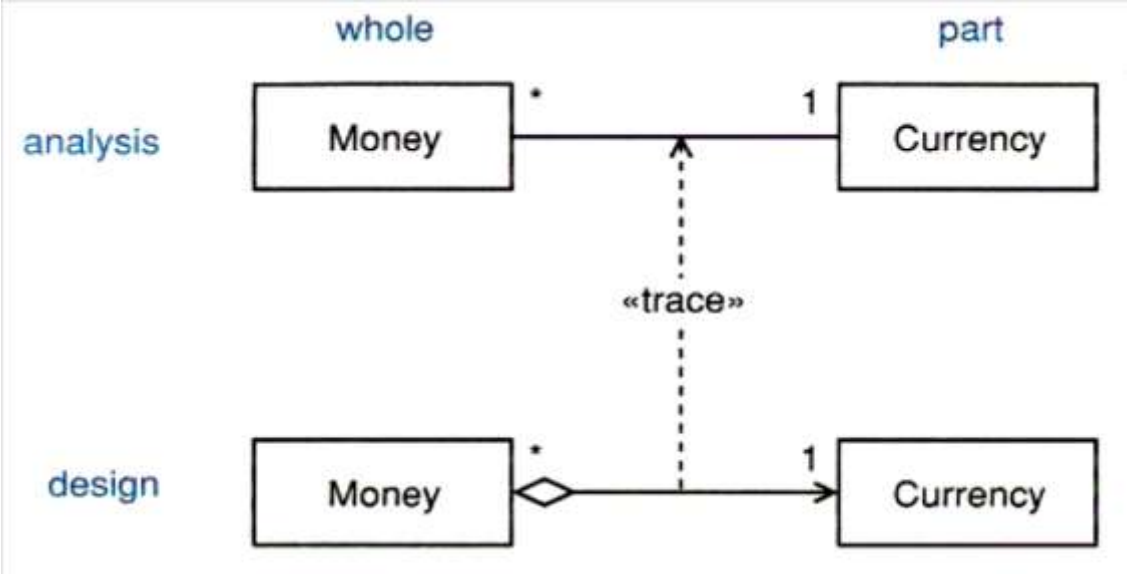
- Add navigability to the model; refine into Composition only if the semantics apply; you may also choose to merge the two classes.





Refining Analysis Relationships: Many-to-One Association

- Add navigability; refine into Aggregation only if the semantics apply.



Refining Analysis Relationships: One-to-Many Association

- There is a collection of objects on the target side:
  - Use an inbuilt array (most OO languages directly support arrays) - they are generally quite inflexible but are usually fast.
  - Use a collection class - they are more flexible than inbuilt arrays and are faster than arrays when searching the collection is required (otherwise they are slower).

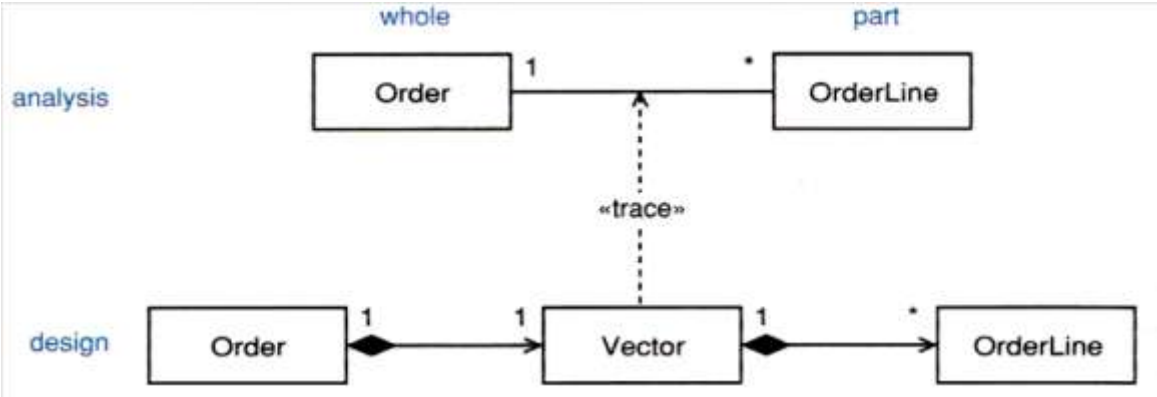
Collections

- Classes specialized so that their instances can manage a collection of other objects.

- All collection classes have operations for:
  - adding objects to the collection;
  - removing objects from the collection;
  - retrieving a reference to an object in the collection;
  - traversing the collection - stepping through the collection from the first object to the last.

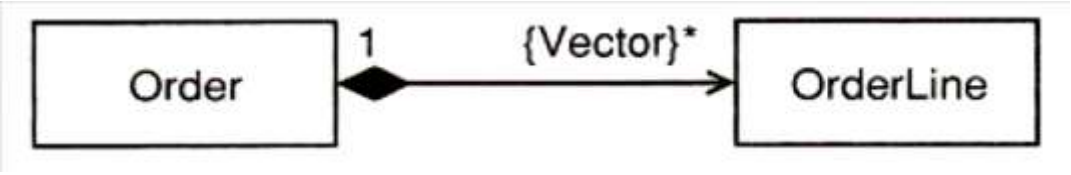
Modeling with Collections

- There are four options:
  1. model the collection class explicitly;



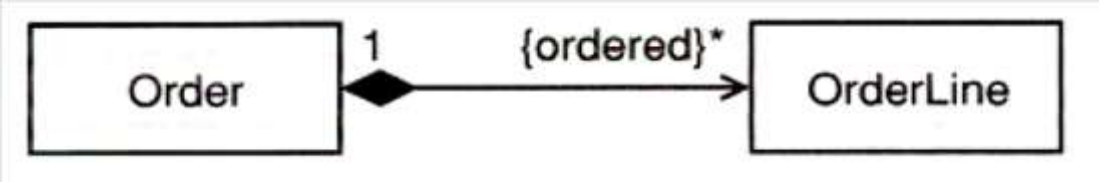
Modeling with Collections

- There are four options:
  1. model the collection class explicitly;
  2. tell the modeling tool which collection to use by adding a property to the relationship;



Modeling with Collections

- There are four options:
  1. model the collection class explicitly;
  2. tell the modeling tool which collection to use by adding a property to the relationship;
  3. tell the programmer what collection semantics are required by adding a property to the relationship;



Modeling with Collections

- There are four options:
  1. model the collection class explicitly;
  2. tell the modeling tool which collection to use by adding a property to the relationship;
  3. tell the programmer what collection semantics are required by adding a property to the relationship;

Standard 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 – an object appears in the collection at most once
{nonunique}	Duplicate elements are allowed in the collection

Modeling with Collections

- There are four options:
  1. model the collection class explicitly;
  2. tell the modeling tool which collection to use by adding a property to the relationship;
  3. tell the programmer what collection semantics are required by adding a property to the relationship;

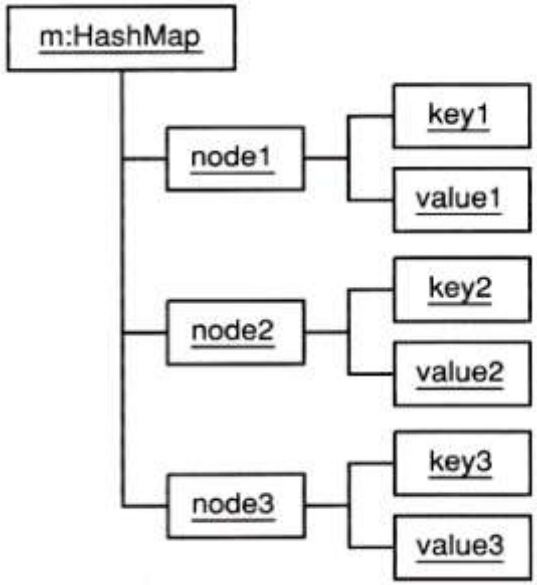
Property	OCL collection
{unordered, nonunique}	Bag
{unordered, unique}	Set
{ordered, unique}	OrderedSet
{ordered, nonunique}	Sequence

Modeling with Collections

- There are four options:
  1. model the collection class explicitly;
  2. tell the modeling tool which collection to use by adding a property to the relationship;
  3. tell the programmer what collection semantics are required by adding a property to the relationship;
  4. instead of refining one-to-many relationships to collection classes, leave it up to the programmers.
- Don't "overmodel" - the choice of a specific collection class is often a tactical issue that can be left to the programmer at implementation time.

Modeling with Collections: The *Map*

- Also known as the *dictionary*;
- Given a key, the corresponding value may be found very quickly;
- Acts like a database table with two columns, the key and the value;
- Keys must be unique.

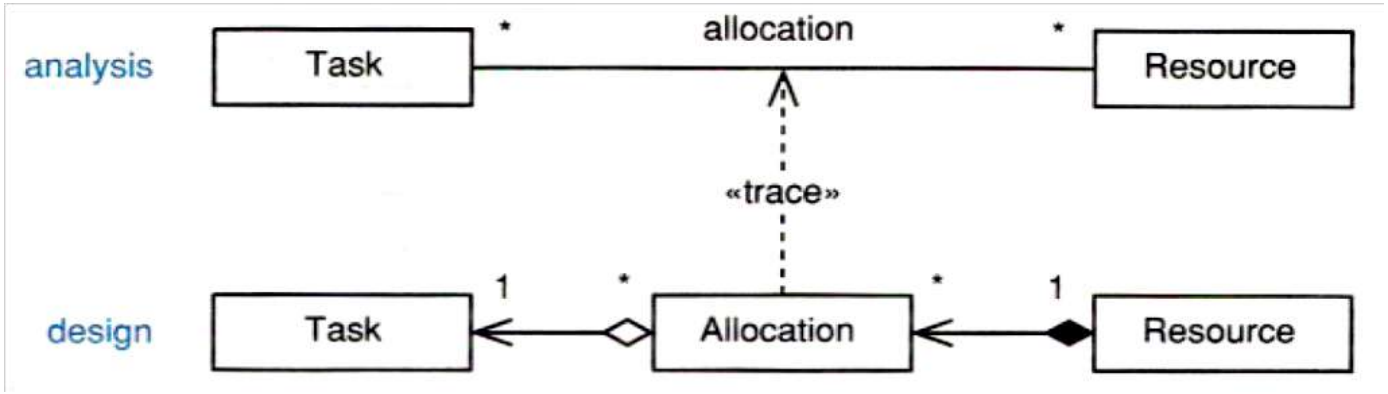


Reifying Analysis Relationships

- Some relationships are pure analysis artifacts and must be made implementable by the process of reification:
  1. Many-to-many associations
  2. Bidirectional associations
  3. Association classes

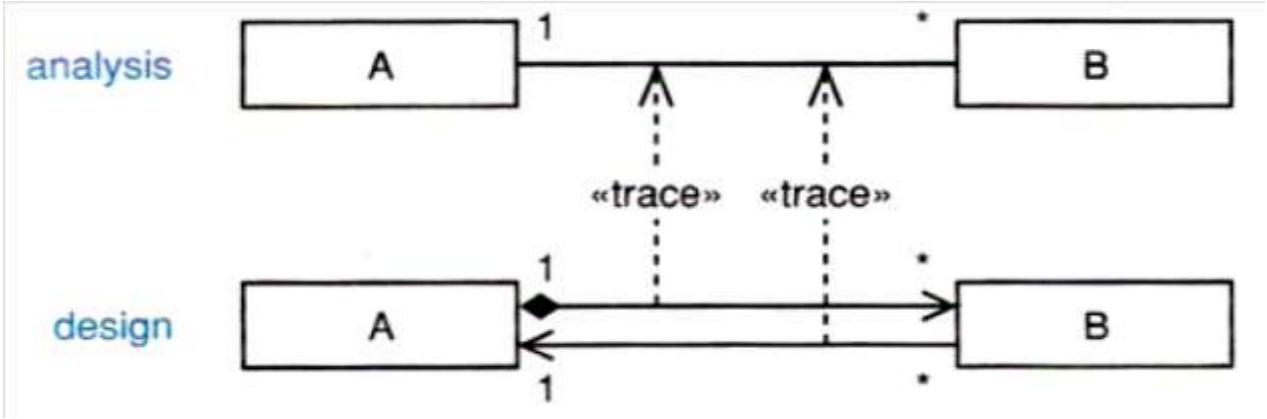
Reifying Relationships: Many-to-Many Associations

1. Reify the relationship into a class.
2. Add navigability; Refine into Aggregation or Composition only if the semantics apply.



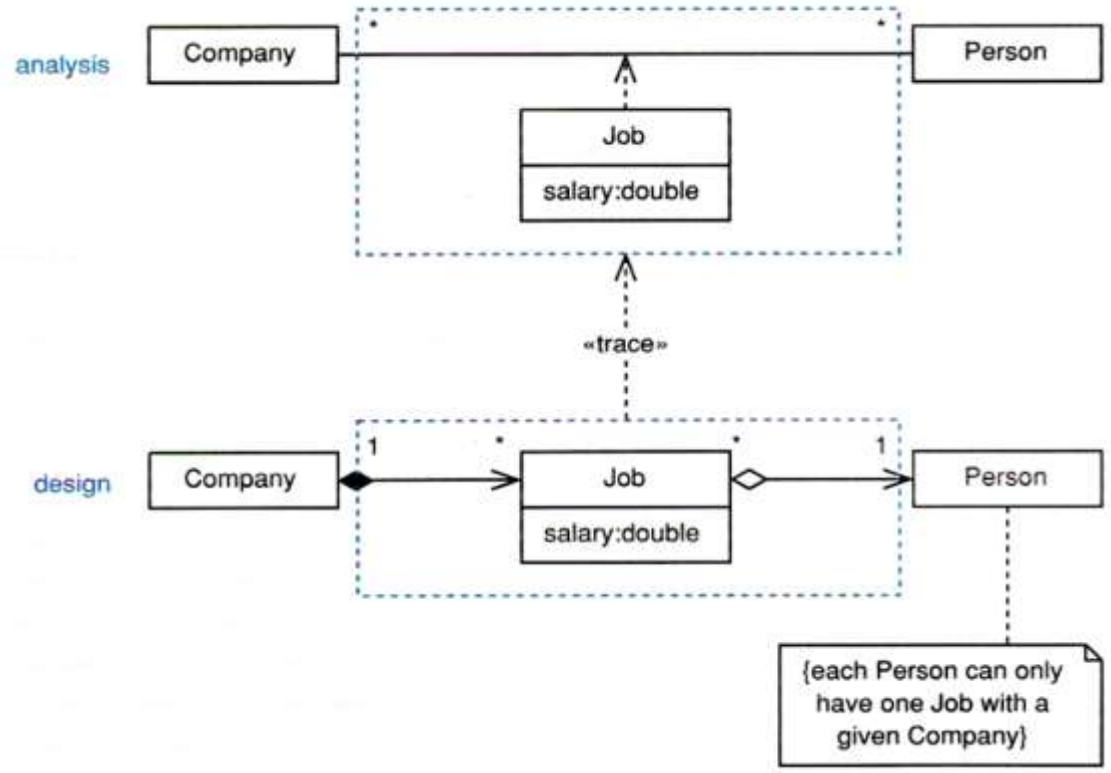
Reifying Relationships: Bidirectional Associations

- Replace with two unidirectional associations or a reified class; refine into Aggregation or Composition only if the semantics apply.



Reifying Relationships: Association Classes

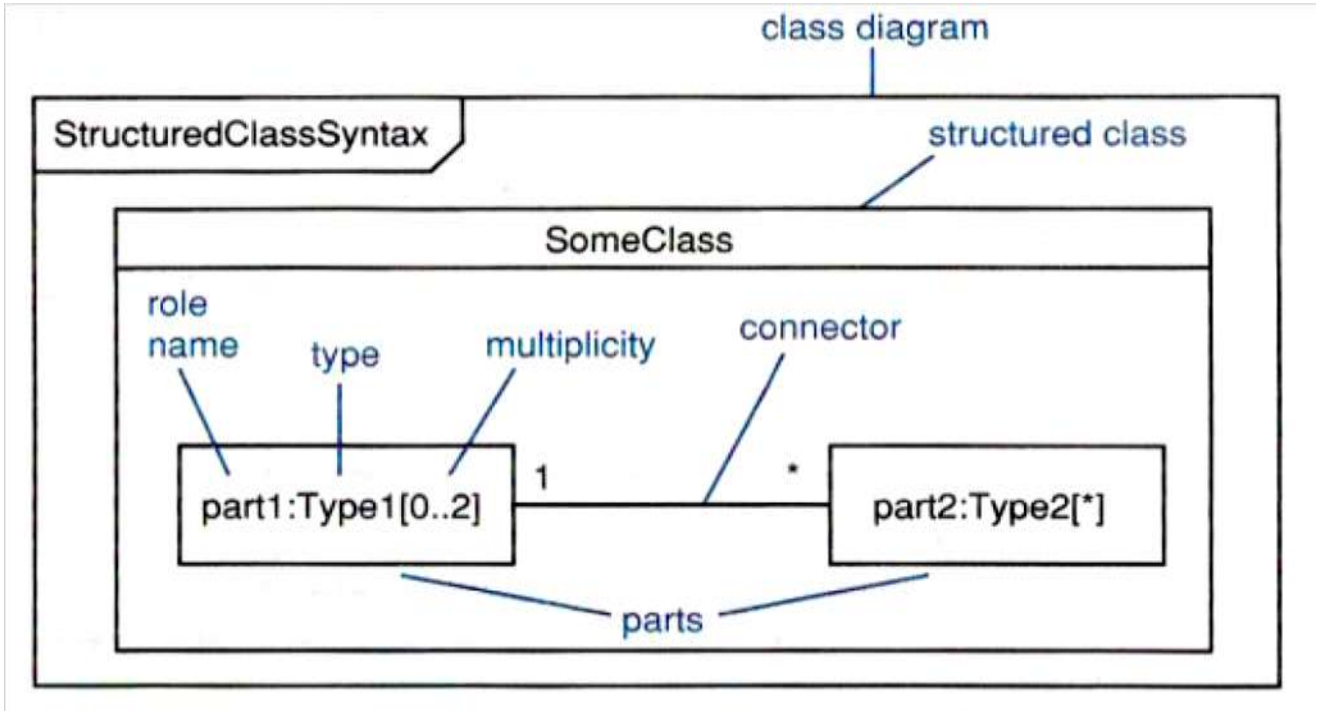
1. Replace with a class (usually with the same name as the association class);
2. Add a constraint in a note to indicate that objects on each end of the reified relationship must form a unique pair.



Structured Classifier

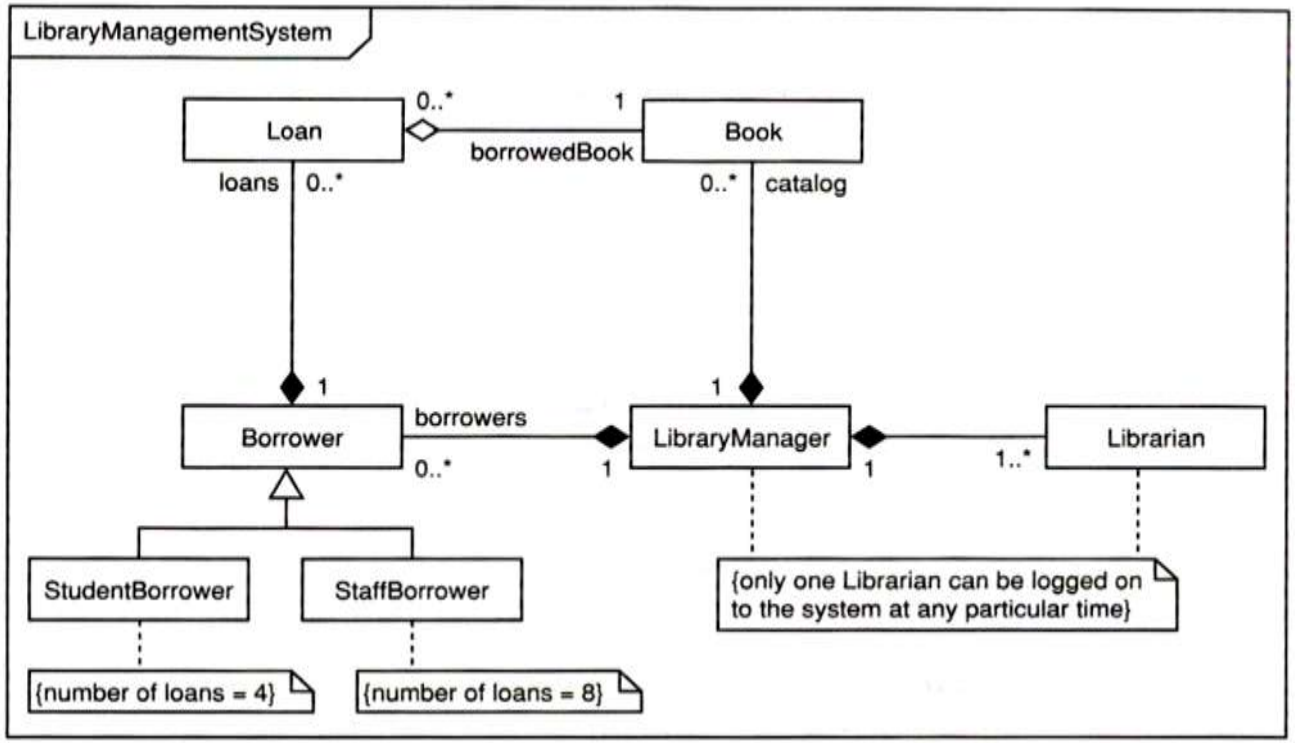
- A classifier (such as a class) that has an internal structure.
- Modeled as parts that are joined with connectors:
  - part - a *role* that one or more instances of a classifier can play in the context of the structured classifier;
  - name - the name of the part;
  - type - the type of objects that can play the role;
  - multiplicity - the number of objects that can play the role at any time;
  - connector - a relationship between parts in the context of a structured classifier.

Structured Classifier: Notation



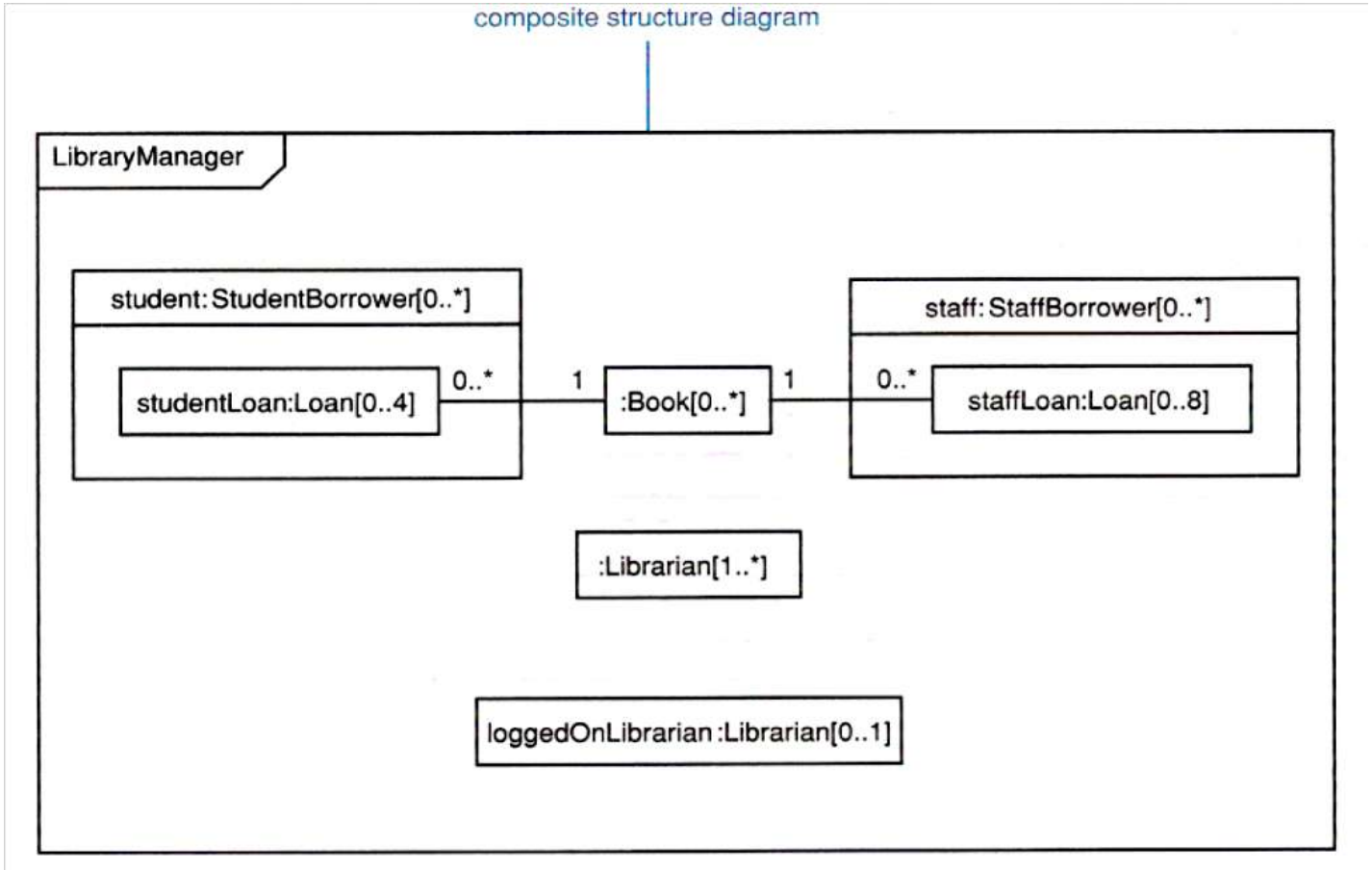
Structured Class

- A class that has an internal structure;
- Has the extra constraint that it owns (has an implicit containment relationship with) all of its parts, connectors, and ports.



Structured Class: Modeling

- Internal structure can be shown on class diagrams or on a composite structure diagram.



Reference

- Arlow, J., Neustadt, I., *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, 2nd Ed. Addison-Wesley, 2005.