



# Module

# Design Class Diagrams

Vasco Amaral  
[vma@fct.unl.pt](mailto:vma@fct.unl.pt)

---

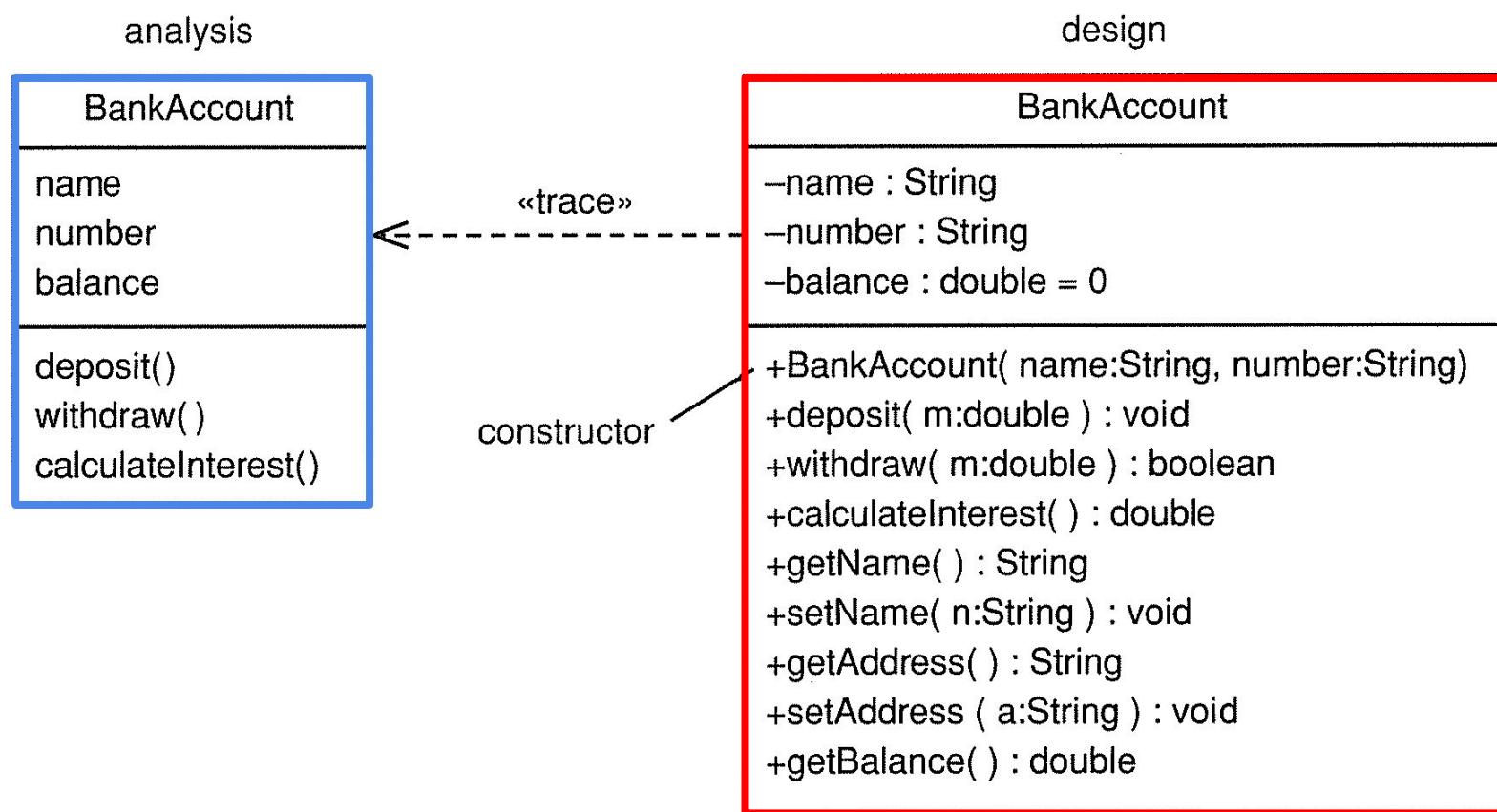
# Design classes

# Design classes are classes whose specifications have been completed so that they can be implemented

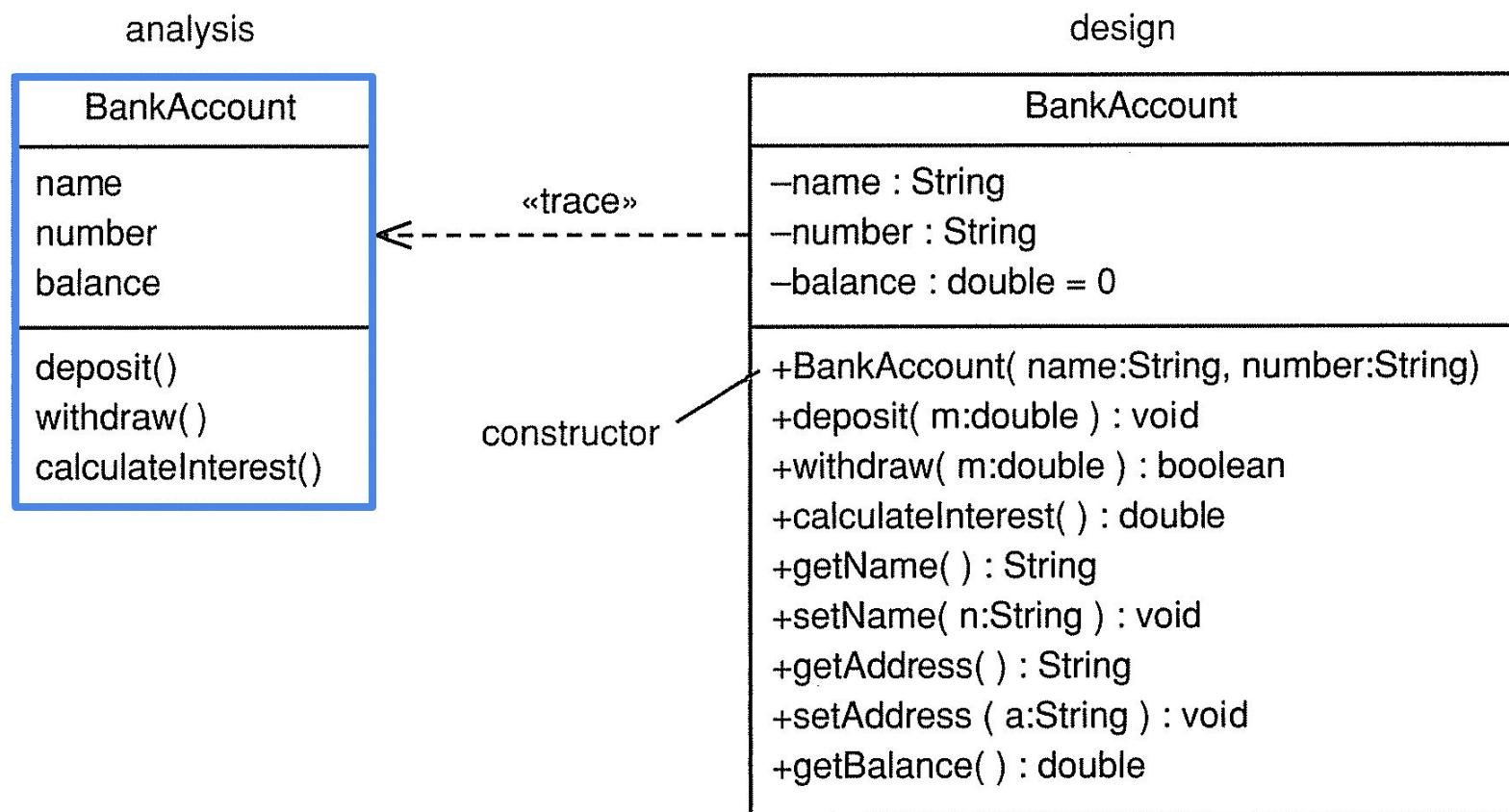
---

- Analysis is about what the system should do
- Design is about how that behavior may be implemented
- Design classes come from
  - The problem domain, via refinement of analysis classes
  - The solution domain, where you can find utility classes, reusable components, GUI frameworks, etc

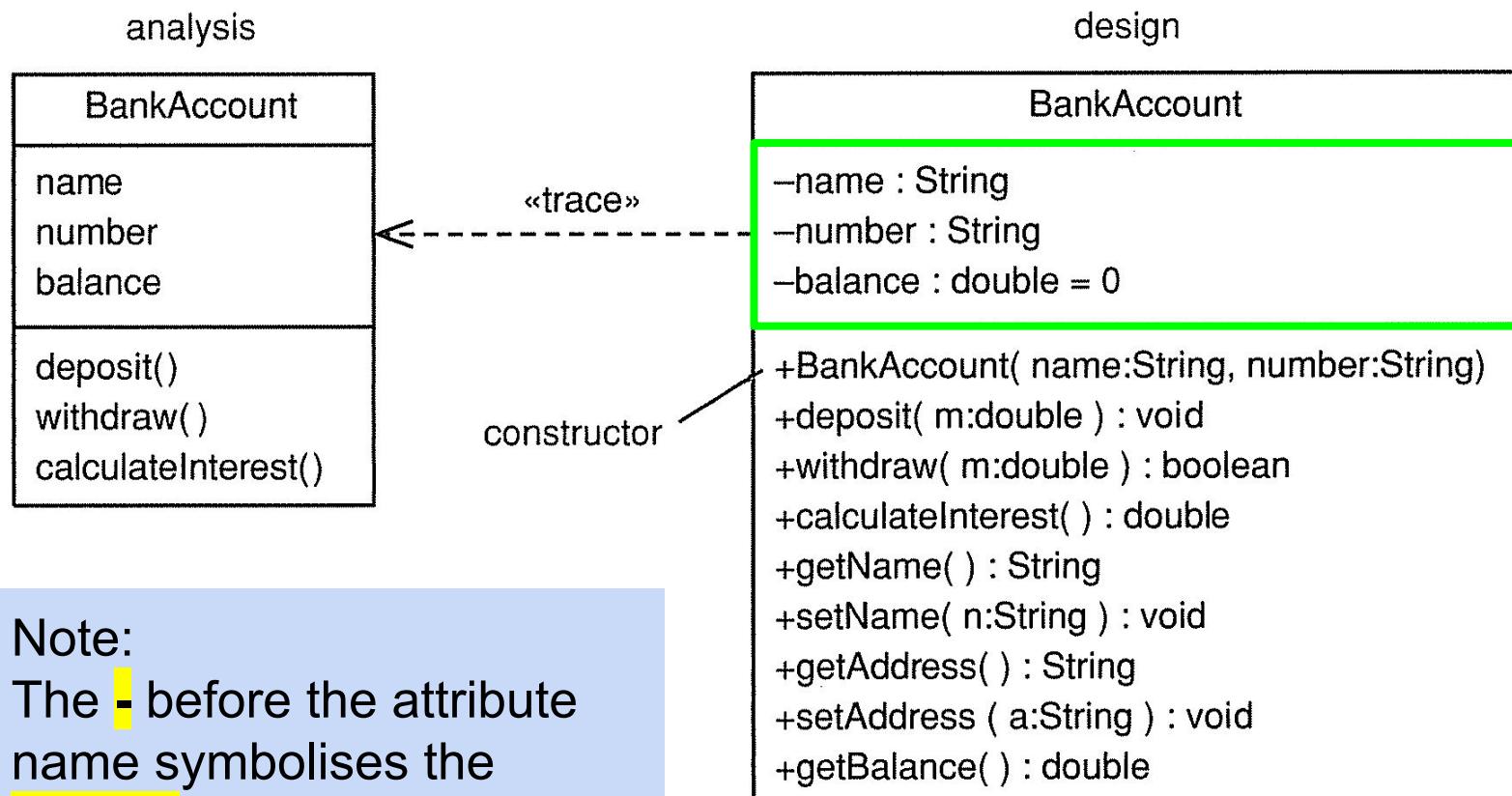
# Design classes can be refined from analysis classes



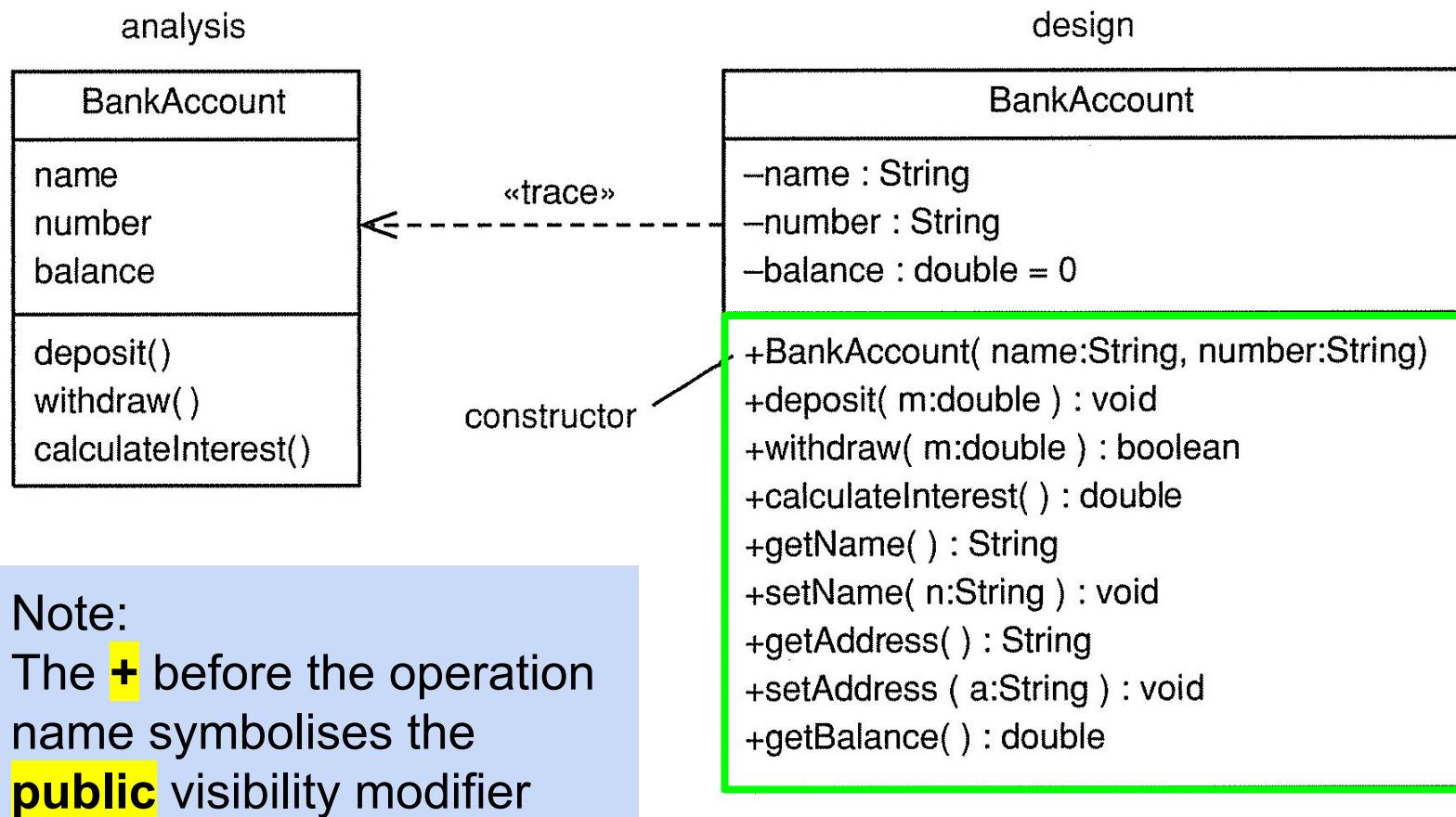
# In the analysis classes we only had a class name, some high-level attributes and operations



In the **design classes** we have a **complete set of attributes** fully specified with name, type, visibility (and optionally a default value)



In the **design classes** we have a **complete set of operations** fully specified with name, parameter list, return type, and visibility modifier



# Well-formed design classes are passed to developers, or used directly for code generation

---

- A class should be assessed from the perspective of its users
- To be well-formed, the class should
  - be complete and sufficient
  - be primitive
  - have a high cohesion of its features
  - have a low coupling with other classes

# Completeness and sufficiency

---

- The public operations of a class define a **contract** between the class and clients of the class
- A **complete** and **sufficient** class gives the class users of the class exactly what they expect - no more, no less
  - **Completeness** - a class should satisfy all reasonable client expectations
  - **Sufficiency** - a class should be as simple and focused as possible

# Primitiveness

---

- A class should **not offer multiple ways of doing the same thing**
  - This is confusing to clients
  - This can lead to **future maintenance and consistency problems**
- Services should be:
  - Simple
  - Atomic
  - Unique
- Aim for the simplest and smallest set of operations
- Add to this set only if you have a proven case for doing so

# High cohesion

---

- Each class should capture a single, well-defined abstraction, using the minimal set of features
- All features (operations and attributes) are designed to implement a small, focused set of operations
- Cohesive classes tend to be
  - Easy to understand
  - Easy to reuse
  - Easy to maintain

# Low coupling

---

- A class should be **associated to a minimal number of other classes** to allow it to fulfill its responsibilities
- Many associations come directly from the analysis mode
- Other associations are introduced by **implementation constraints, or by the wish to reuse code** - the latter need to be examined carefully
- Some coupling is desirable
  - High coupling within a subsystem indicates that the subsystem is cohesive
  - High coupling between subsystems indicates that the system is likely hard to maintain and evolve

# Inheritance

---

- During analysis, you should only use inheritance where there is a clear and unambiguous “is-a” relationship between analysis classes
- During design, you may also consider using inheritance in a tactical way, to reuse code
  - Inheritance is used here to facilitate the implementation of the child class - use this with discretion

---

# Refining analysis relationships

# Analysis associations must be refined to design relationships that are implementable in the target OO language

---

- Many relationships between analysis classes are not directly implementable as-is
  - Bi-directional associations are not directly supported
  - Association classes are not directly supported
  - Many-to-many associations are not directly supported

# Refinements

---

- Associations to aggregation, or composition relationships, where appropriate
- Implementing one-to-many associations
- Implementing many-to-one associations
- Implementing many-to-many associations
- Implementing bidirectional associations
- Implementing association classes

# Desirable features of design associations

---

- Mandatory features for design associations
  - navigability
  - multiplicity on both ends
- Recommended features for design associations
  - association name, or
  - role name at least on the target end

---

# Aggregation and composition

# Aggregation and composition

## Aggregation

A loose type of relationship between objects (e.g. a computer and its peripherals)



## Composition

A very strong type of relationship between objects (e.g. a tree and its leaves)

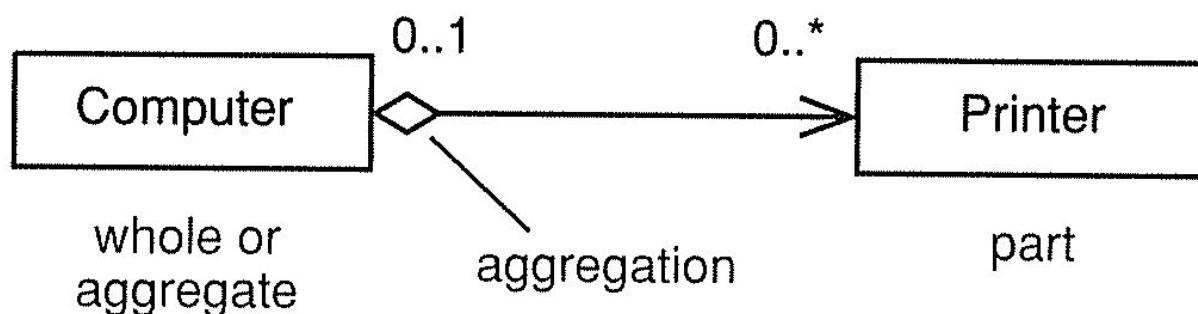


---

# Aggregation

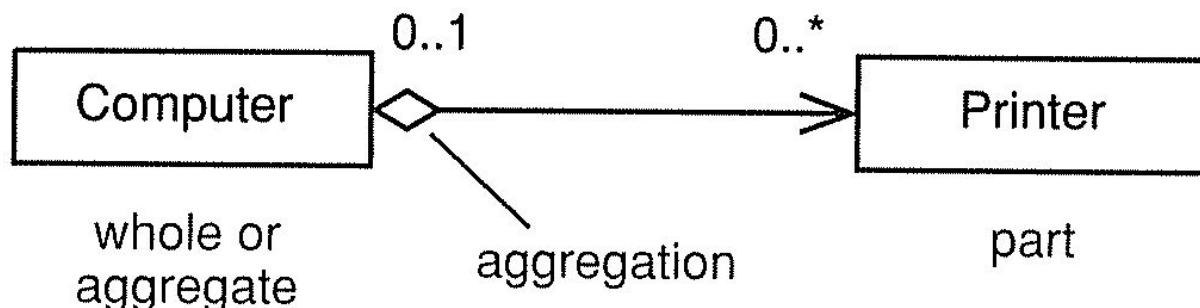
# Aggregation is a whole-part relationship

- A type of whole part relationship where
  - The aggregate is made of many parts
  - The aggregate uses services of its parts
  - The aggregate plays the role of the whole
  - If you only have navigability from the whole to the part, the part is not even aware that it is part of the whole



# Aggregation example semantics

- A computer may be attached to 0 or more printers
- At any one point in time, a printer is connected to at most one computer
- Over time, many computers may use a given printer
- The printer may exist even if there are no attached computers
- The printer is, really, independent from the computer

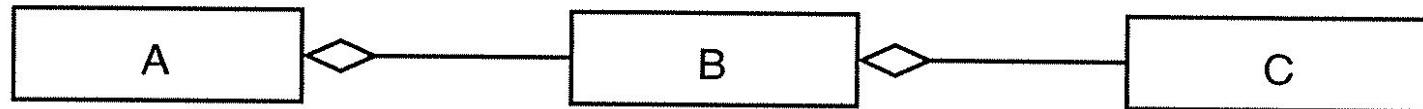


# Aggregation semantics summary

---

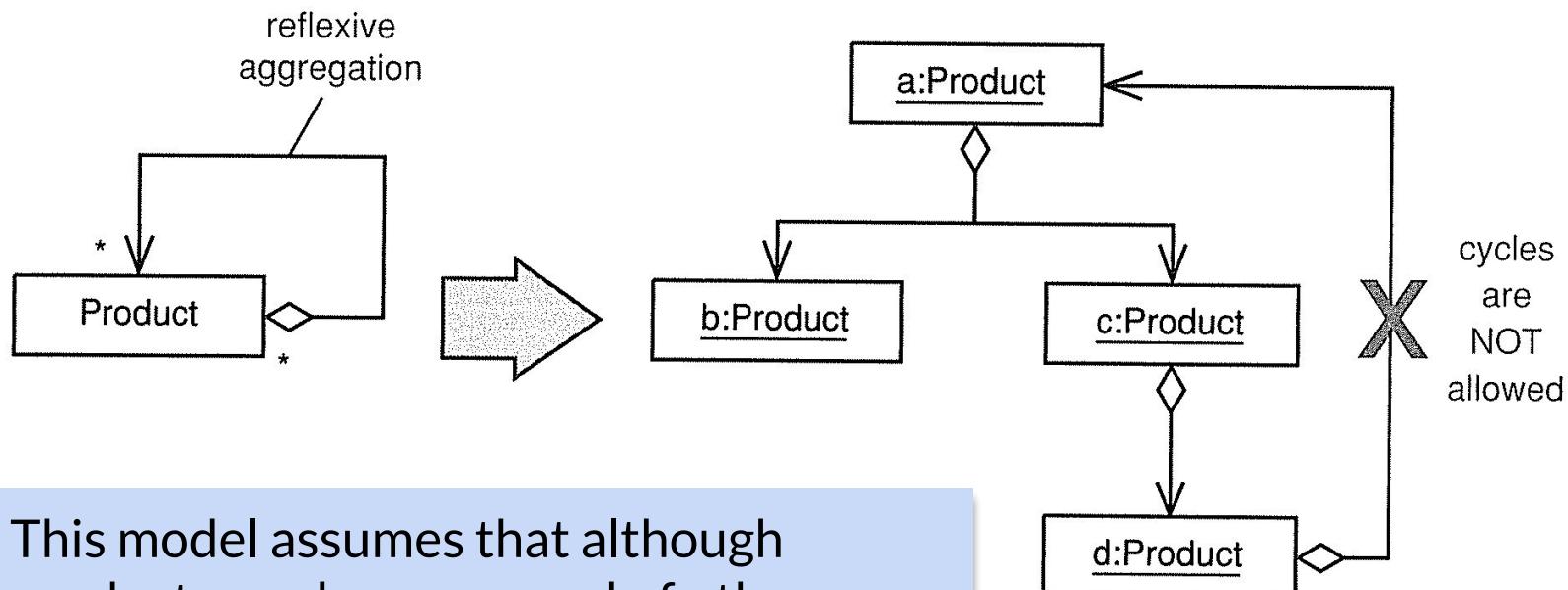
- The aggregate can sometimes exist independently from the parts, sometimes not
- The parts can always exist independently of the aggregate
- The aggregate is, in some sense, incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates

# Aggregation is transitive



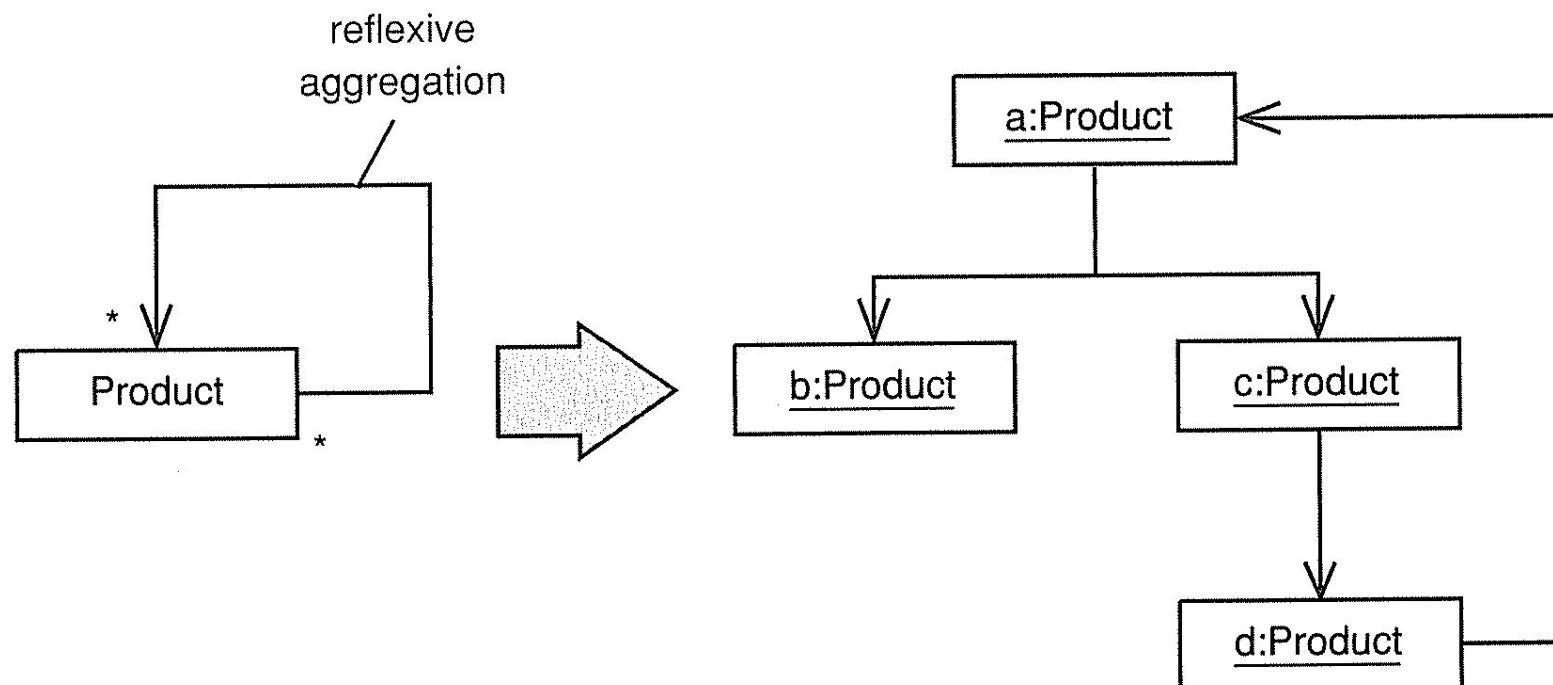
aggregation is transitive: if C is part of B and B is part of A, then C is part of A

# Aggregation is asymmetric - no object can be part of itself



This model assumes that although products can be composed of other products, these are all different products and the asymmetry constraint is preserved.

# Association is symmetric - an object can be associated to itself



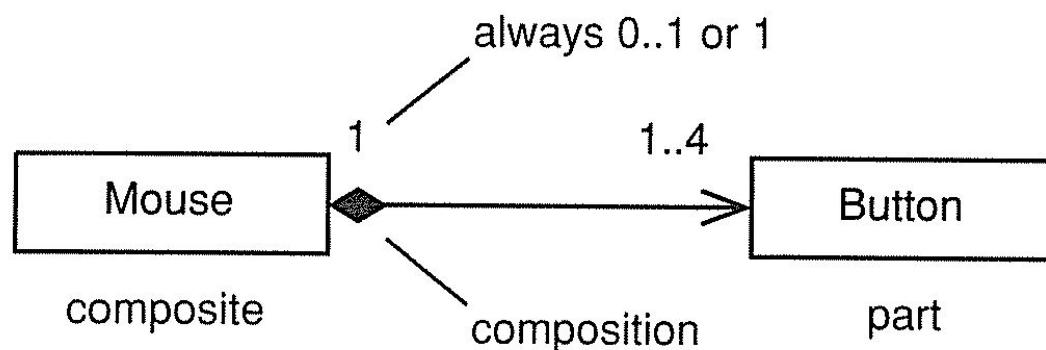
---

# Composition

# Composition is a stronger form of aggregation

---

- Like aggregation, it is a whole-part relationship, transitive and asymmetric
- Unlike aggregation, in composition the **parts have no independent life outside of the whole**
- In composition, **each part belongs to at most one and only one whole**
- **If you destroy the whole object, you also destroy all its parts**



# Composition semantics

---

- The parts can only belong to one composite at a time - there is no possibility of shared ownership of a part
- The composite has sole responsibility for the disposition of all its parts - this means 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

# Aggregation vs composition

## Aggregation

You may form reflexive aggregation hierarchies and networks - an object can be part of more than one aggregate



## Composition

You may only have reflexive composition hierarchies - an object can only be part of one composite at any point in time



# Composition and attributes: choose the most adequate for increased clarity, usefulness and readability of the model

---

- A part in a composite is equivalent to an attribute
  - An attribute can be seen as a composition relationship between the composite class and the class of the attribute
- However, we do use both ways of expressing the same thing, often in the same diagrams. Why?
  - Attributes may be of primitive data types. You could stereotype them as <>primitive<>, but this would just clutter the model, so primitive type attributes are always modelled as attributes
  - There are several utility classes, like Time, Date, or String, that are used very often. Again, although you could model these as well, this would clutter the diagram, so these too, are modelled as attributes

---

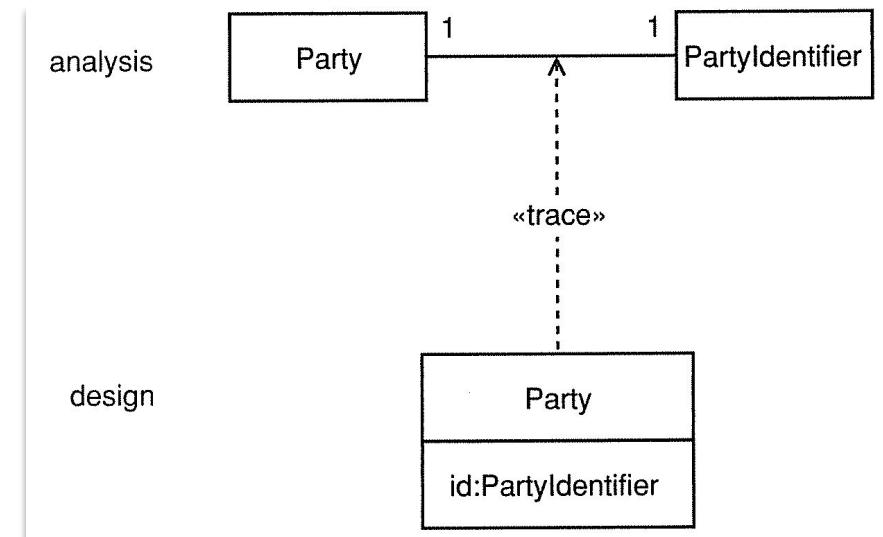
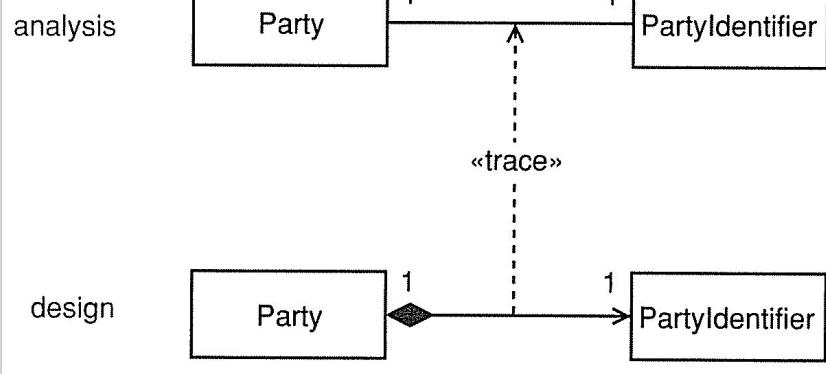
# Refining analysis relationships

# Analysis associations should be refined into aggregation, or composition, when possible

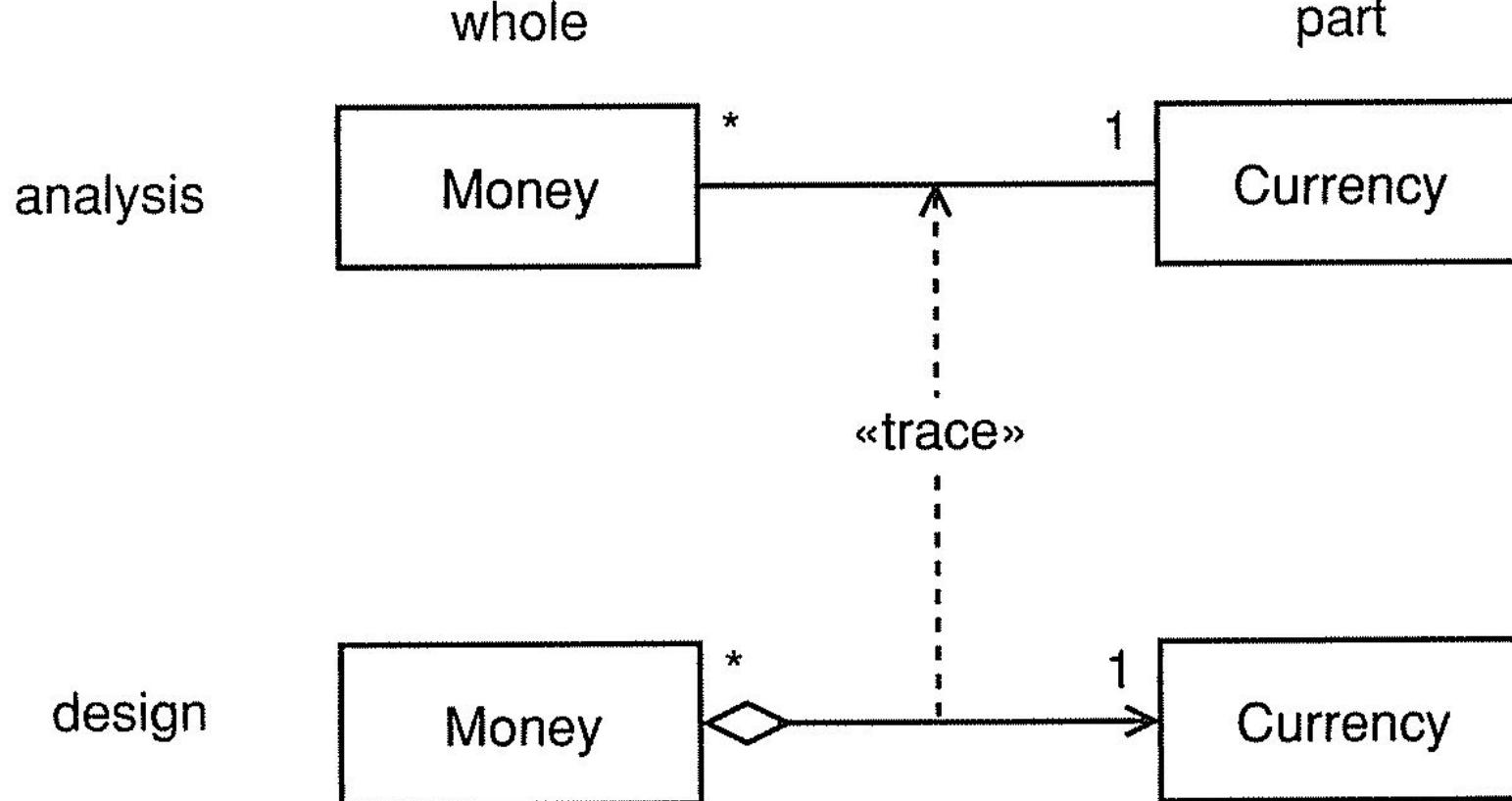
---

- Redefine associations into aggregation or composition, where possible
  - The exception is when there would be a cycle in the aggregation graph
- Add multiplicities and role names to the association, if they are absent
- Decide which side of the association is the whole, and which is the part
- Look at the multiplicity of the whole side
  - If it is 0..1, or exactly 1, you may be able to use composition
  - Otherwise, use aggregation
- Add navigability from the whole to the part
  - Design associations must be unidirectional

# One-to-one associations often lead to composition, or even an attribute



**Many-to-one associations lead to aggregation, provided there are no cycles in the aggregation graph**



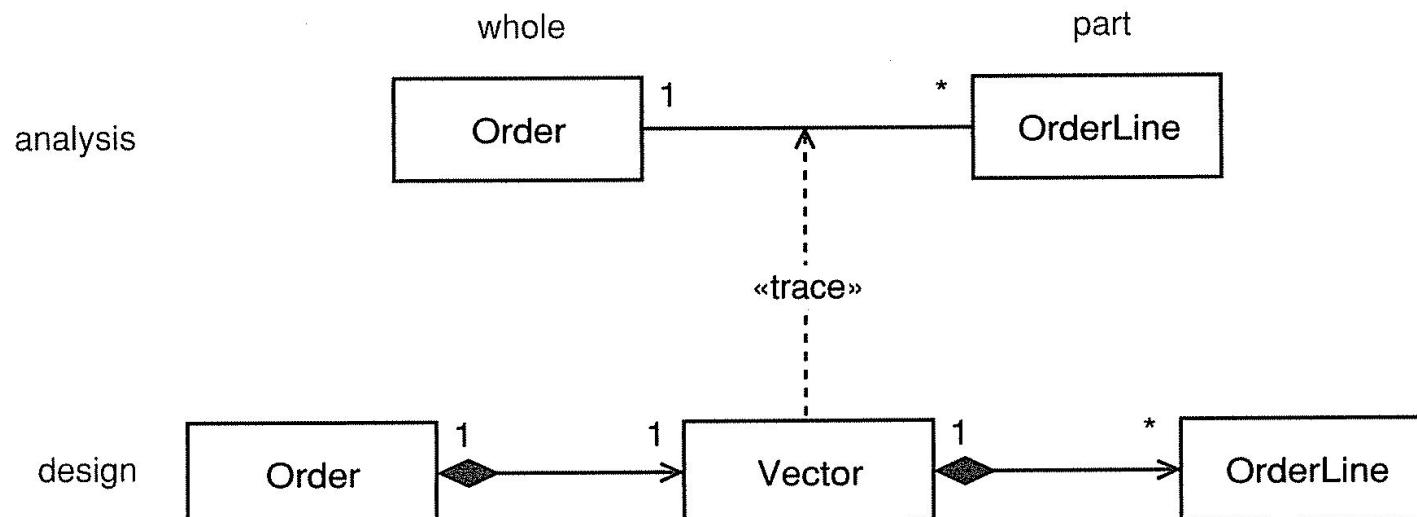
# One-to-many associations

---

- There is a collection of objects on the part side of the relationship
  - To implement it, you either use native collection support
    - An array
    - A collection class

# Collections

- A class whose instances specialize in managing collections of objects
- Collection classes typically support
  - adding objects to the collection
  - removing objects from the collection
  - retrieving a reference to an object within the collection
  - traversing the collection, by visiting its objects one by one - an iterator



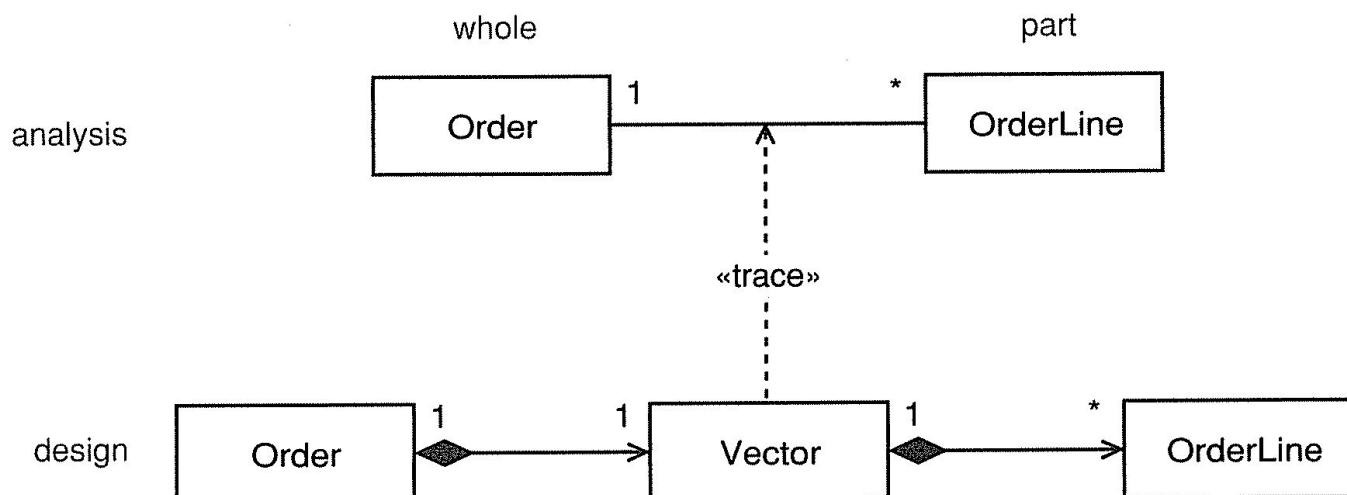
# Strategies when modelling with collections

---

1. Model the collection explicitly
2. Use a tagged value
3. Specify the desired semantics rather than an implementation class
4. Leave it to the programmers

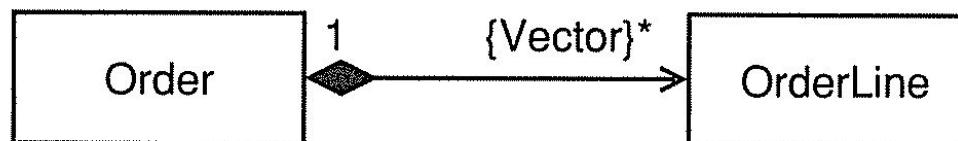
# 1. Model the collection explicitly

- Advantage - it is explicit
- Disadvantage - adds a lot of clutter to the model
  - Choice of collection class is usually a tactical programming decision - leave it to the programmer to decide
  - If, for some reason, the choice of collection class is strategic design rather than a tactical decision, then specify it



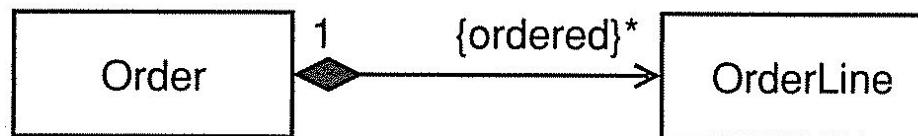
## 2. Use a tagged value

- Specify how each one-to-many association should be implemented
- Add a tagged value to the association specifying the code generation properties for each one-to-many association



### 3. Specify the desired semantics rather than an implementation class

- You can specify the semantics of the collection using a property tag, leaving the implementation details to the programmer
  - Advantages: concise, giving precise enough information to the programmer
  - Disadvantage: not enough for automatic code generation



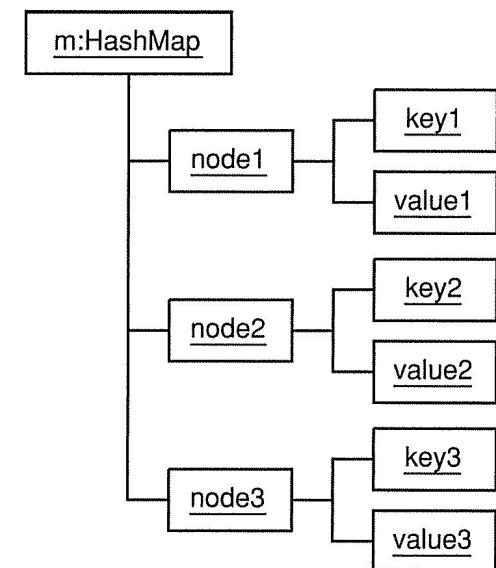
# Properties can be combined

Basic 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 unique - no repeated objects
{nonunique}	Duplicate elements allowed in the collection

Combined Property	OCL collection (other languages have something similar)
{unordered, nonunique}	Bag
{unordered, unique}	Set
{ordered, unique}	OrderedSet
{ordered, nonunique}	Sequence

# What if you want a Map?

- Maps are optimized to quickly return a value, given a key
- UML does not have a standard way of representing maps
- You can still represent a map with a tagged value, but it is a good idea to add a note to your diagram explaining it, as it is not standard
  - {map, keyname} (this is implementation agnostic)
  - {HashMap} (this uses a Java HashMap)
  - {SortedMap} (this uses a Java SortedMap)
  - ...



## 4. Don't bother refining one-to-many collection classes - leave it to the programmers



Hey! hey!  
Don't bother,  
undefined, undefined, undefined, undefined...

---

# Reified relationships

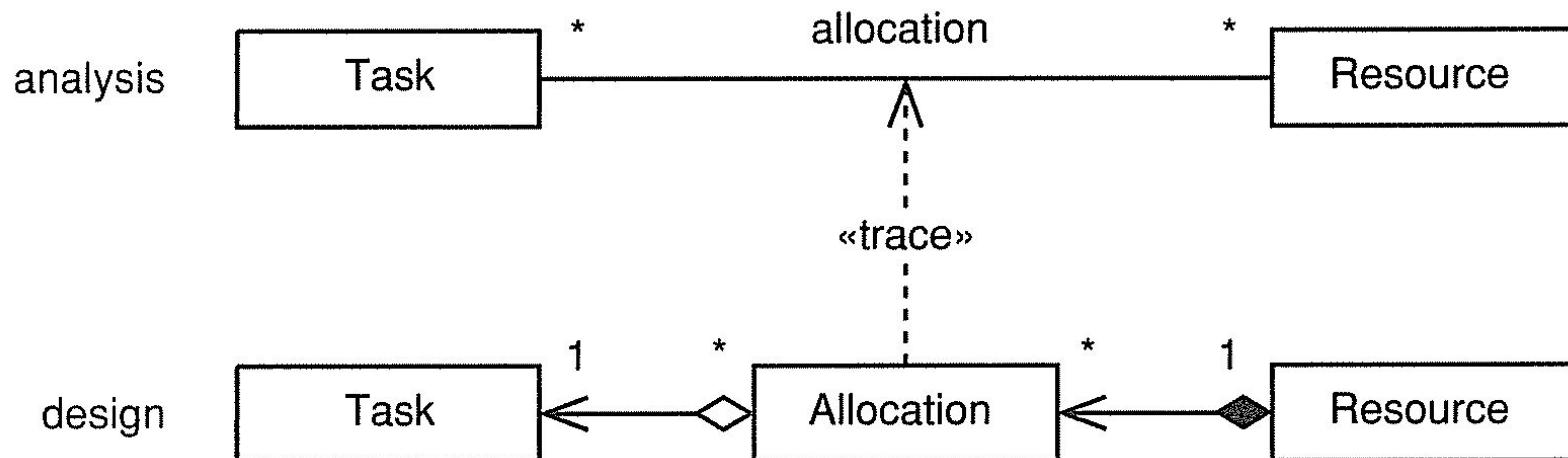
# Reification means to make concrete, or real

---

- You need to reify the following analysis relationships, as they are not supported by common OO languages
  - many-to-many associations
  - bidirectional associations
  - association classes

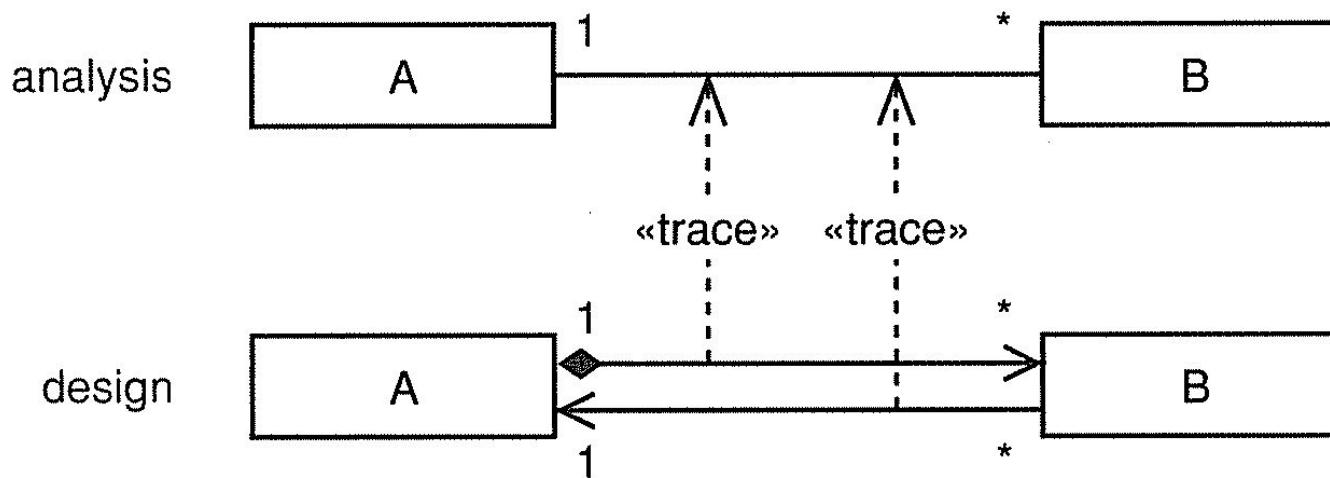
# Many-to-many associations

- Supported by some object databases
- Not supported directly in most OO languages
  - They need to be reified into normal classes, aggregations, compositions, and dependencies
- In analysis you could be vague about ownership and navigation details
- In design, you need to be precise
- In this particular example, we opted by a resource-centric design

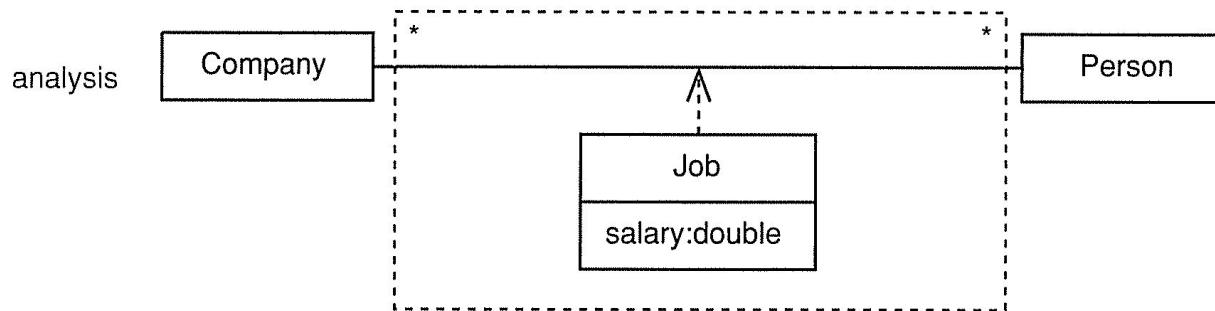


# Bidirectional associations

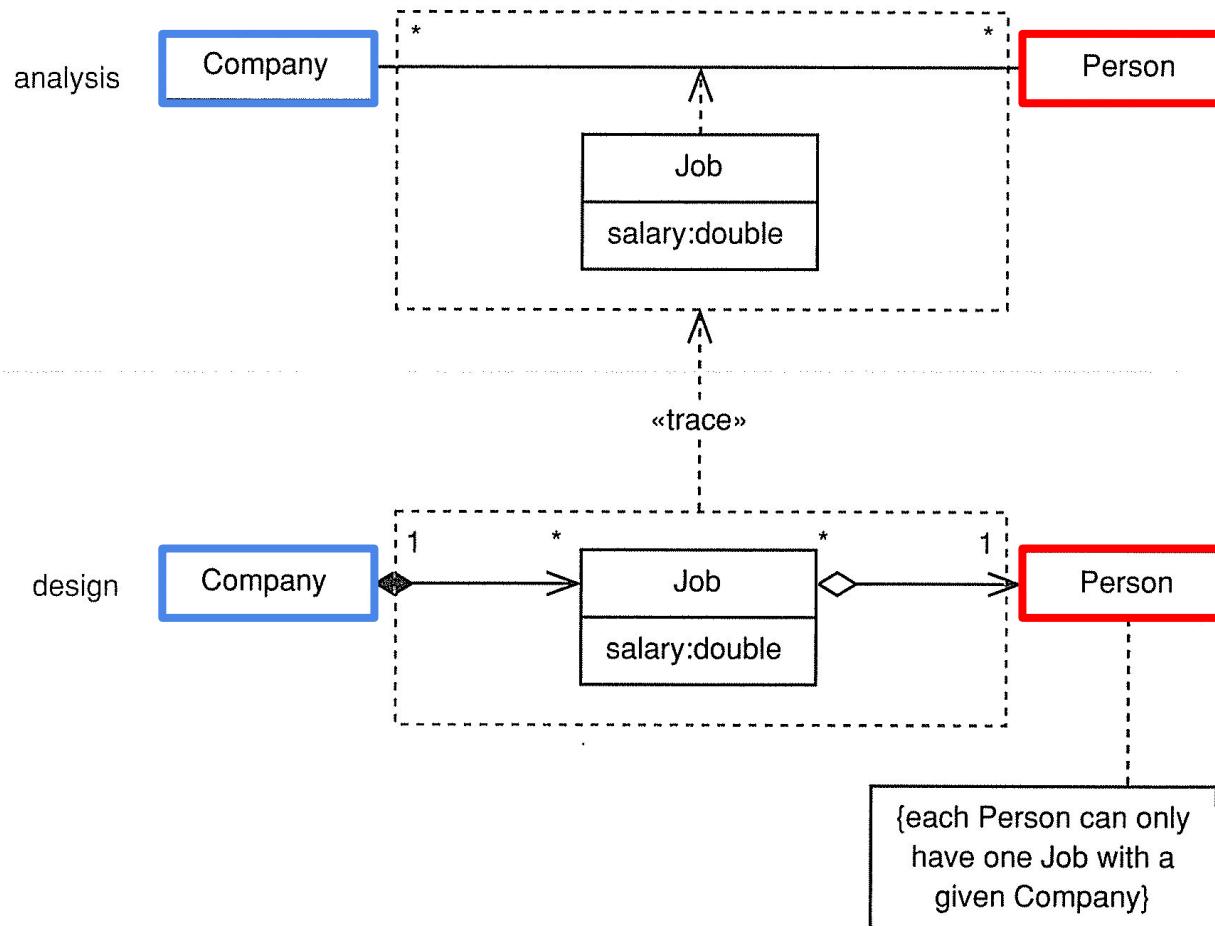
- Not supported directly in most OO languages
  - We need to reify the bidirectional analysis association into two unidirectional associations, or dependencies
  - Remember to keep the asymmetry relationship - an object may not be part of itself, in aggregation or composition



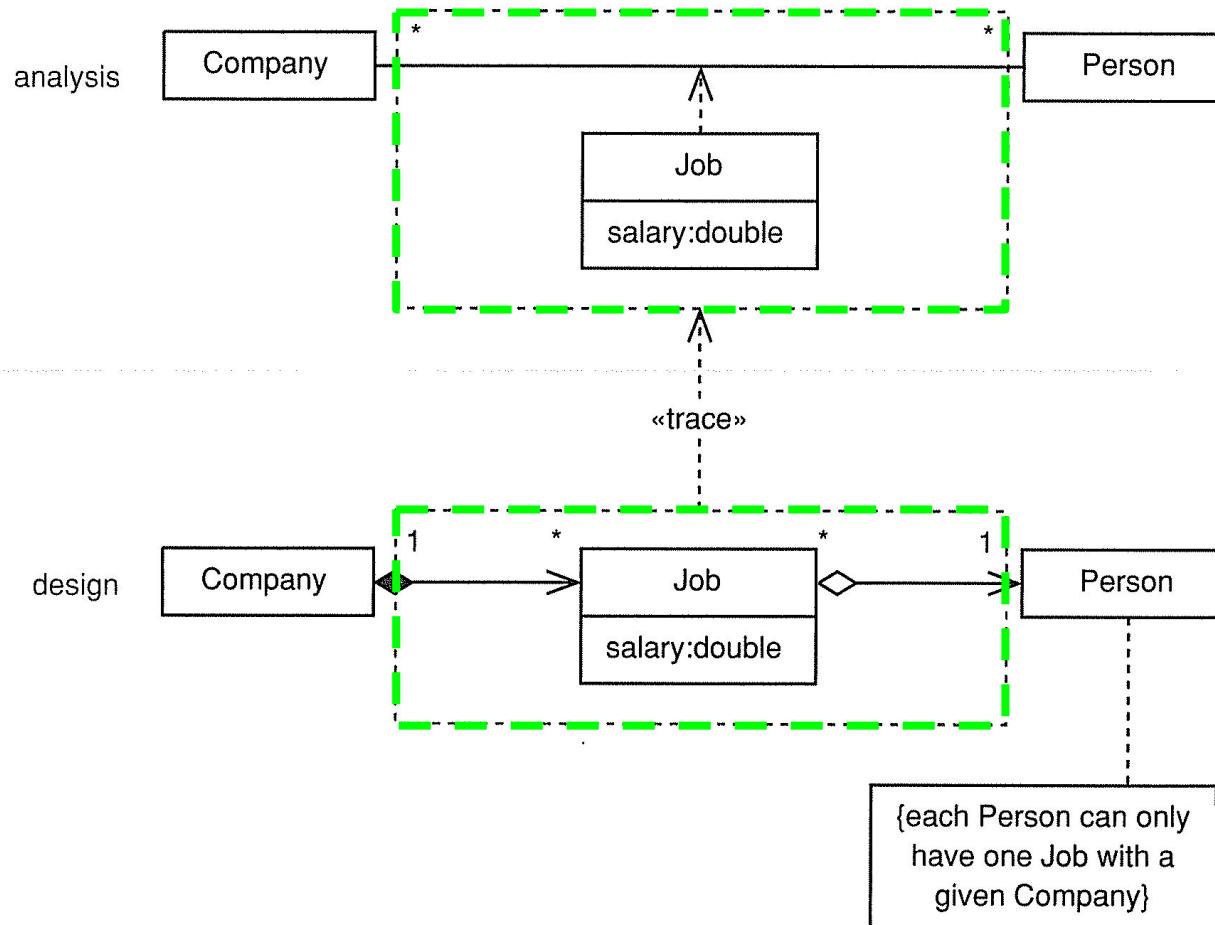
# Association classes are an analysis artifact - they are not supported by OO languages



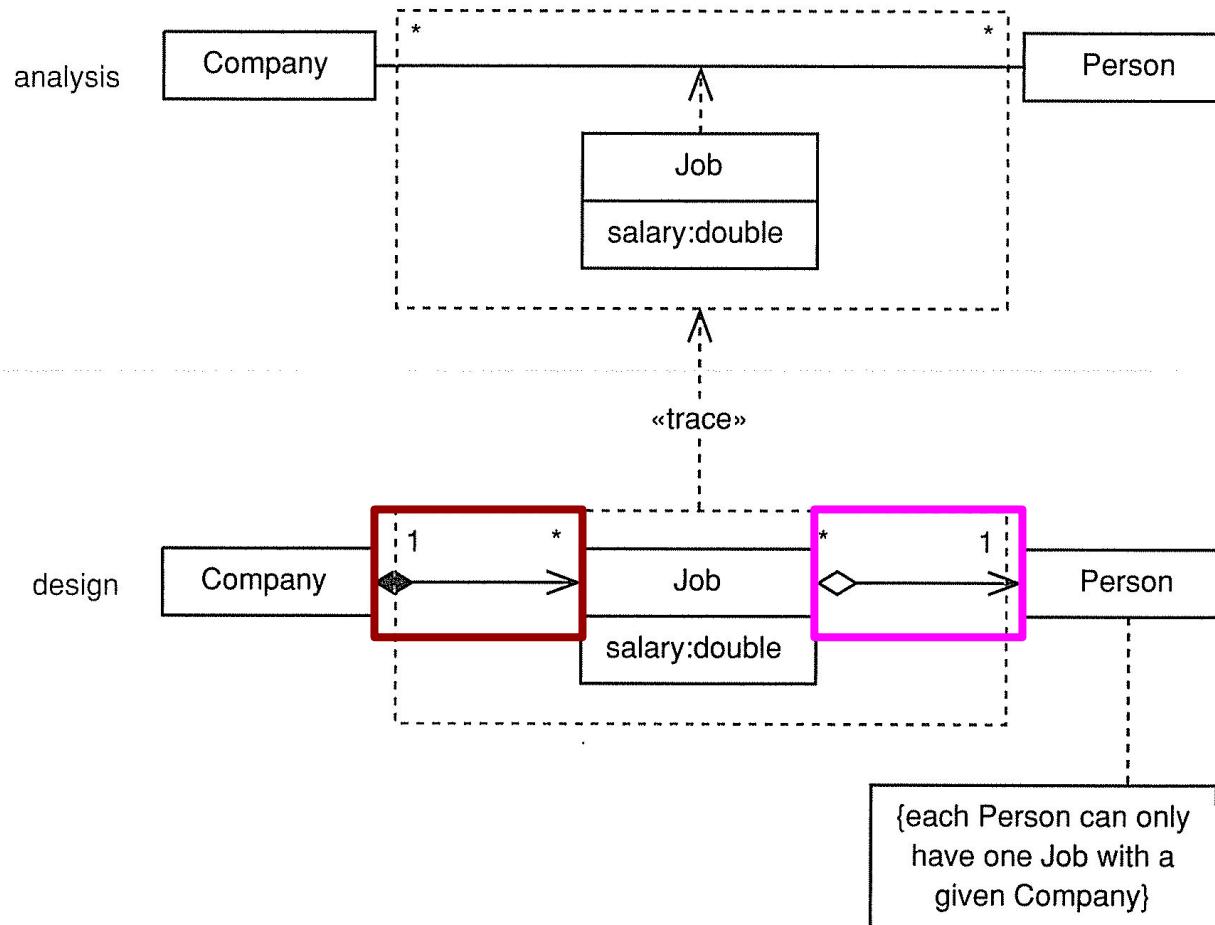
# You decide which side of the association plays the role of the **whole**, and which plays the **part**



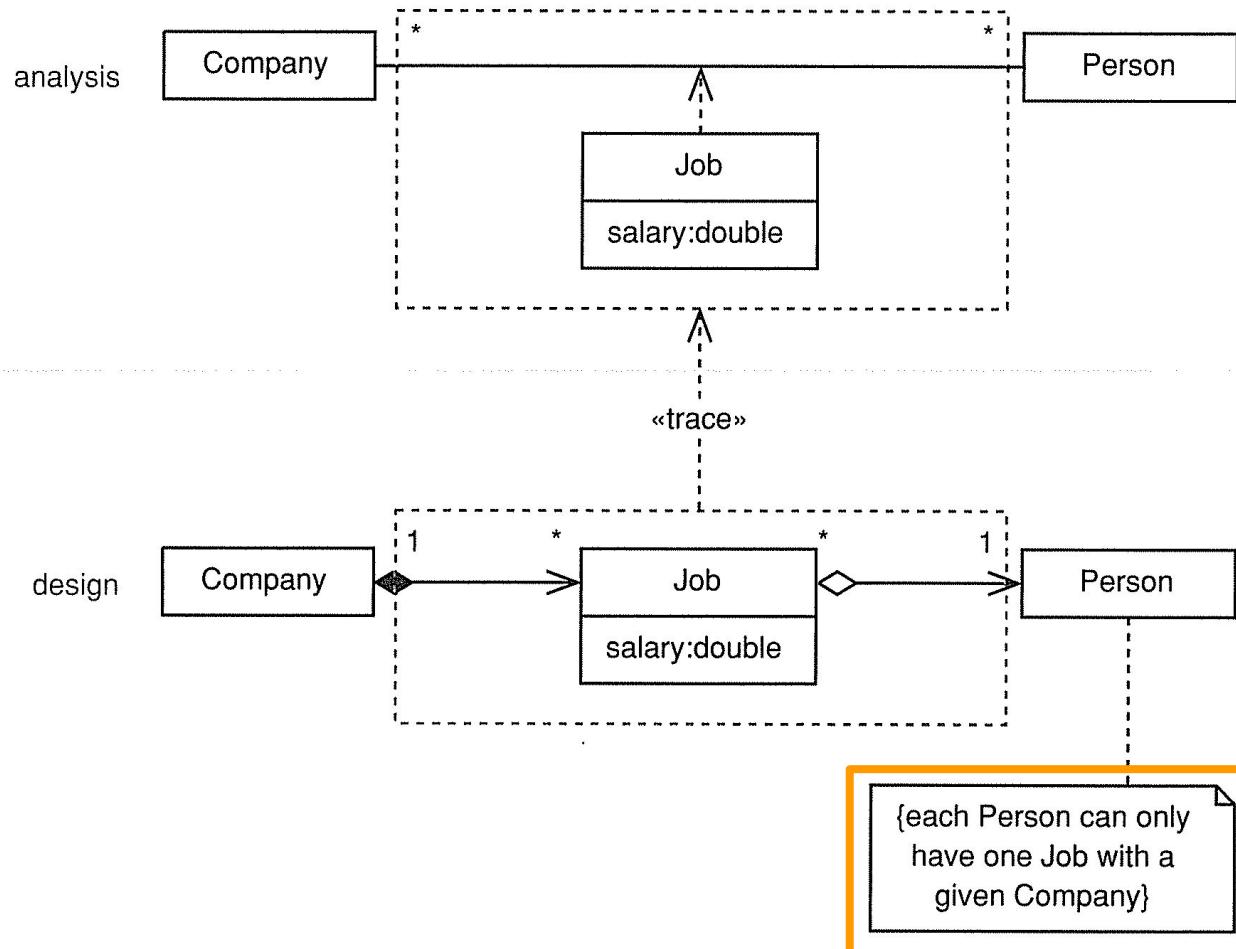
The **association class** is turned into a class that serves as a “middle man” between the associated classes



## Use a combination of **composition** and **aggregation** (or even dependency, to capture the association class semantics



# You can keep the semantics of a unique pair by adding a **note** with the appropriate constraint



---

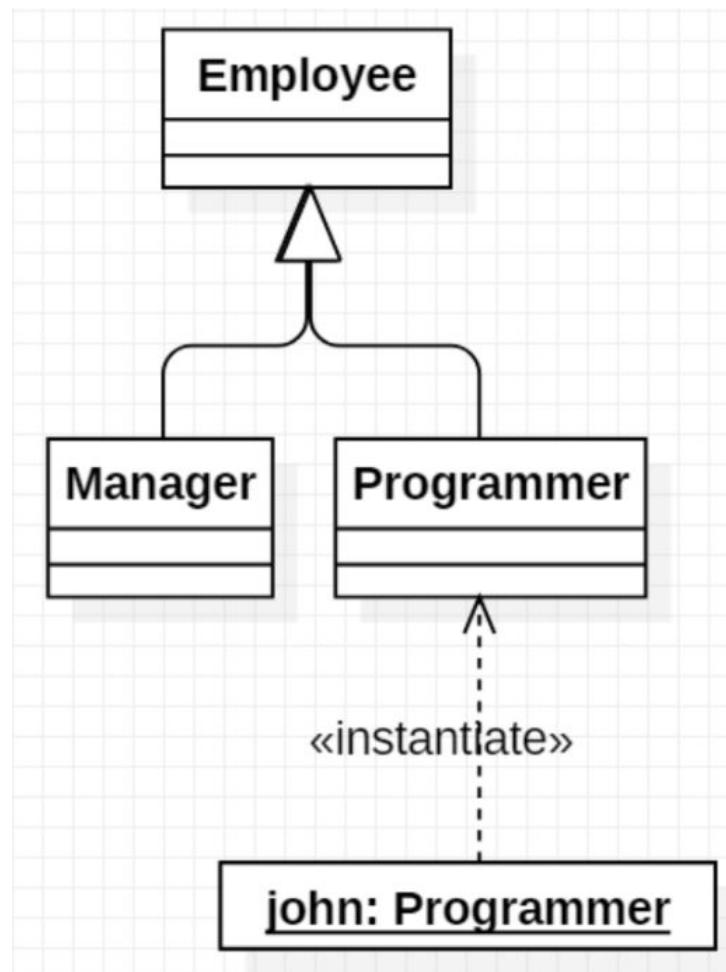
# Aggregation vs inheritance

# Inheritance is the strongest form of coupling between classes - it is an inflexible relationship

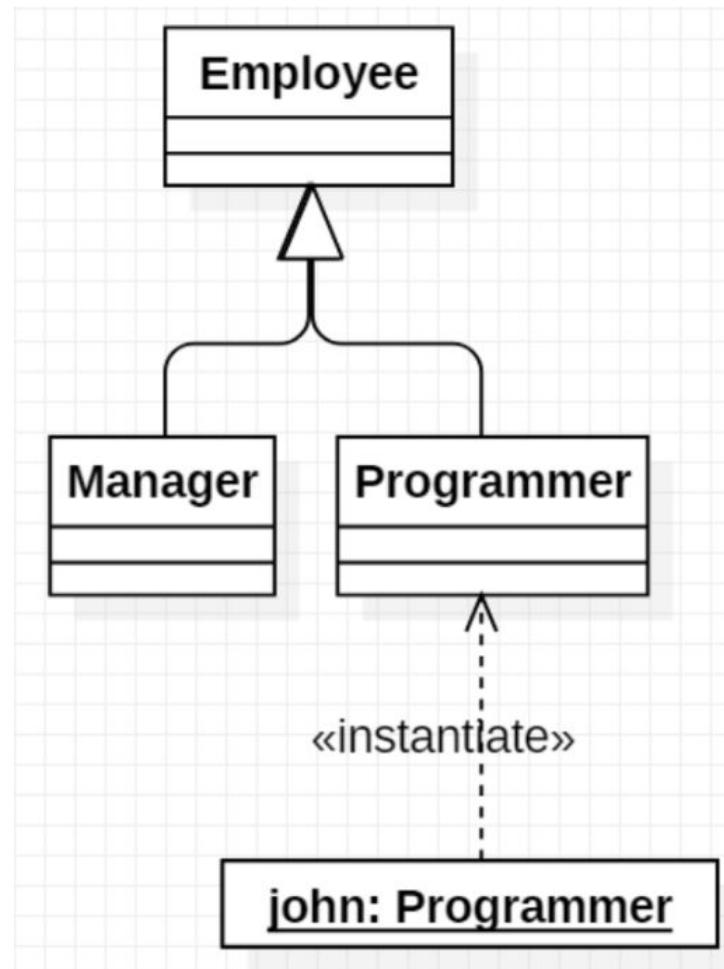
---

- Inheritance is the strongest form of coupling between classes
- Encapsulation is weak within the class hierarchy
  - Changes in a base class impact strongly on its children
- Inheritance is inflexible
  - Inheritance relationships are fixed at compile time
  - In contrast, aggregations and compositions can evolve at runtime

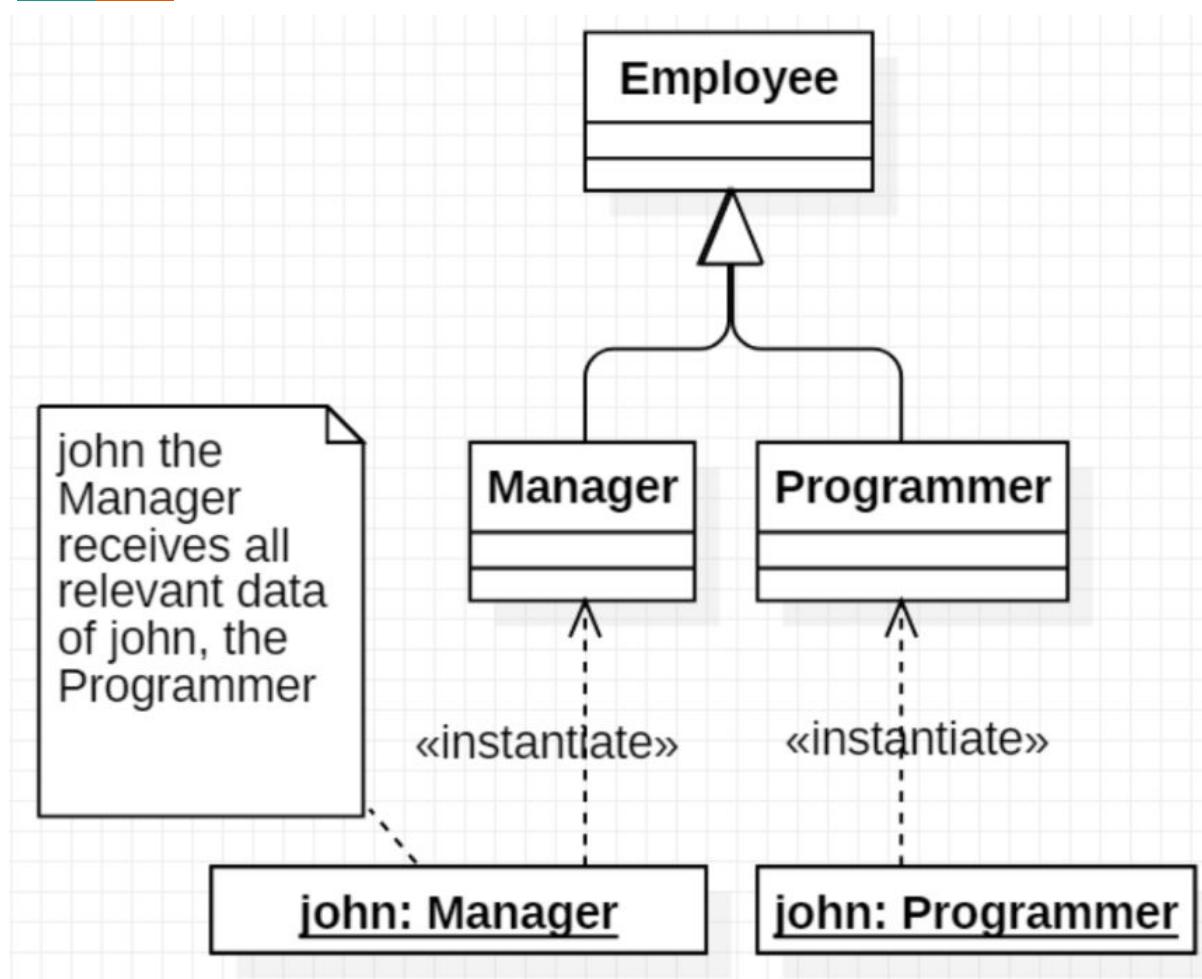
# What is wrong with this diagram?



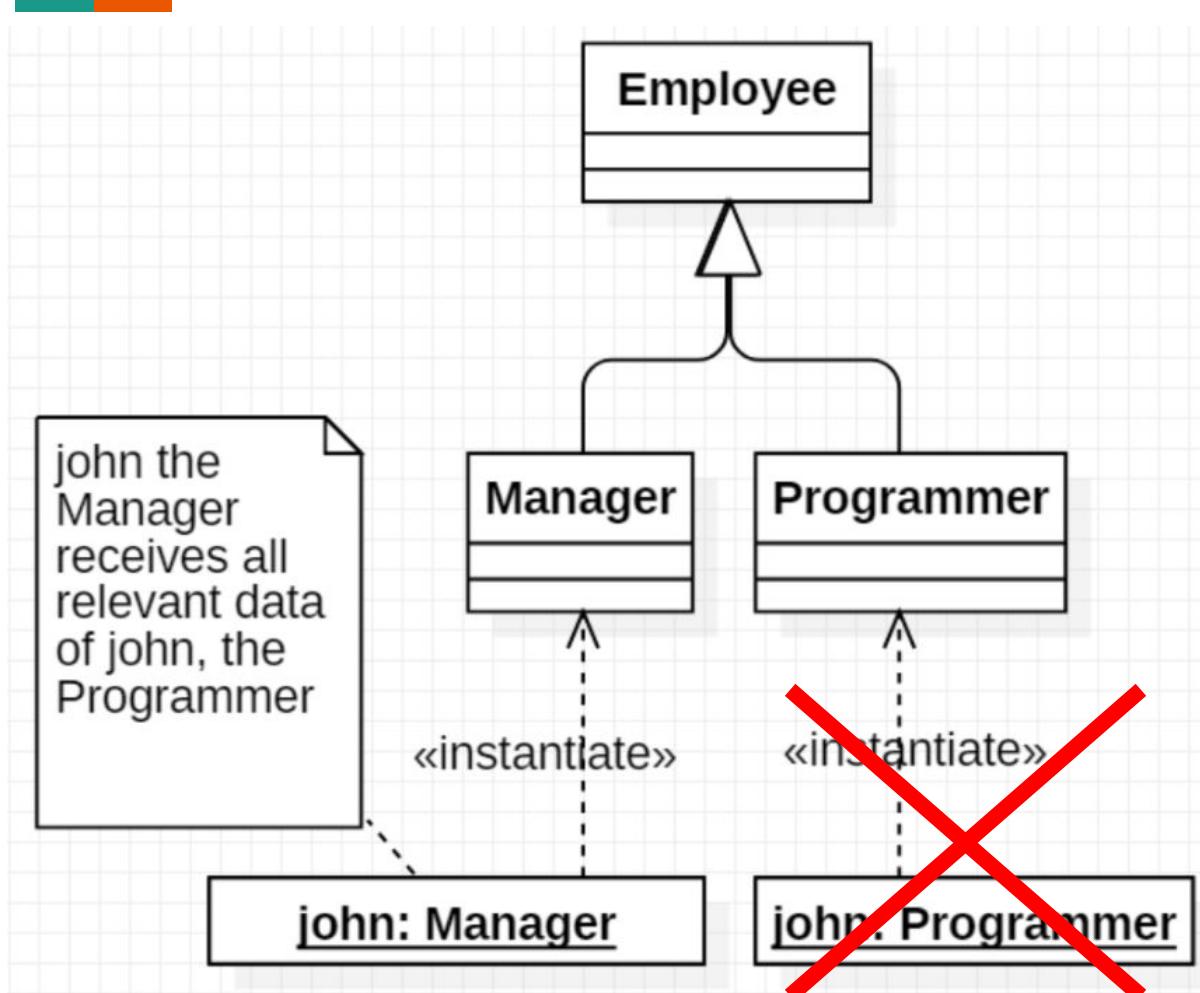
# What if you want to promote John to become a manager? You can't change its class at runtime!



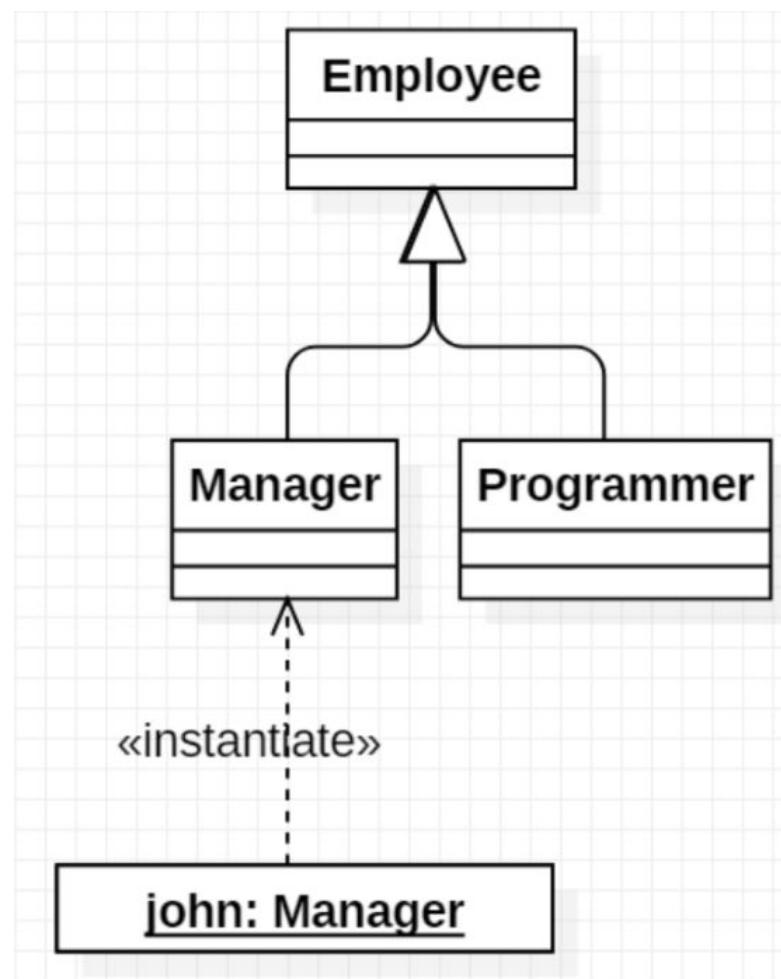
You can create a new Manager for representing John, copy all the relevant data from John the Programmer to John the Manager, and add John the Manager to any relevant collection



# Finally, **delete** John, the programmer

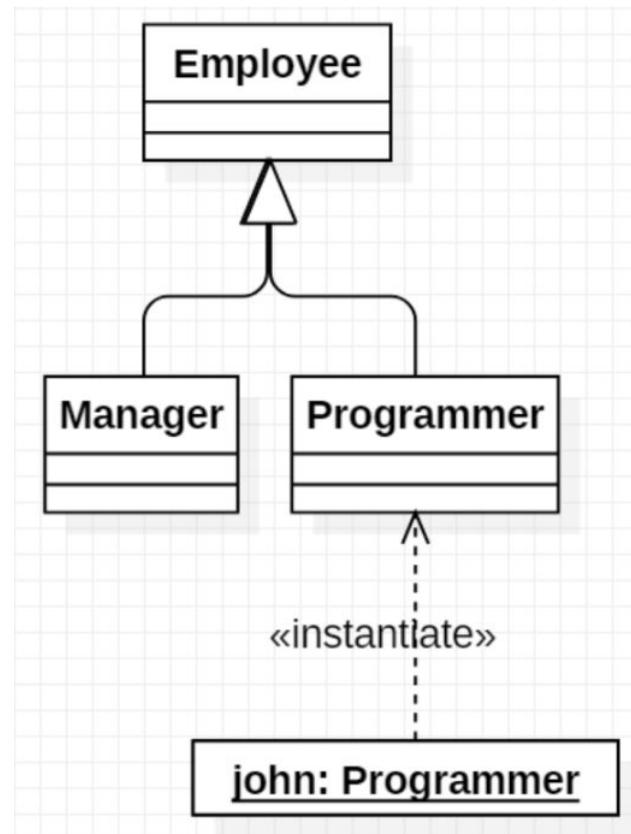


This process was cumbersome, because we used the wrong abstraction

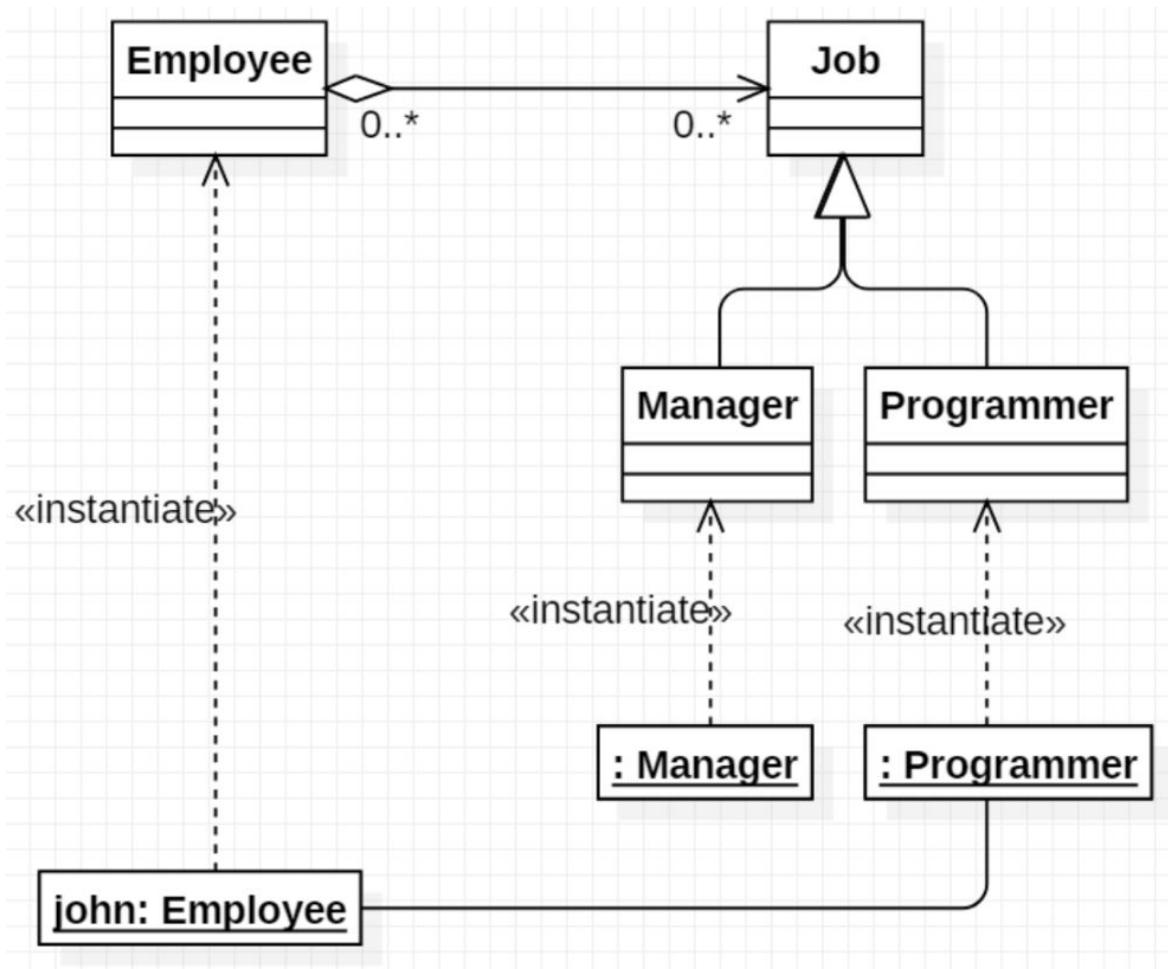


# Is an employee *just* his job, or is it rather that an employee *has* a job?

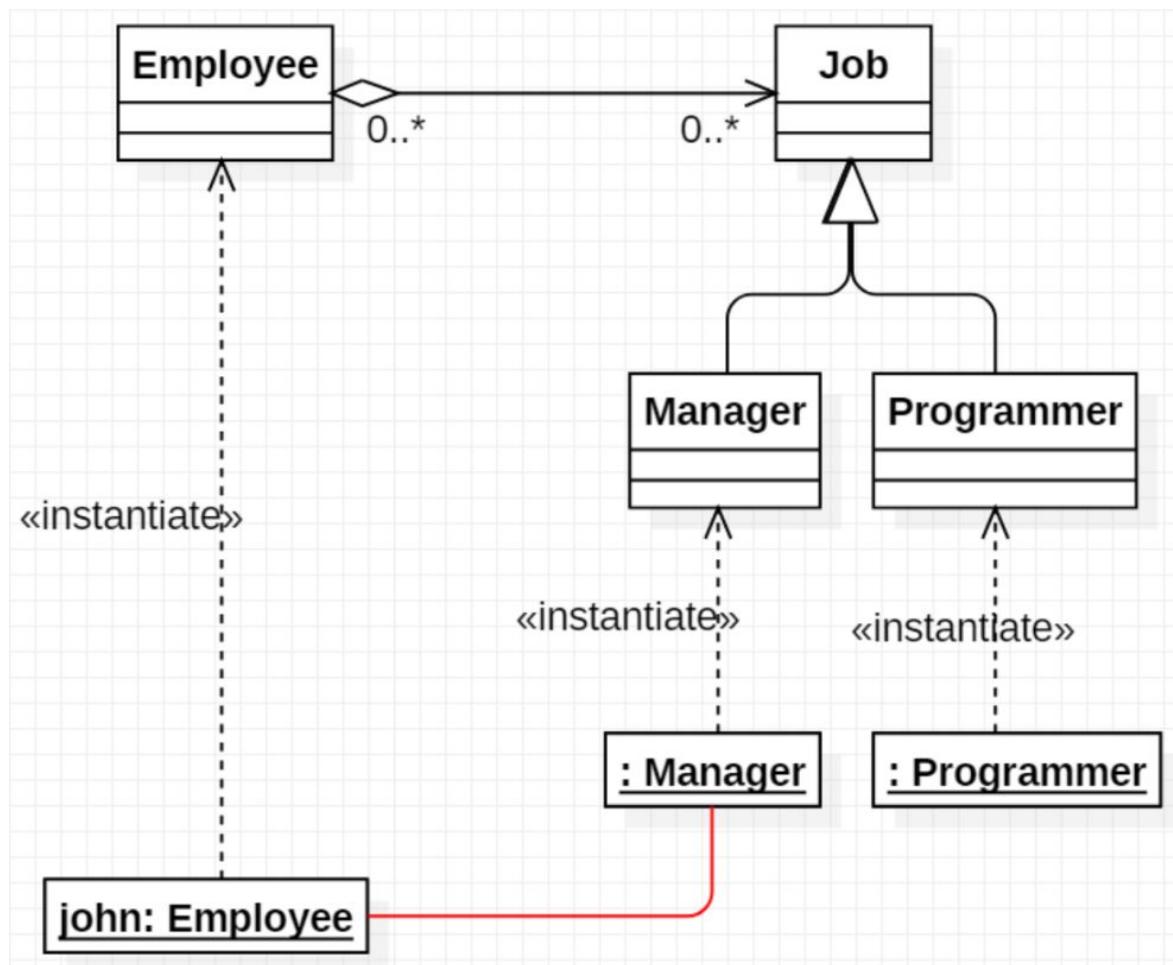
- This is a common modelling mistake. Subclasses should always represent **a special kind of** rather than **a role played by**.
- A job is a role played by an employee and does not really indicate the kind of employee.
- That said, maybe there are several kinds of jobs in a company and these would be better candidates for an inheritance hierarchy.



# Using aggregation, you can get the correct semantics and make your model more flexible!



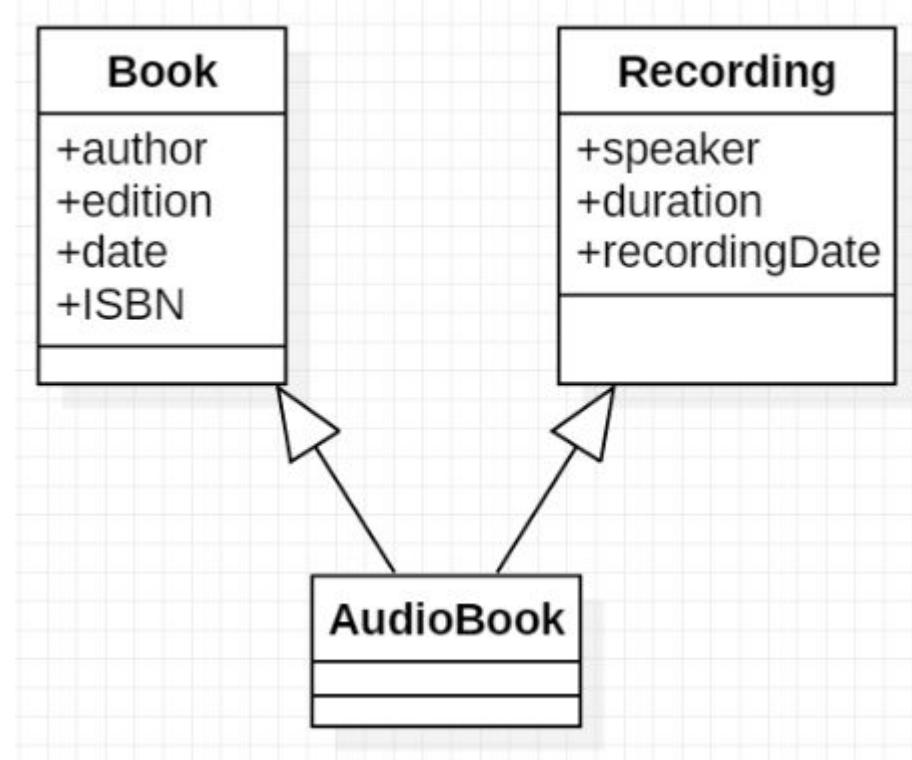
Now, you can **promote John to Manager** with a simple assignment



---

# Multiple inheritance

# With **multiple inheritance**, a class may inherit features from several super classes



# Multiple inheritance allows a class to have more than one parent

---

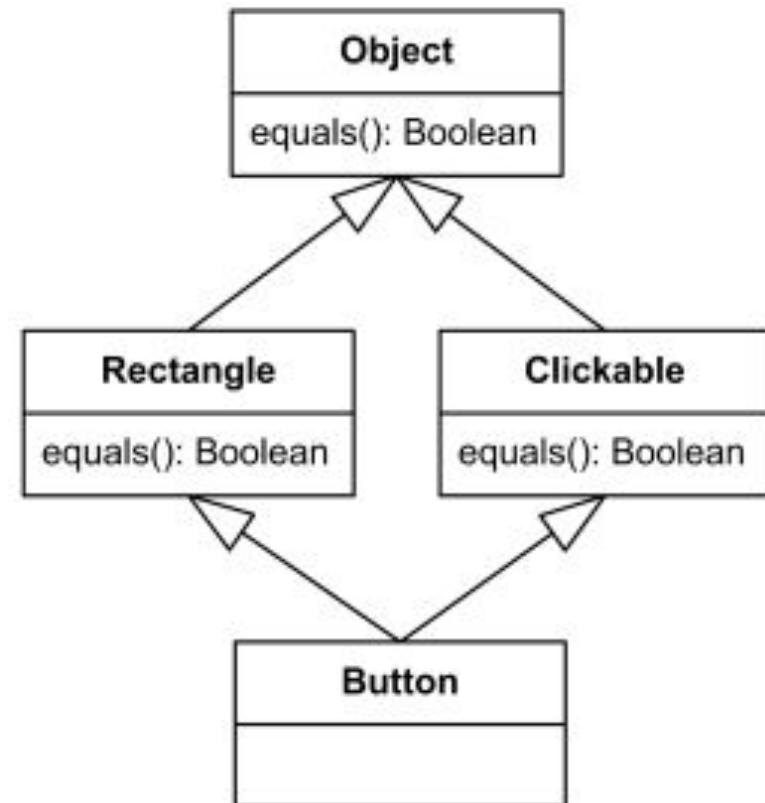
- This is not supported by all programming languages (e.g. Java and C# do not support it)
- All parent classes must be semantically disjoint (i.e. orthogonal), to avoid unforeseen interactions among them
- The “is kind of” and substitutability principles must apply between a subclass and all its superclasses
- The superclasses should have no parent in common, to avoid a well-known anti-pattern - the multiple inheritance diamond



**Diamonds are not a girl's best friend.  
Not if she really is a class model...**

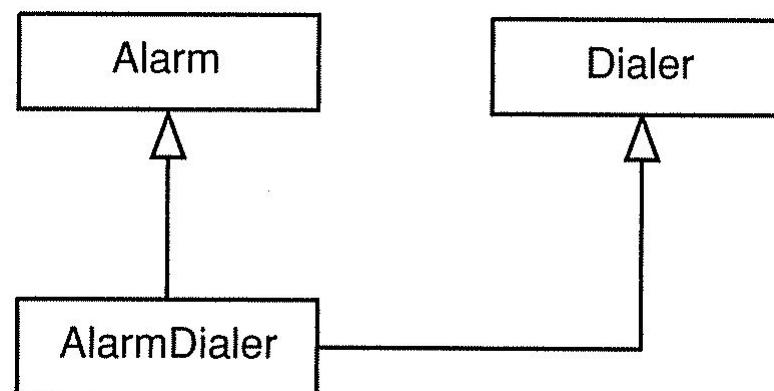
# Which implementation of equals() should Button use?

- Languages supporting multiple inheritance, such as C++, have language-specific solutions to mitigate this problem
- Other languages, such as Java, or C#, do not support multiple inheritance, to prevent this kind of problem



# How to properly use multiple inheritance? With a Mixin!

- Mixin classes are designed to be “mixed in” using multiple inheritance
- Mixins provide a safe and powerful idiom
- In this example, **Dialer** is a mixin. All it does is to dial a phone number. Not too useful on its own, but it provides a widely used service in a cohesive package, being therefore a good candidate for reuse.



---

# Inheritance vs interface realization

# Inheritance vs interface realization

---

- With inheritance, the inheriting class gets
  - interface - the public operations of the base classes
  - implementation - the attributes, relationships, protected and private operations of the base classes
- With interface realization, the implementing class gets
  - interface - a set of public operations, attributes and relationships with no implementation
- Both inheritance and interface realization define a contract that subclasses and implementing classes must implement
- Interface realization is useful when you want to define a contract but are not concerned about the implementation - this is more flexible and robust than inheritance

---

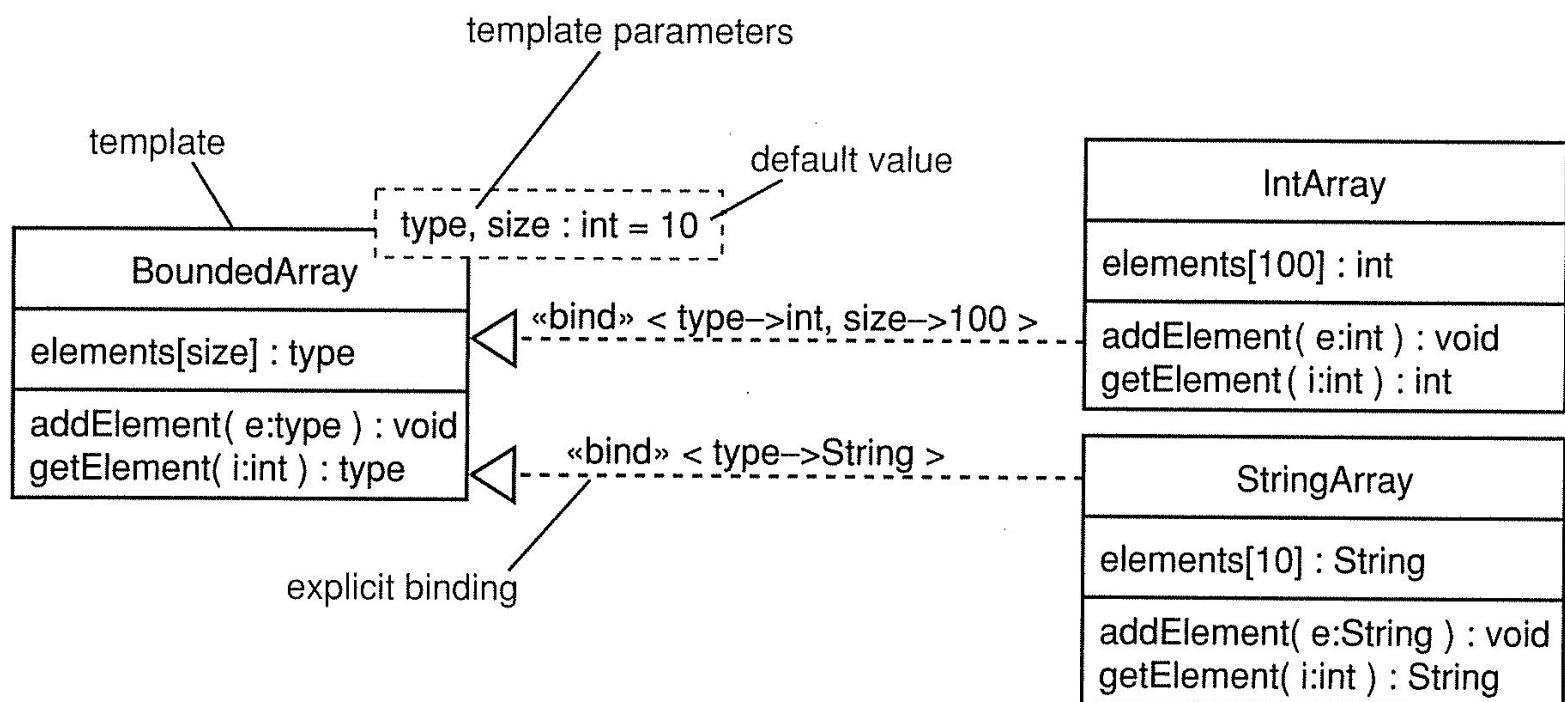
# Templates

# We can use templates to parameterize a class

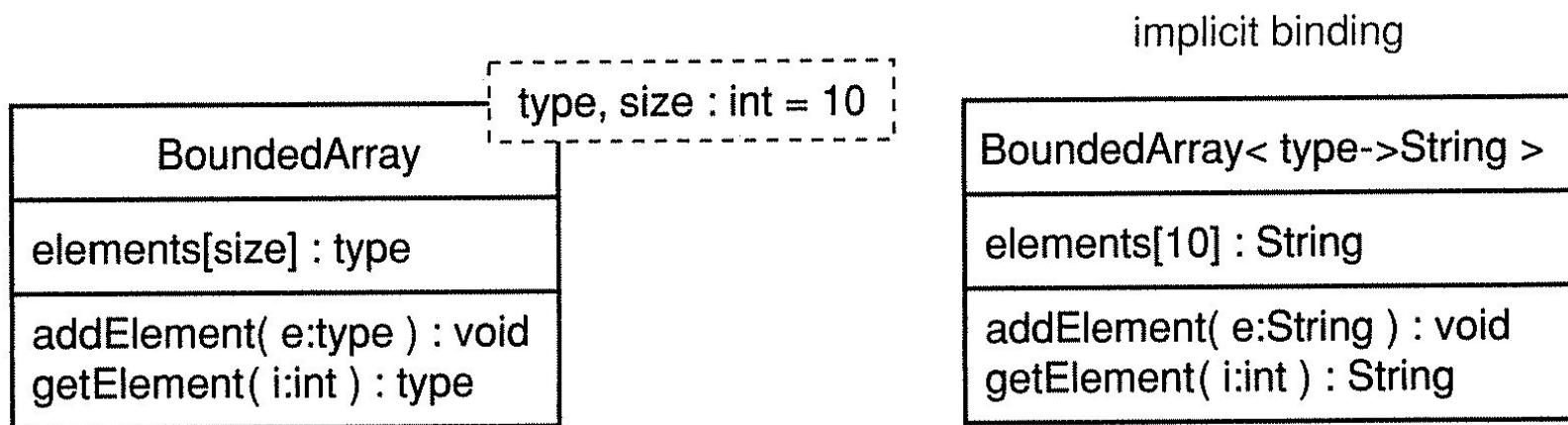
- Instead of defining the actual types of attributes, operation return values and operation parameters, we can define those as placeholders, or parameters
- These placeholders are then replaced by actual types to create new classes

BoundedIntArray	BoundedDoubleArray	BoundedStringArray
size : int elements[ ] : int	size : int elements[ ] : double	size : int elements[ ] : String
addElement( e:int ) : void getElement( i:int ) : int	addElement( e:double ) : void getElement( i:int ) : double	addElement( e:String ) : void getElement( i:int ) : String

In this example, **BoundedArray** is <<bind>> to specific instantiated classes: **IntArray** and **StringArray**



# Same example, BoundedArray is <<bind>> to specific instantiated class: in this case, a String

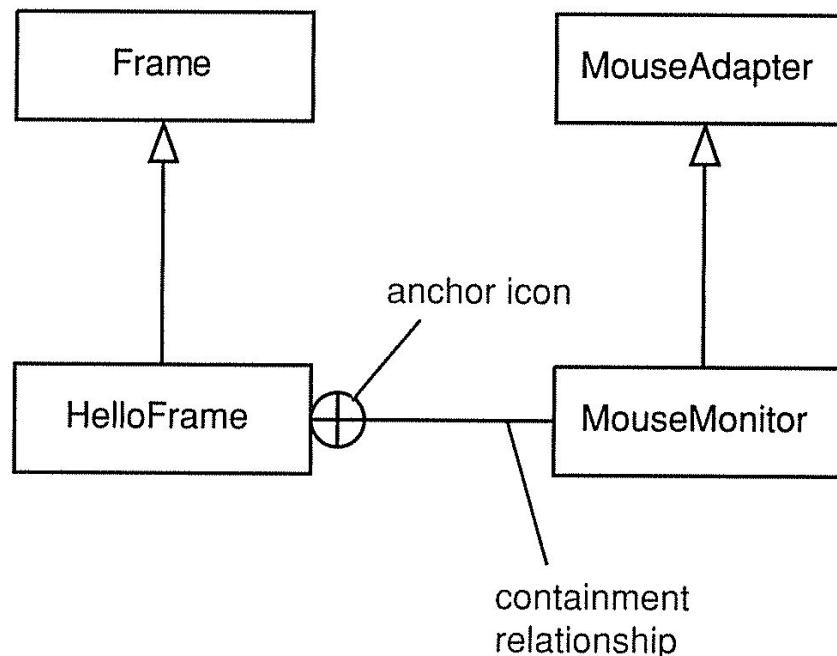


---

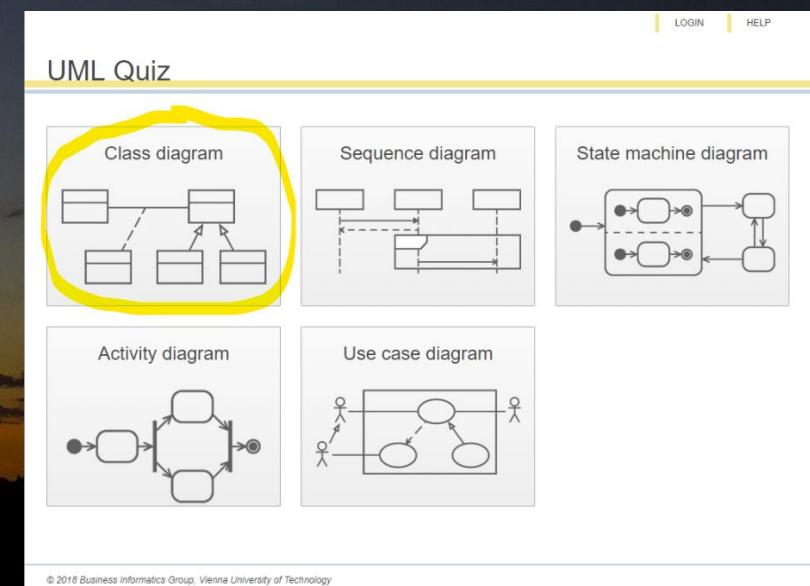
# Nested classes

# A nested class is a class inside a class.

- The nested class is only accessible by its outer class, or by objects contained by the outer class
- These classes are, in general, only used in design time
- In programming languages, such as Java, nested classes are often used for event handling



Did you really understand Design Class Diagrams? Test yourself at:  
<http://elearning.uml.ac.at/>



The screenshot shows a web-based quiz interface titled "UML Quiz". At the top right are "LOGIN" and "HELP" buttons. Below the title are five boxes, each containing a UML diagram type. The "Class diagram" box is highlighted with a yellow circle. The other four boxes are labeled "Sequence diagram", "State machine diagram", "Activity diagram", and "Use case diagram".

- Class diagram:** Shows two classes connected by a directed association. One class has two associations pointing to another class.
- Sequence diagram:** Shows three lifelines with various message exchanges between them.
- State machine diagram:** Shows a state transition graph with states and transitions.
- Activity diagram:** Shows a flow of activities with decision points and loops.
- Use case diagram:** Shows actors interacting with use cases.

At the bottom of the page, a small copyright notice reads: "© 2018 Business Informatics Group, Vienna University of Technology".

# Bibliography

---

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”,  
Second Edition, Addison-Wesley 2006

- Chapter 17

---

# Structured classifiers

# We can explore the relationship of a composite classifier with its internal parts

---

- This may help to
  - Design a class
  - Design a use case
  - Design a subsystem
- Supports focusing on the internal works of the corresponding classifier

# A structured classifier is a classifier that has an internal structure

---

- The structure is modelled as parts that are joined by connectors
- The interaction of a structured classifier is modelled by its interfaces and ports

# A part is the role that one or more instances of a classifier may play in the context of the structured classifier

---

A part contains:

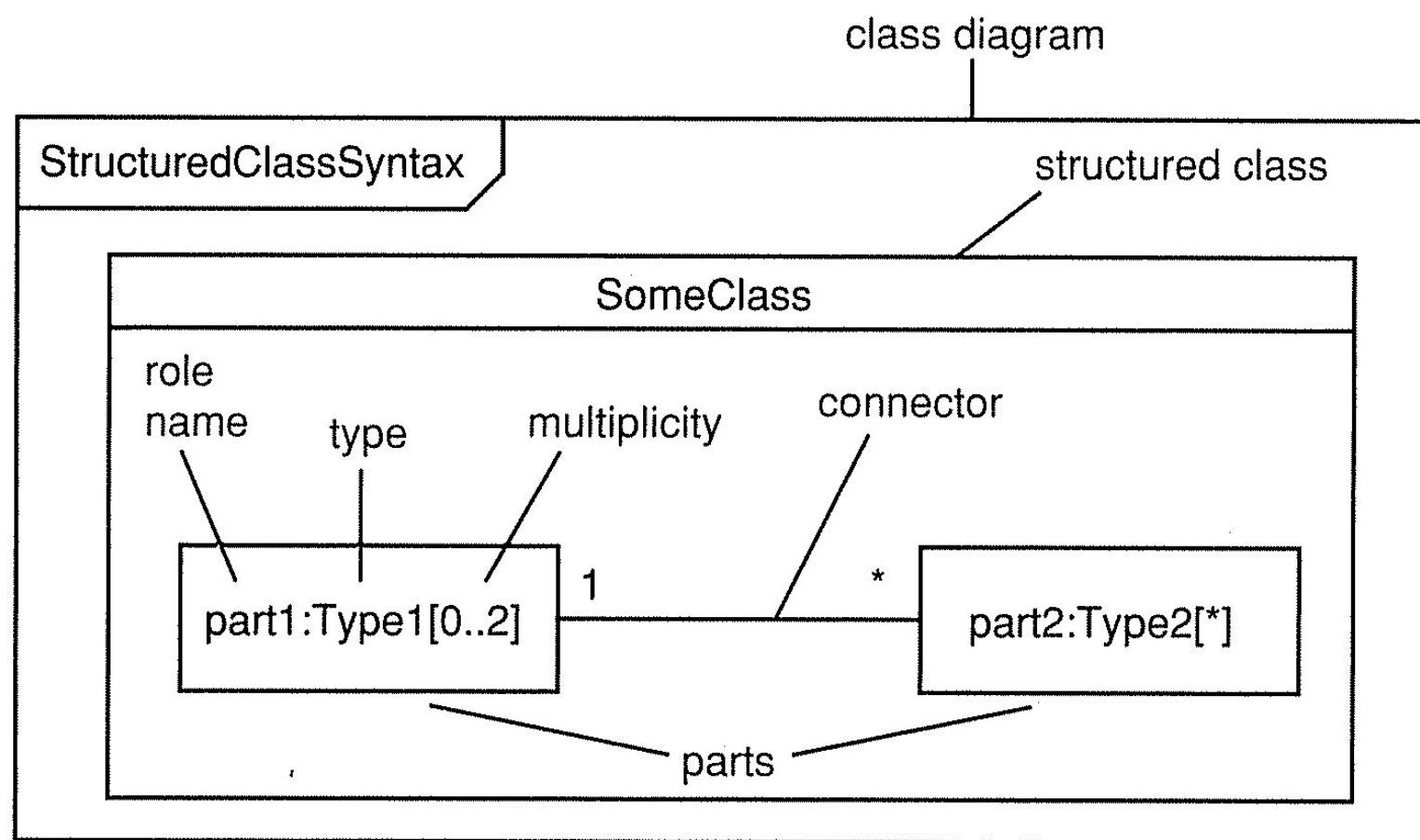
- Role name
  - Descriptive name for the role that instances play in the context of a structured classifier
- Type
  - Only instances of this type (or a subtype of this type) can play the role
- Multiplicity
  - The number of instances that can play a particular role at any given moment

# Connectors represent relationships between parts

---

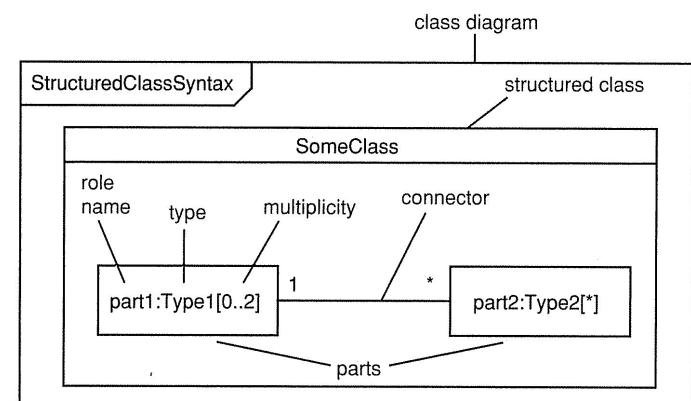
- A connector indicates that parts can communicate with each other
- There is a relationship between the instances playing those parts over which communication can occur
- Relationships may map to associations between the classes, or even some ad hoc relationship for temporary collaboration to perform some task
- Connectors and parts only exist within the context of a particular classifier

# Structured classifier syntax

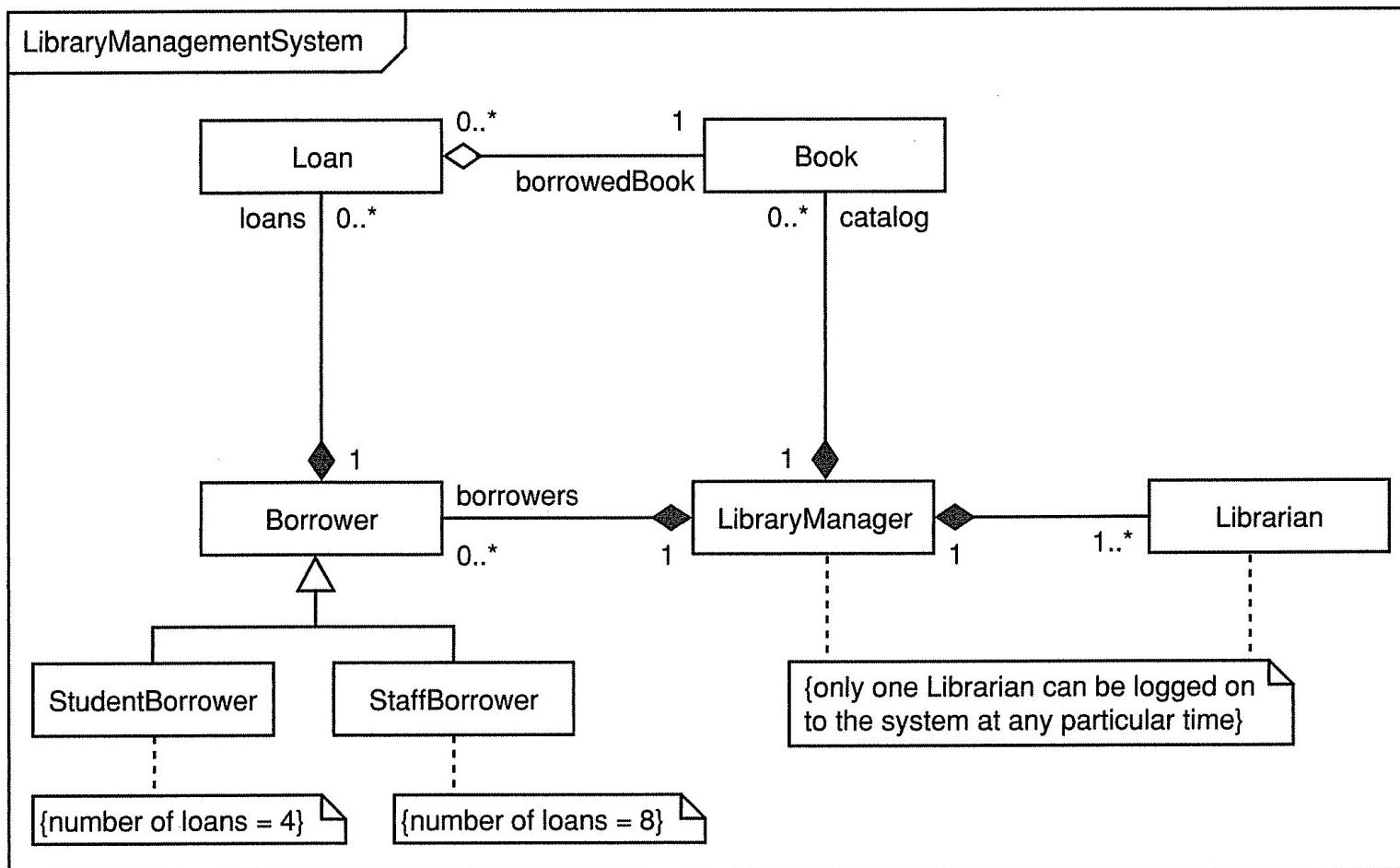


# Structured classifier syntax

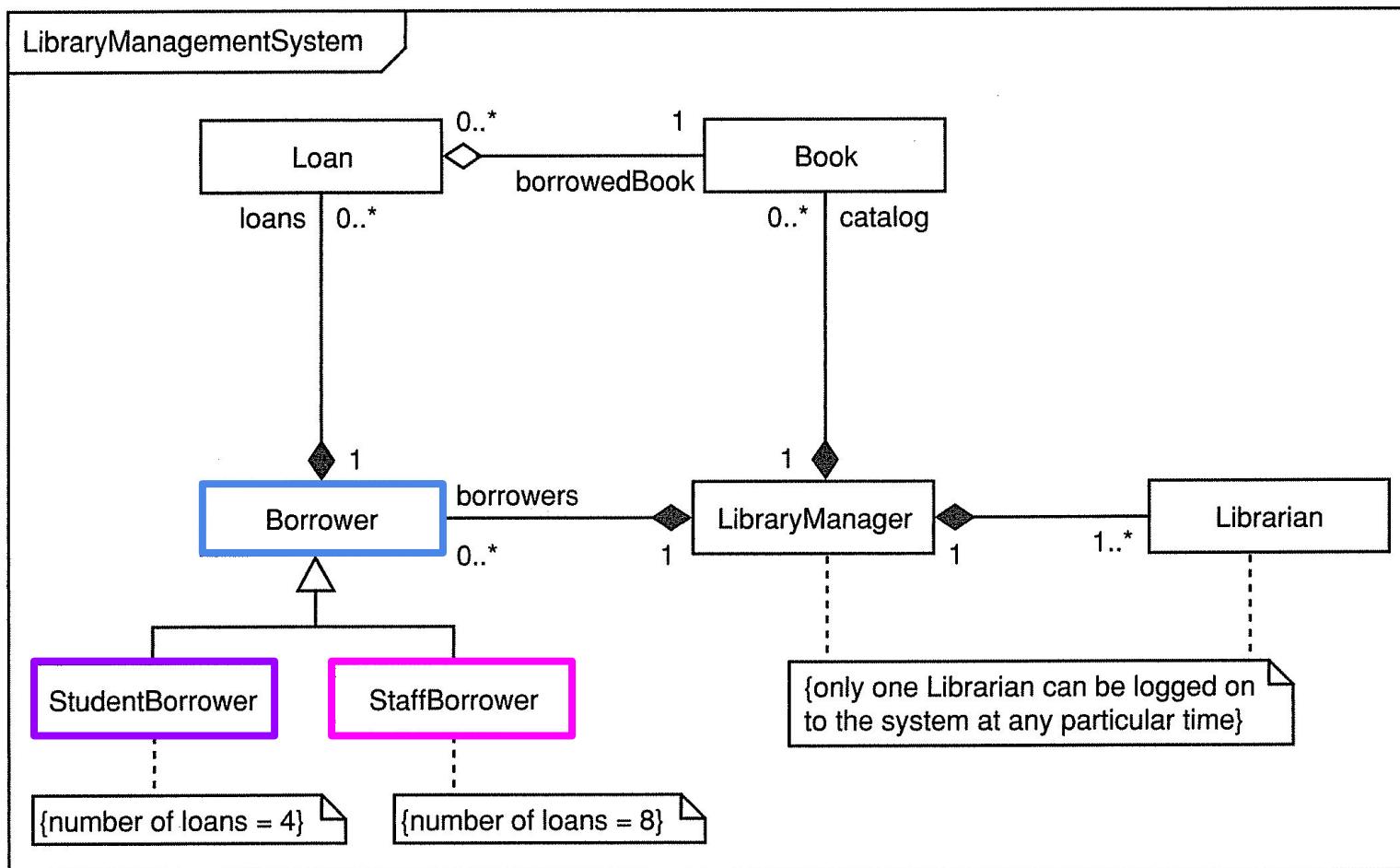
- The parts collaborate in the context of the structured classifier
- The parts represent roles that instances of a classifier can play in the context of the structured classifier
  - The parts do not represent classes
- The connector is a relationship between two parts indicating that the instances playing roles specified by the parts can communicate in some way
- Focus on internal implementation and external interface



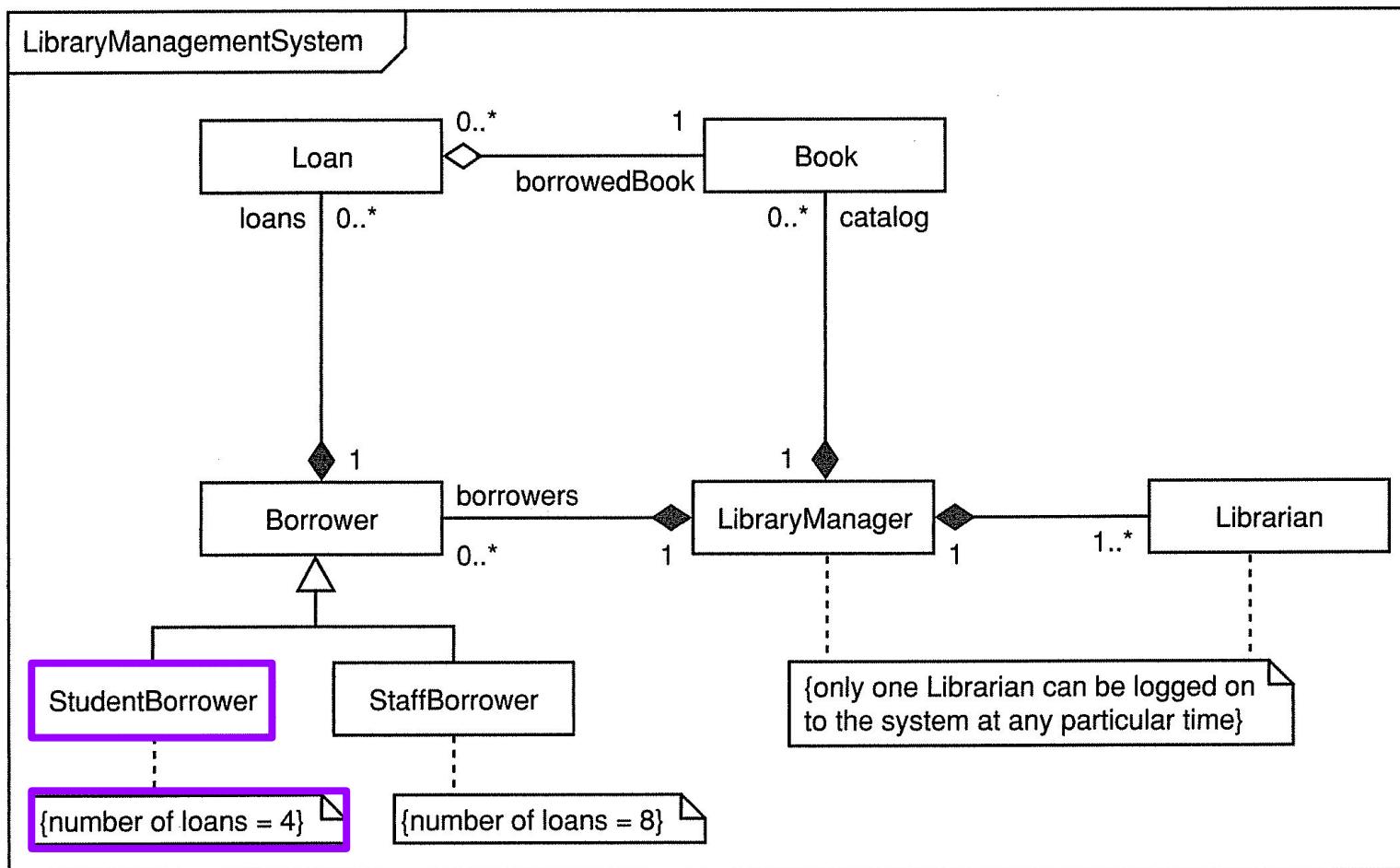
# Structured class example - a library management system



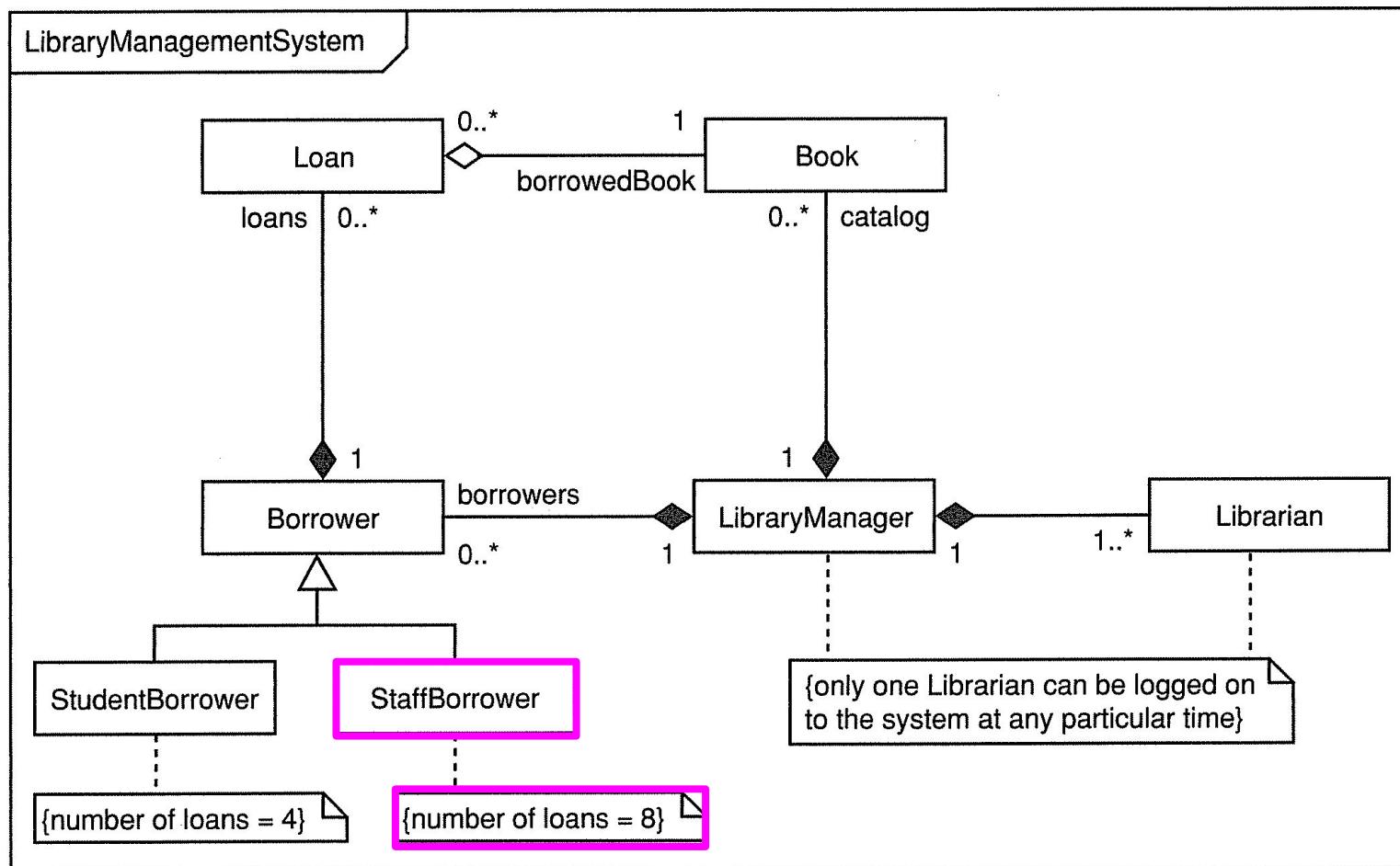
# There are two types of borrowers: students and staff



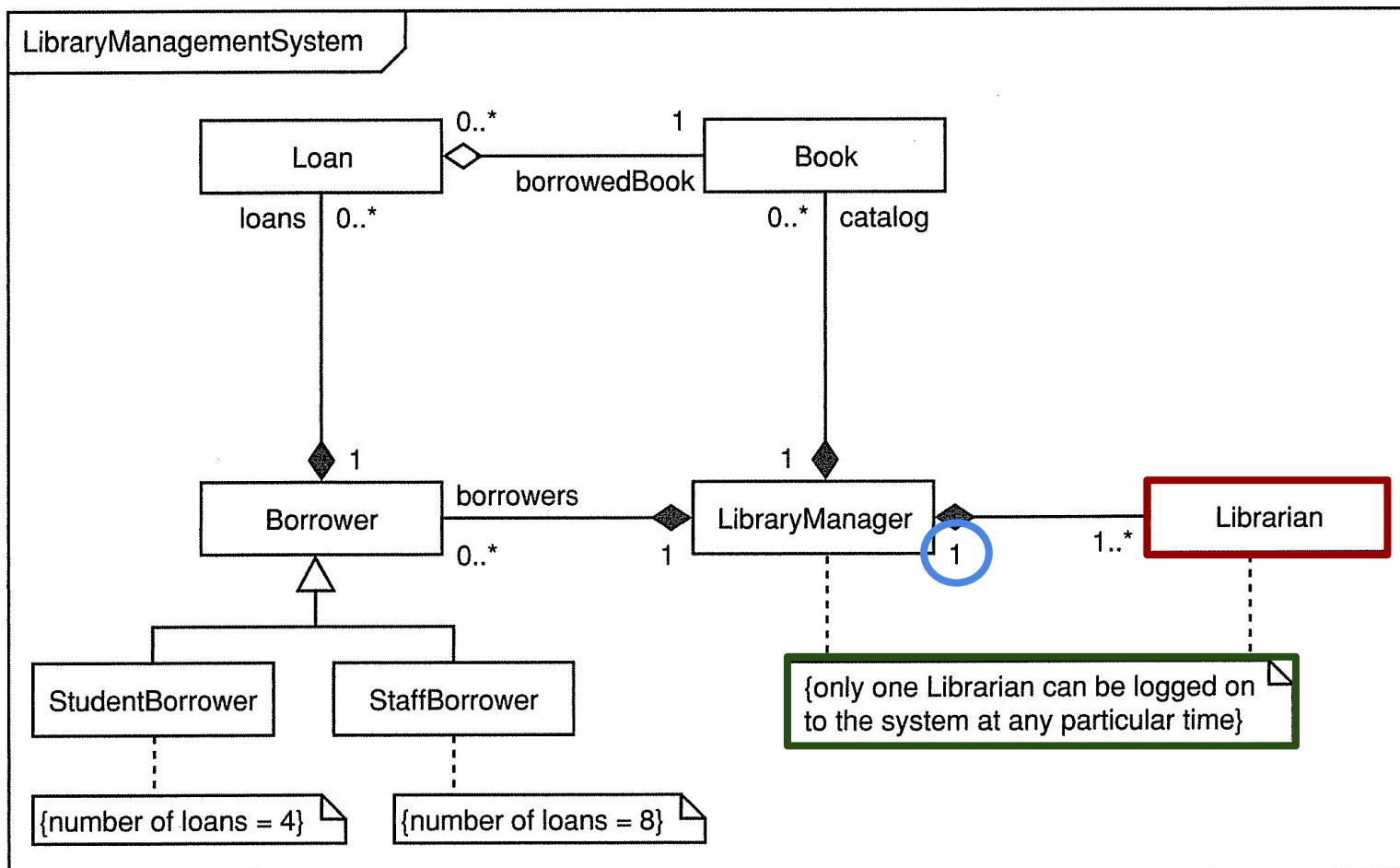
# Students can borrow up to 4 books



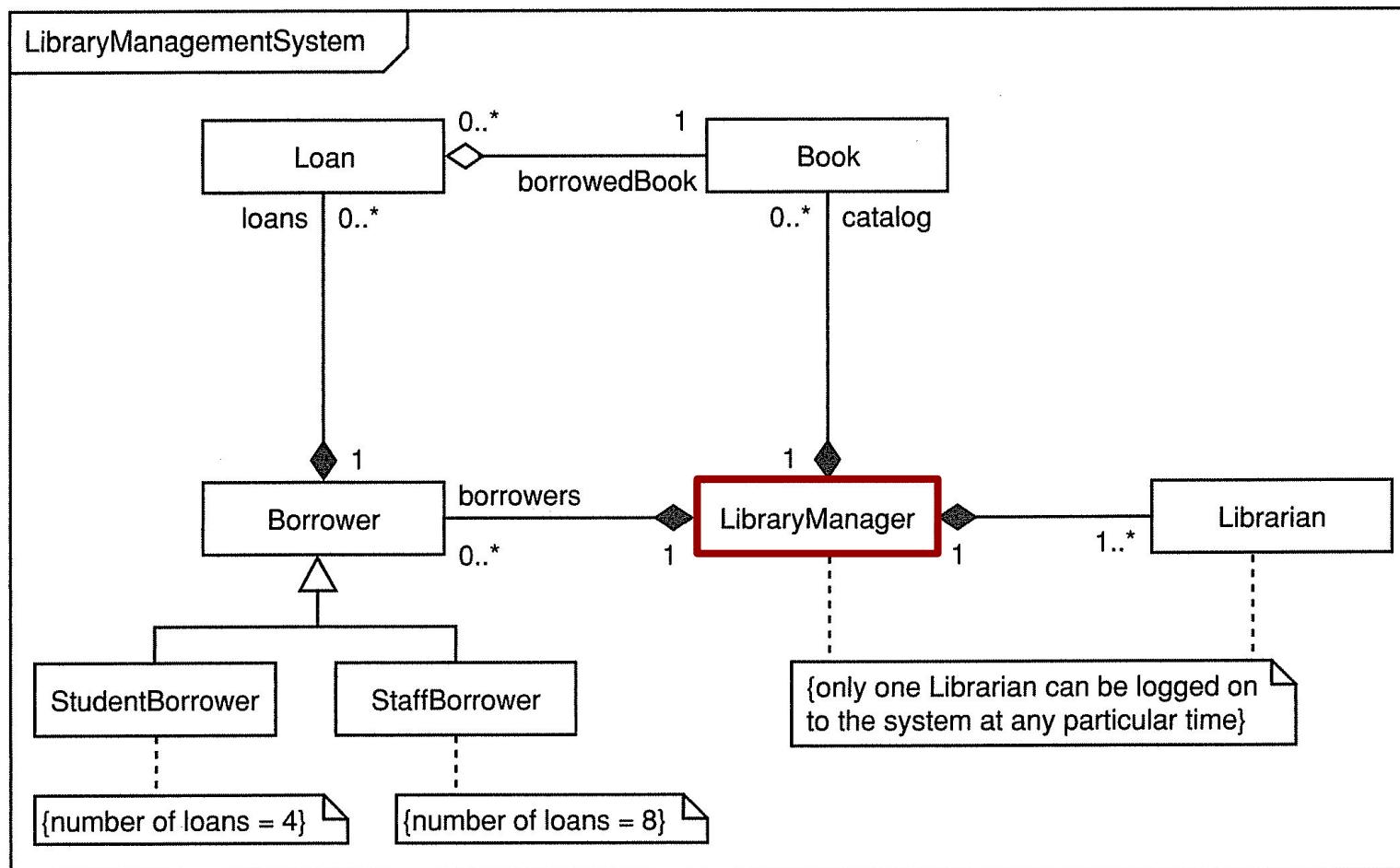
# Staff can borrow up to 8 books



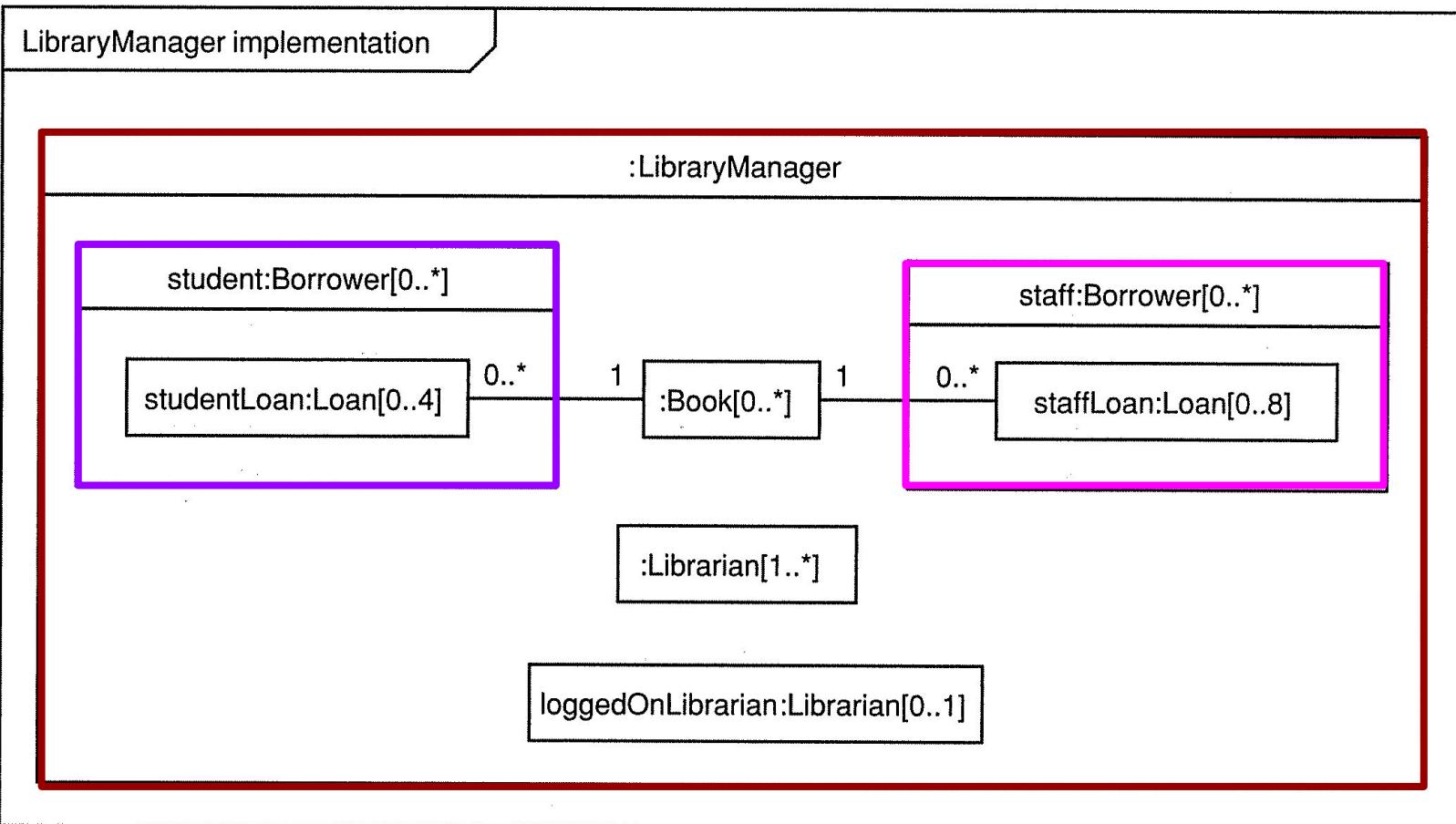
# Only one librarian can be logged onto the system at any particular time



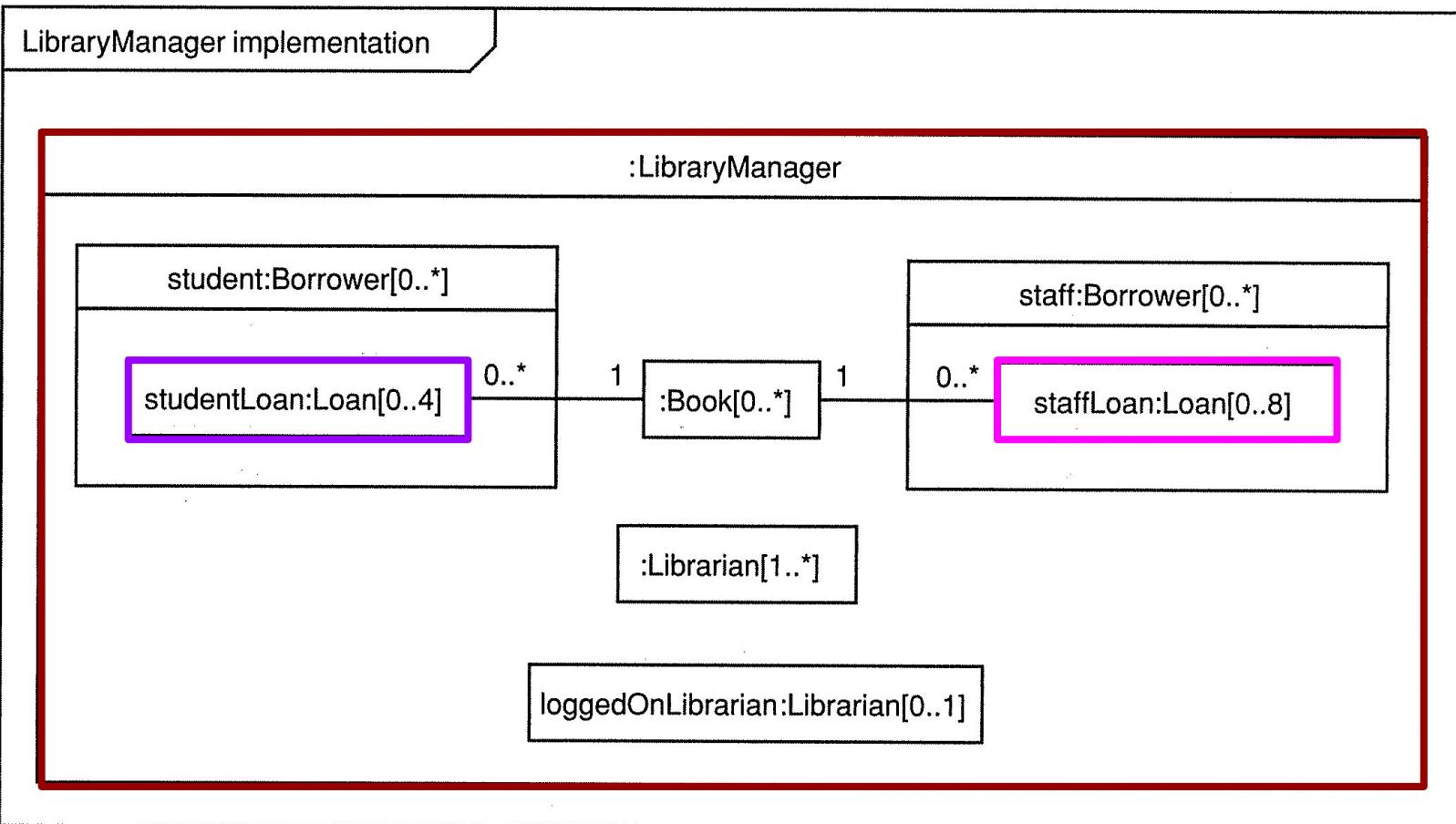
# Now, let us zoom into the **LibraryManager** and have a look at its implementation



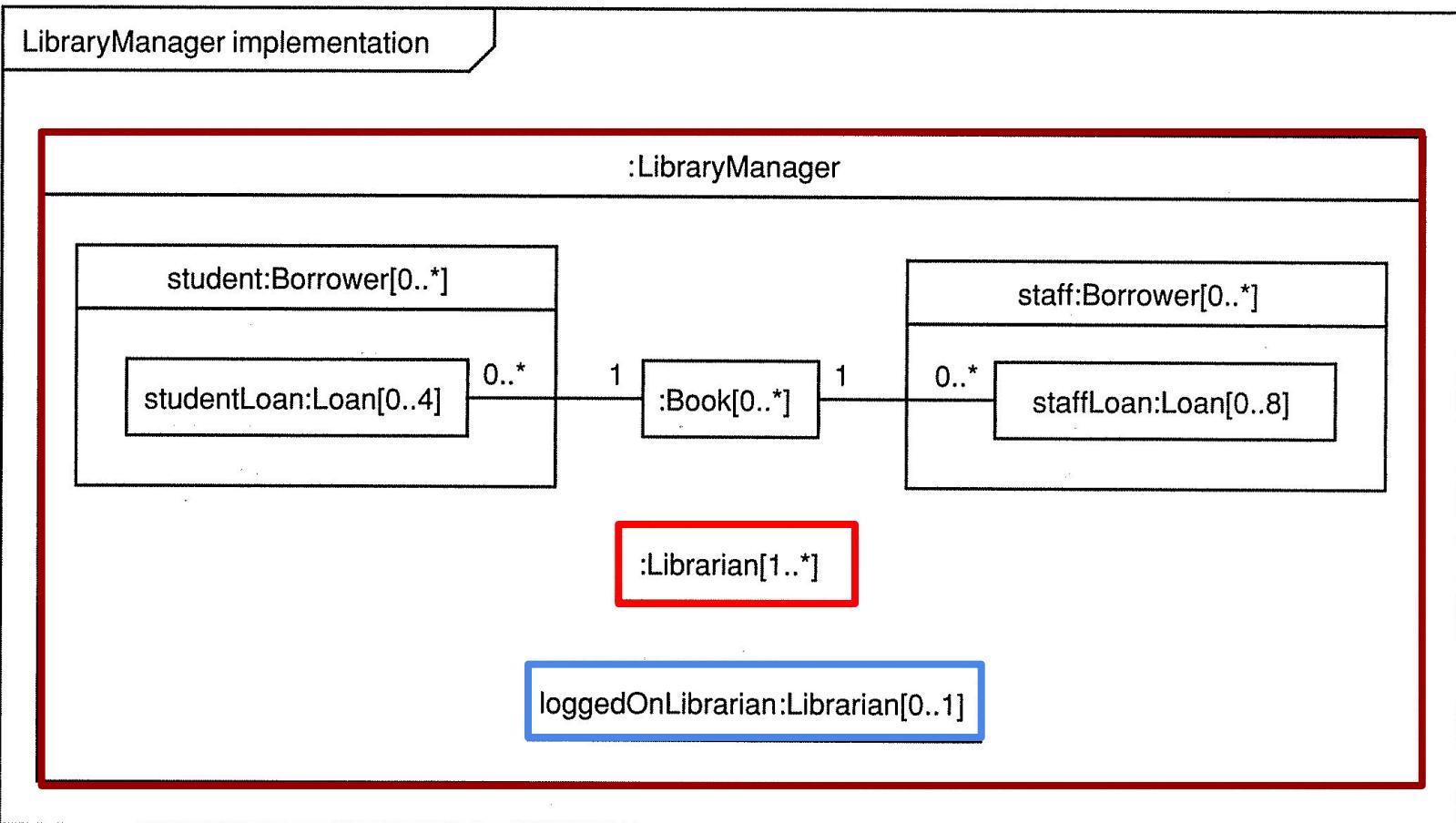
From the perspective of the **LibraryManager**, there are two types of borrowers: **students** and **staff**



**Students can borrow up to 4 books at a time (studentLoan).  
Staff can borrow up to 8 books at a time (staffLoan).**

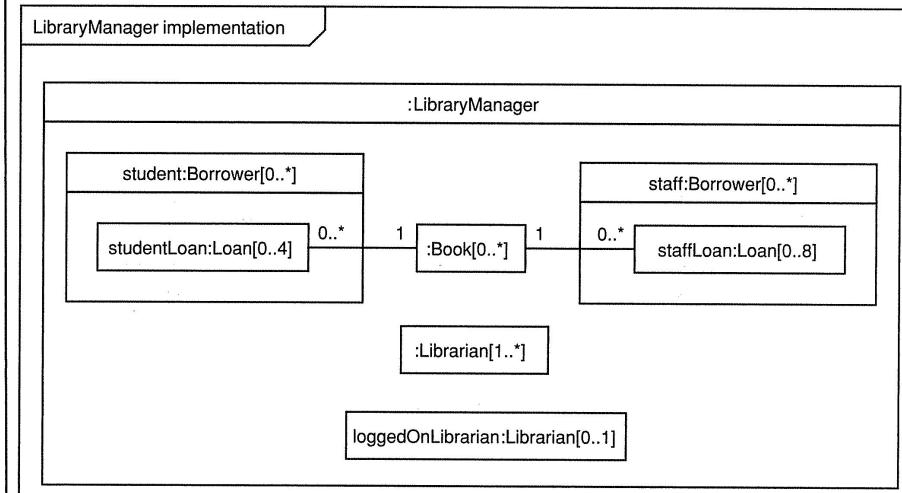
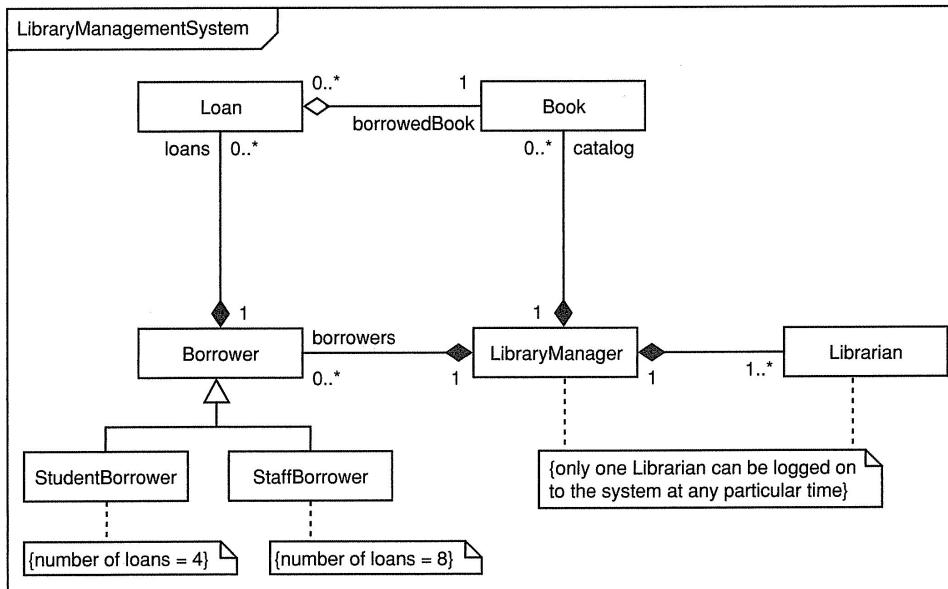


There are several **librarians**, but **only one can be logged on at any given time**



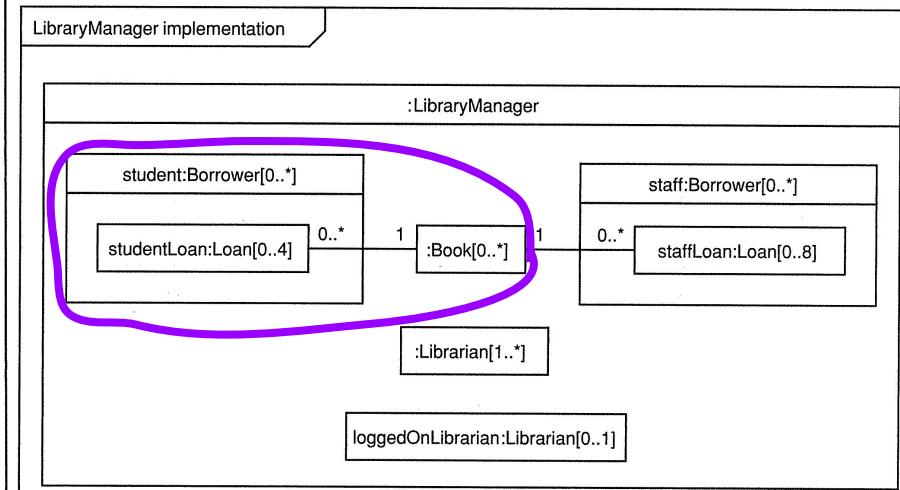
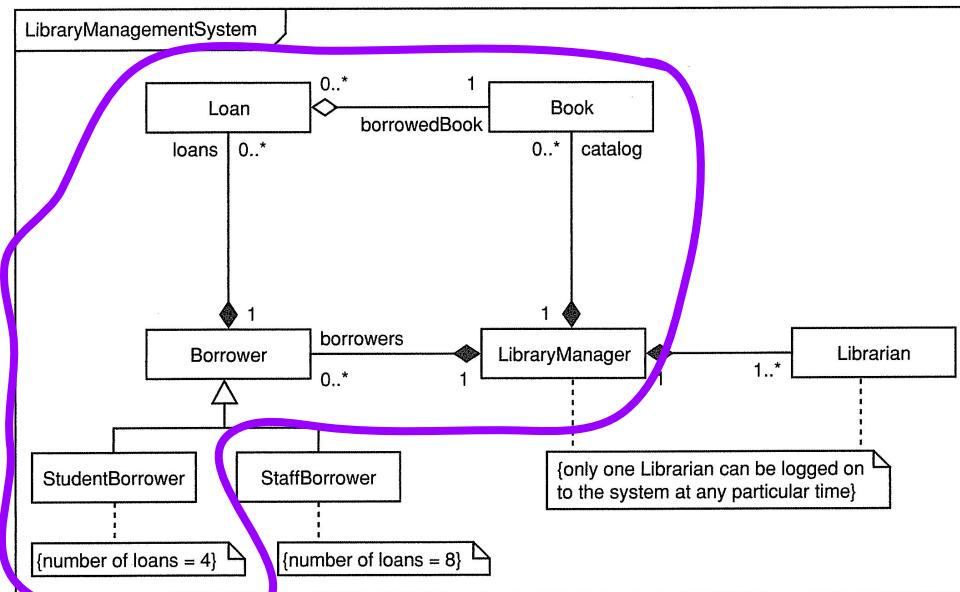
# Some association roles may map to part roles

- The roles played by instances in LibraryManager can differ from the roles classes play in their associations with LibraryManager, due to refinement



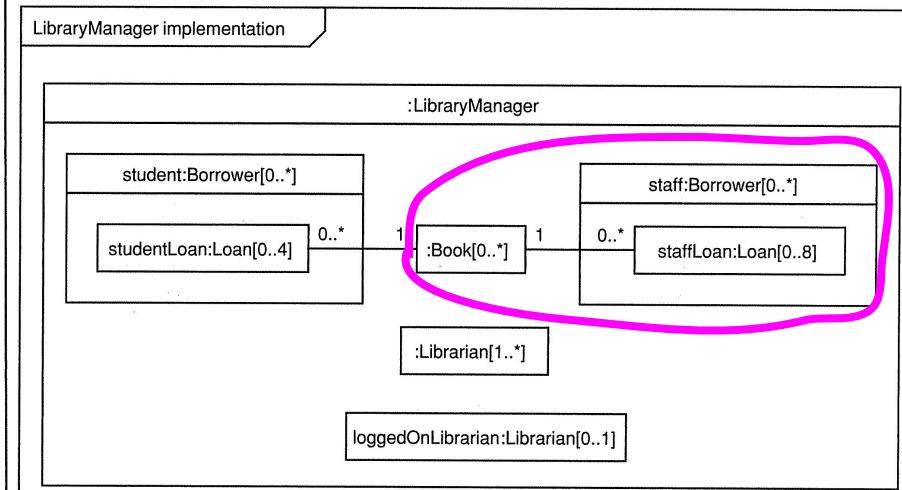
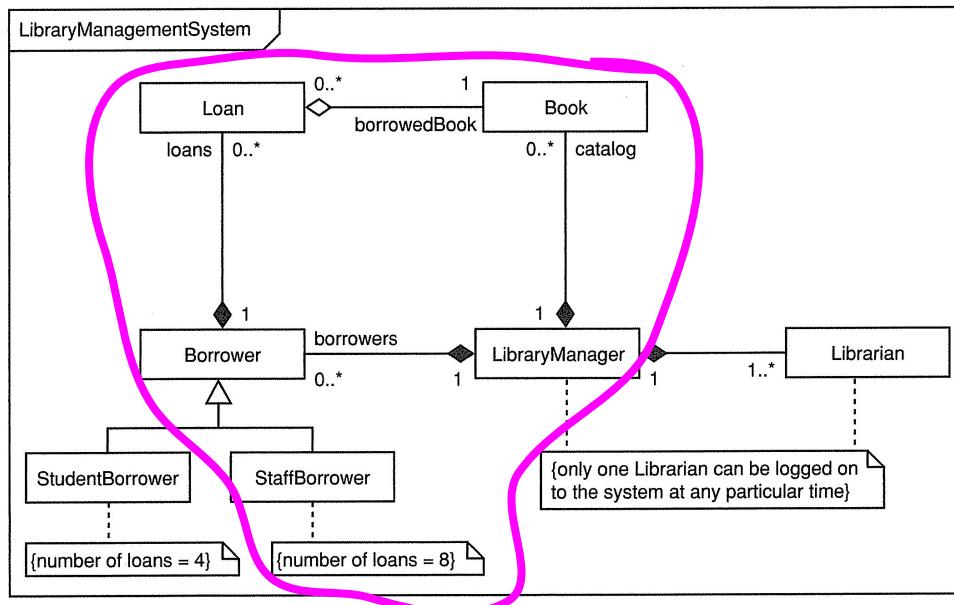
# Borrower class has the role borrowers

- This is refined into a more specific role played by the Borrower subclass StudentBorrower, for students

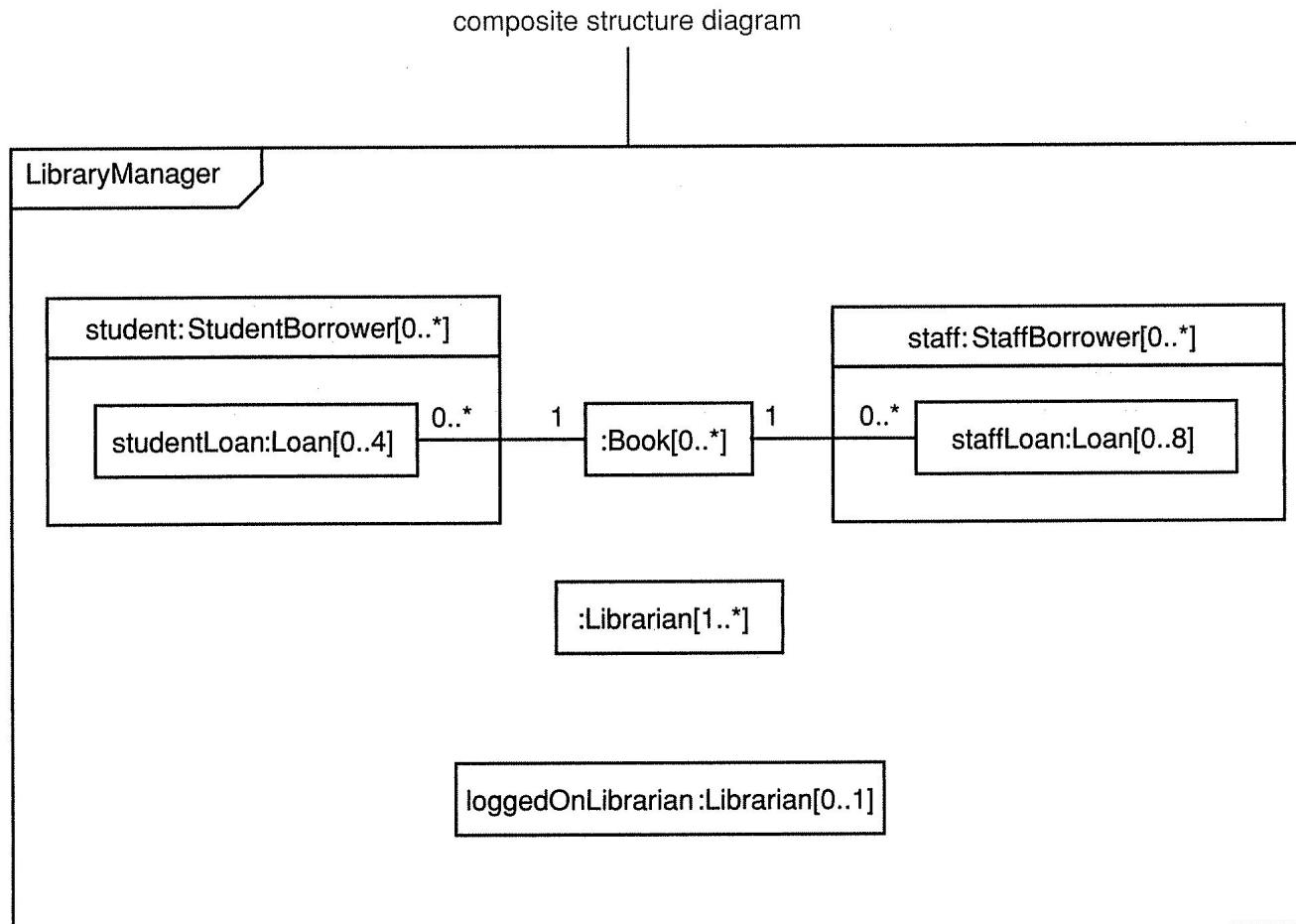


# Borrower class has the role borrowers

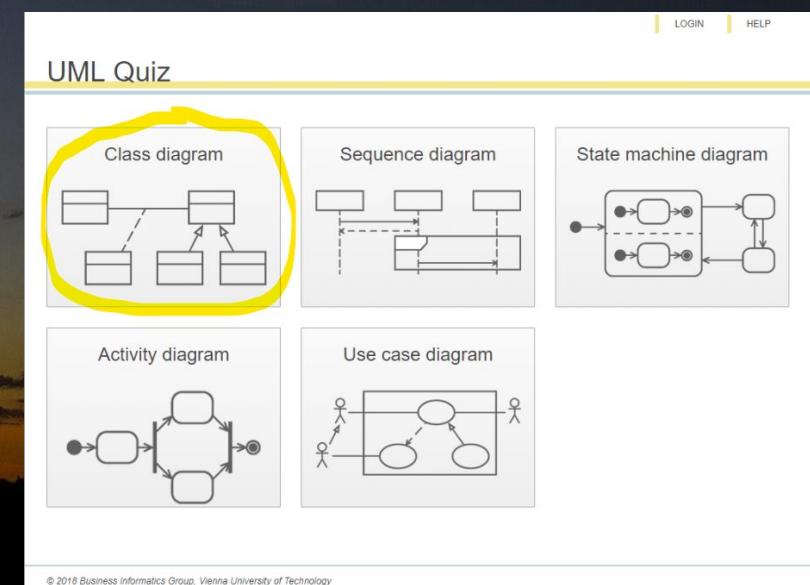
- This is refined into a more specific role played by the Borrower subclass StaffBorrower, for staff



# LibraryManager can also be represented in its own composite structure diagram



Did you really understand Design Class Diagrams? Test yourself at:  
<http://elearning.uml.ac.at/>



The screenshot shows a user interface for a UML quiz. At the top right are 'LOGIN' and 'HELP' buttons. The main title is 'UML Quiz'. Below it are five boxes, each containing a UML diagram type:

- Class diagram:** Shows two classes connected by a directed association. A yellow circle highlights this box.
- Sequence diagram:** Shows three lifelines with various message exchanges.
- State machine diagram:** Shows a state transition with initial and final states.
- Activity diagram:** Shows a flow from an initial node through several regions and back to a final node.
- Use case diagram:** Shows actors (represented by ovals with lines) interacting with use cases (represented by rounded rectangles).

At the bottom of the page is a copyright notice: © 2018 Business Informatics Group, Vienna University of Technology.

# Bibliography

---

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”,  
Second Edition, Addison-Wesley 2006

- Chapter 18