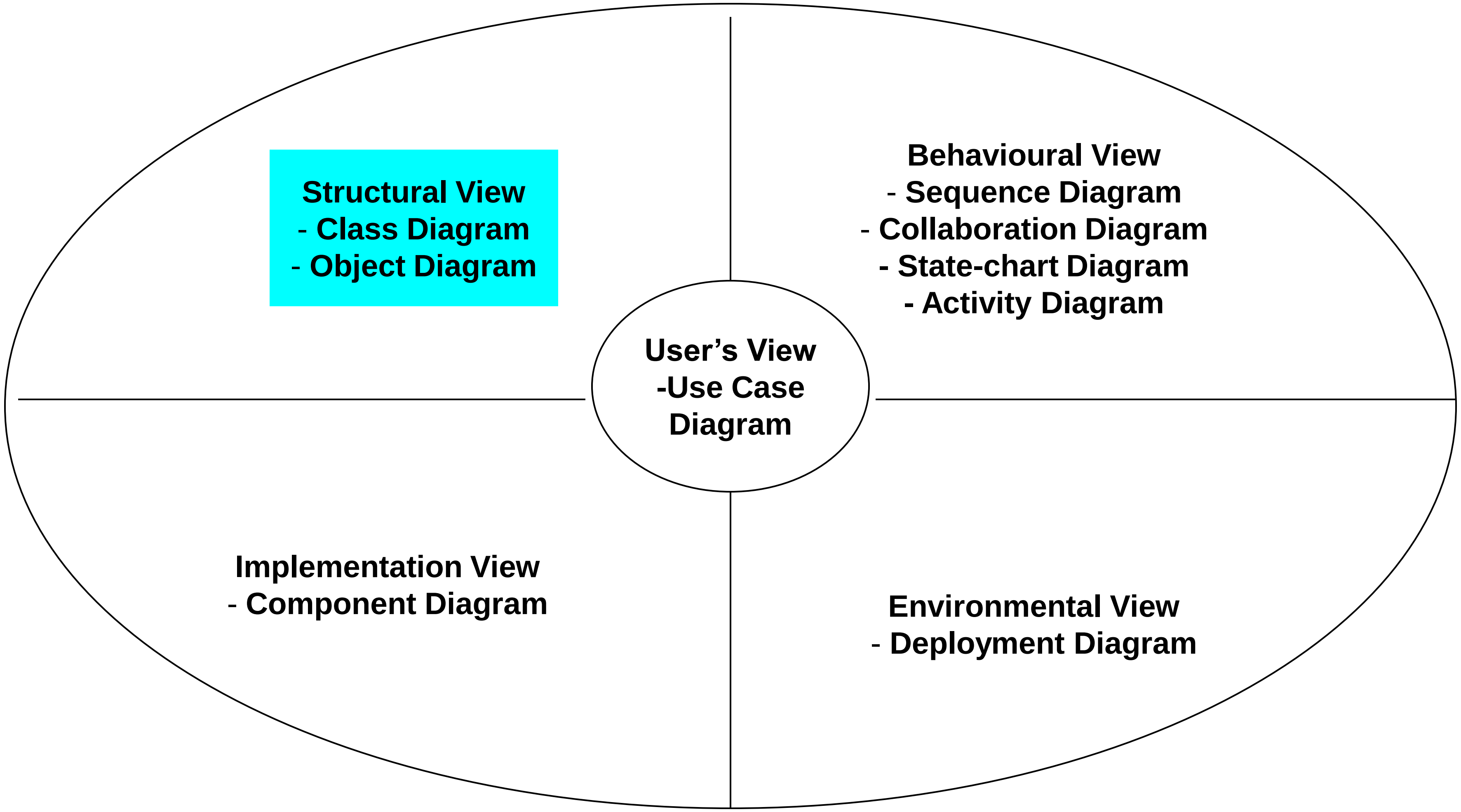
This image cannot currently be displayed.

Chapter 2

Structural View

UML Diagrams



Diagrams and views in UML

Class Diagram

- It is a schema, pattern or template for describing many possible instances of data.
- It describes the object class.
- A class diagram shows the existence of classes and their relationships in the logical view of a system
- UML modeling elements in class diagrams are:
  - Classes, their structure and behavior.
  - relationships components among the classes like association, aggregation, composition, dependency and inheritance
  - Multiplicity and navigation indicators
  - Role names or labels.

Object Diagram

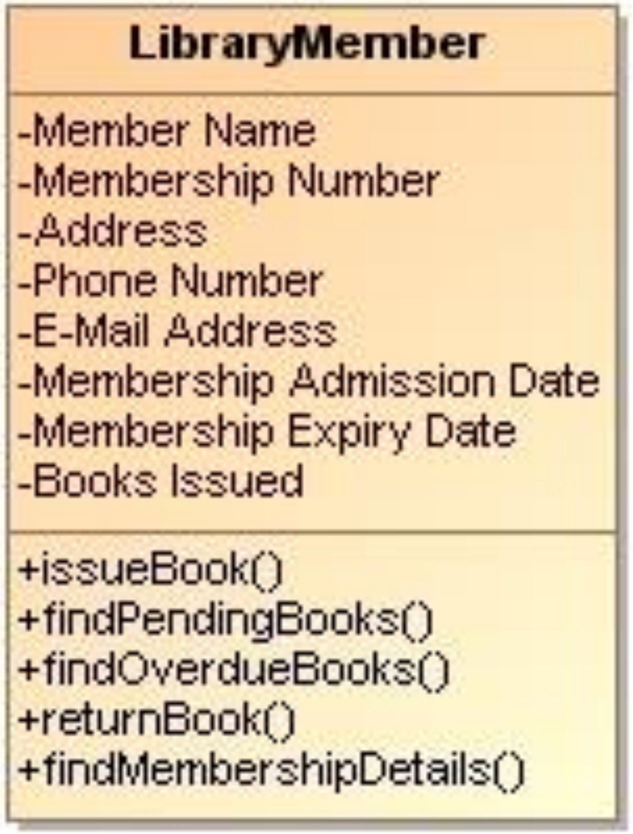
- It describes how a particular set of objects relate to each other. An object diagram describes object instances.

Classes

- Entities with common features, i.e. attributes and operations
- Classes are represented as *solid outline rectangle* with compartments
- Compartments for name, attributes, and operations.
- Attribute and operation compartments are optional depending on the purpose of a diagram.

Classes

- Represent the types of data themselves
- A class is simply represented as a box with the name of the class inside
- The diagram may also show the attributes and operations



Major Types of classes

Concrete classes

- A concrete class is a class that is instantiable; that is it can have different instances.
- Only concrete classes may be leaf classes in the inheritance tree.

Abstract classes

- An abstract class is a class that has no direct instance but whose descendants classes have direct instances.
- An abstract class can define the protocol for an operation without supplying a corresponding method we call this as an *abstract operation.*
- An abstract operation defines the form of operation, for which each concrete subclass should provide its own implementation.

Attributes

- An attributes is a data value held by the objects in a class.

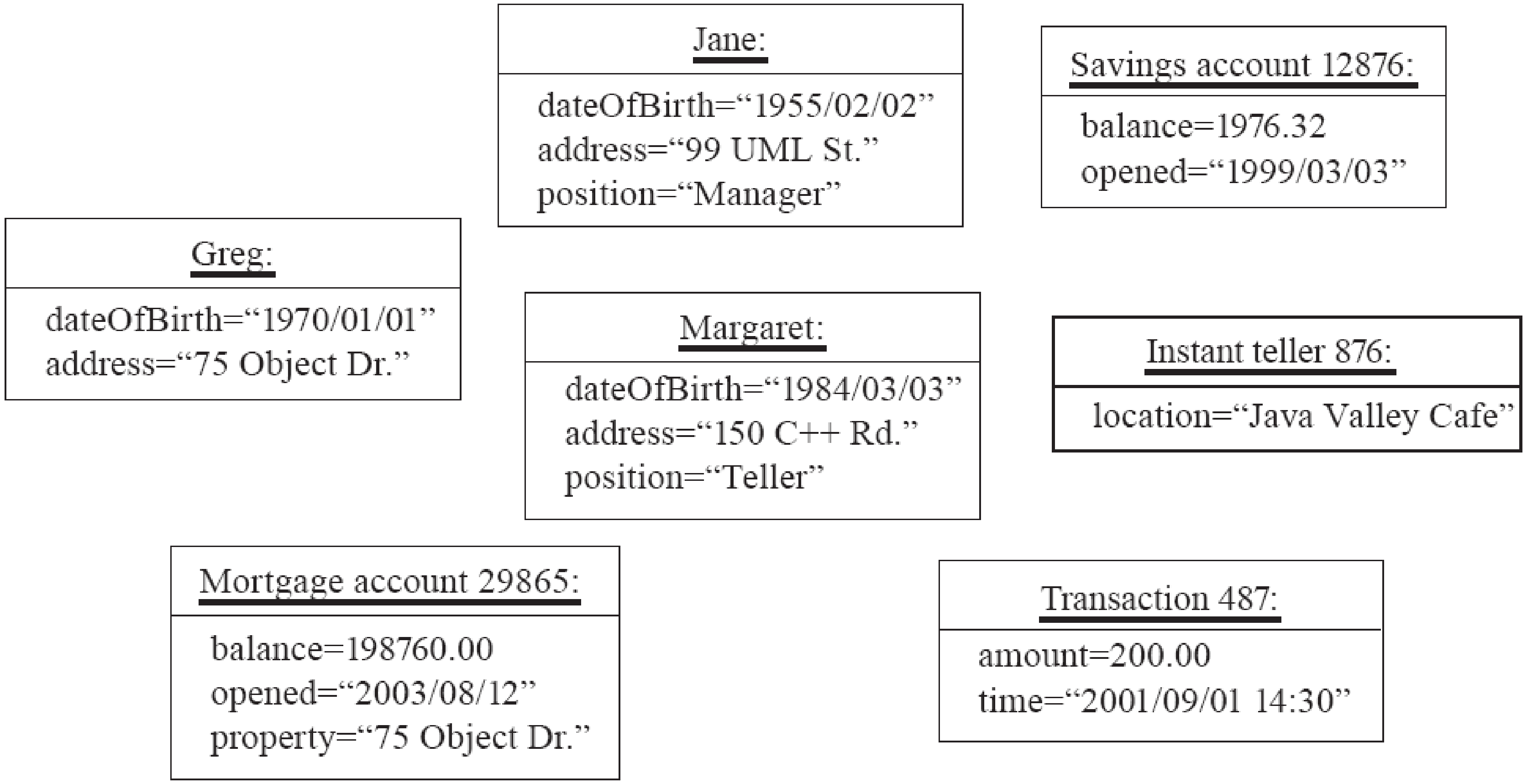
Operations and methods

- An operation is a function or transformation that may be applied to or by object in class.

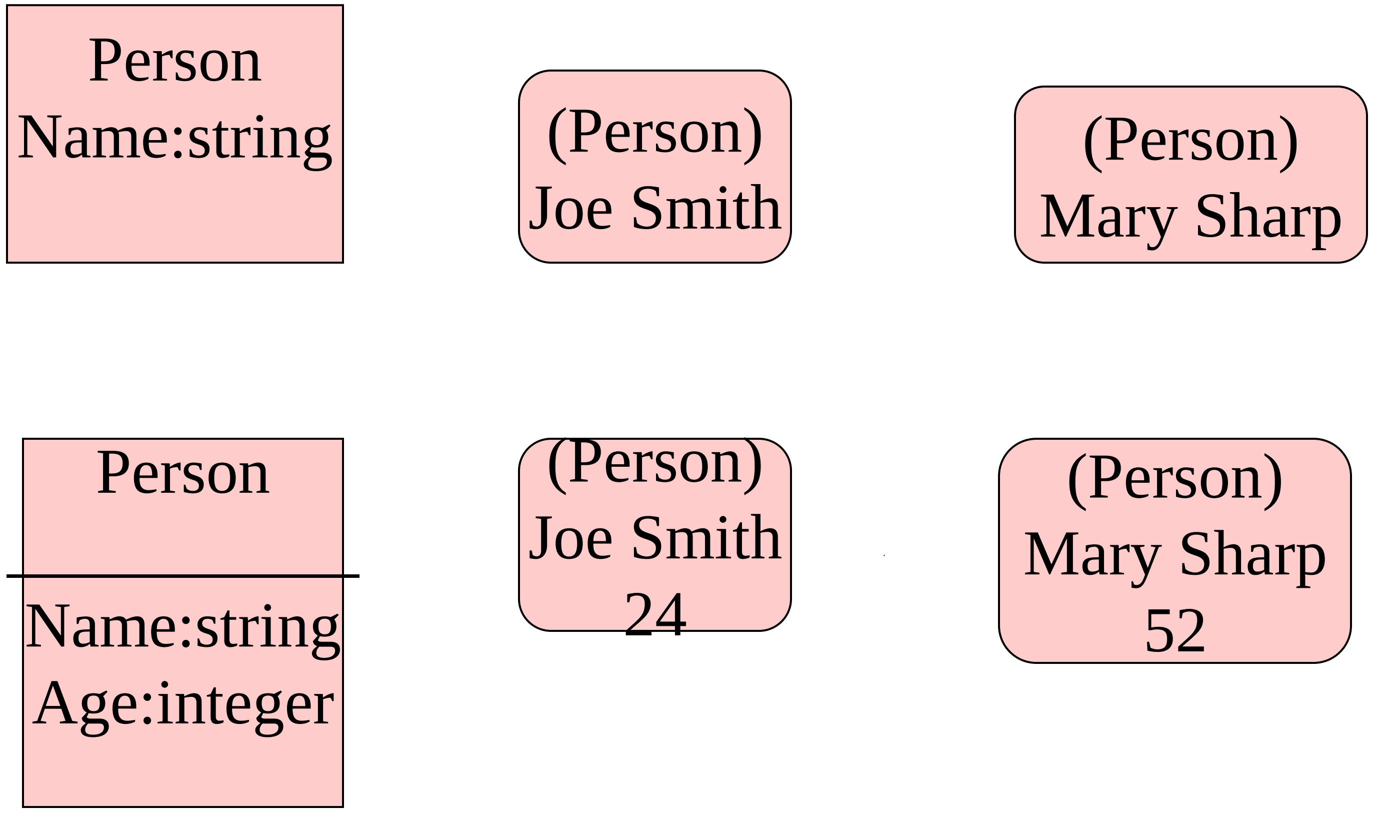
Objects

- A chunk of structured data in a running software system
- Has *properties*
    - Represent its state
- Has *behavior*
    - How it acts and reacts
    - May simulate the behavior of an object in the real world

Objects


Jane:
dateOfBirth="1955/02/02"
address="99 UML St."
position="Manager"
Savings account 12876:
balance=1976.32
opened="1999/03/03"
Greg:
dateOfBirth="1970/01/01"
address="75 Object Dr."
Margaret:
dateOfBirth="1984/03/03"
address="150 C++ Rd."
position="Teller"
Instant teller 876:
location="Java Valley Cafe"
Mortgage account 29865:
balance=198760.00
opened="2003/08/12"
property="75 Object Dr."
Transaction 487:
amount=200.00
time="2001/09/01 14:30"

Classes and Objects



Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class
- *Film*
  - Class; instances are individual films.
- *Reel of Film*:
  - Class; instances are physical reels
- *Film reel with serial number SW19876*
  - Instance of **ReelOfFilm**
- *Science Fiction*
  - Instance of the class **Genre**.
- *Science Fiction Film*
  - Class; instances include ‘Star Wars’
- *Showing of ‘Star Wars’ in the Phoenix Cinema at 7 p.m.***:**
  - Instance of **ShowingOfFilm**

Instance Variables

- Variables defined inside a class corresponding to data present in each instance
  - Attributes
    - Simple data
    - E.g. name, dateOfBirth
  - Associations
    - Relationships to other important classes
    - E.g. supervisor, coursesTaken

Variables vs. Objects

- A variable
  - *Refers* to an object
  - May refer to different objects at different points in time
- An object can be referred to by several different variables at the same time
- *Type* of a variable
  - Determines what classes of objects it may contain

Class variables

- A *class variable's* value is *shared* by all instances of a class.
  - Also called a *static* variable
  - If one instance sets the value of a class variable, then all the other instances see the same changed value.
  - Class variables are useful for:
    - Default or 'constant' values (e.g. PI)
    - Lookup tables and similar structures

Caution: *do not over-use class variables*

Relationship

- Association
- Aggregation
- Composition
- Inheritance
- Dependency
- Instantiation

Association Relationship

- It is a group of links with common structure and common semantics.
- Are often implemented in programming languages as pointers from one object to another.
- It denotes a semantic connection between two classes
- It shows bi-directional connection between two classes
- It is a weak coupling as associated classes remain somewhat independent of each other

Association Relationship

- Enables objects to communicate with each other
- Usually binary but in general can be n-ary
- A class can be associated with itself (**recursive** association)
- An arrowhead used along with name, indicates direction of association
- Multiplicity indicates number of instances taking part in the association

Multiplicity

- It specifies how many instance of one class may relate to each instance of another class.
- Types
  - One to one
  - Many to many
  - One to many
  - Zero or one or optional multiplicity

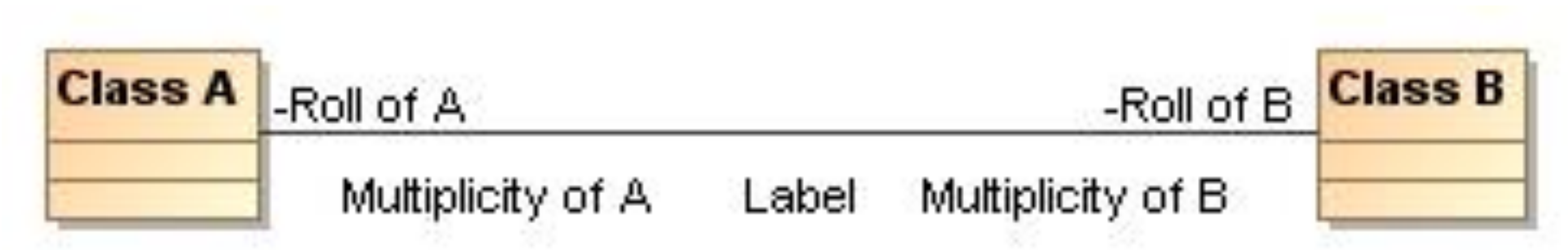
Multiplicity Indicators.

Indicator	**Meaning**
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

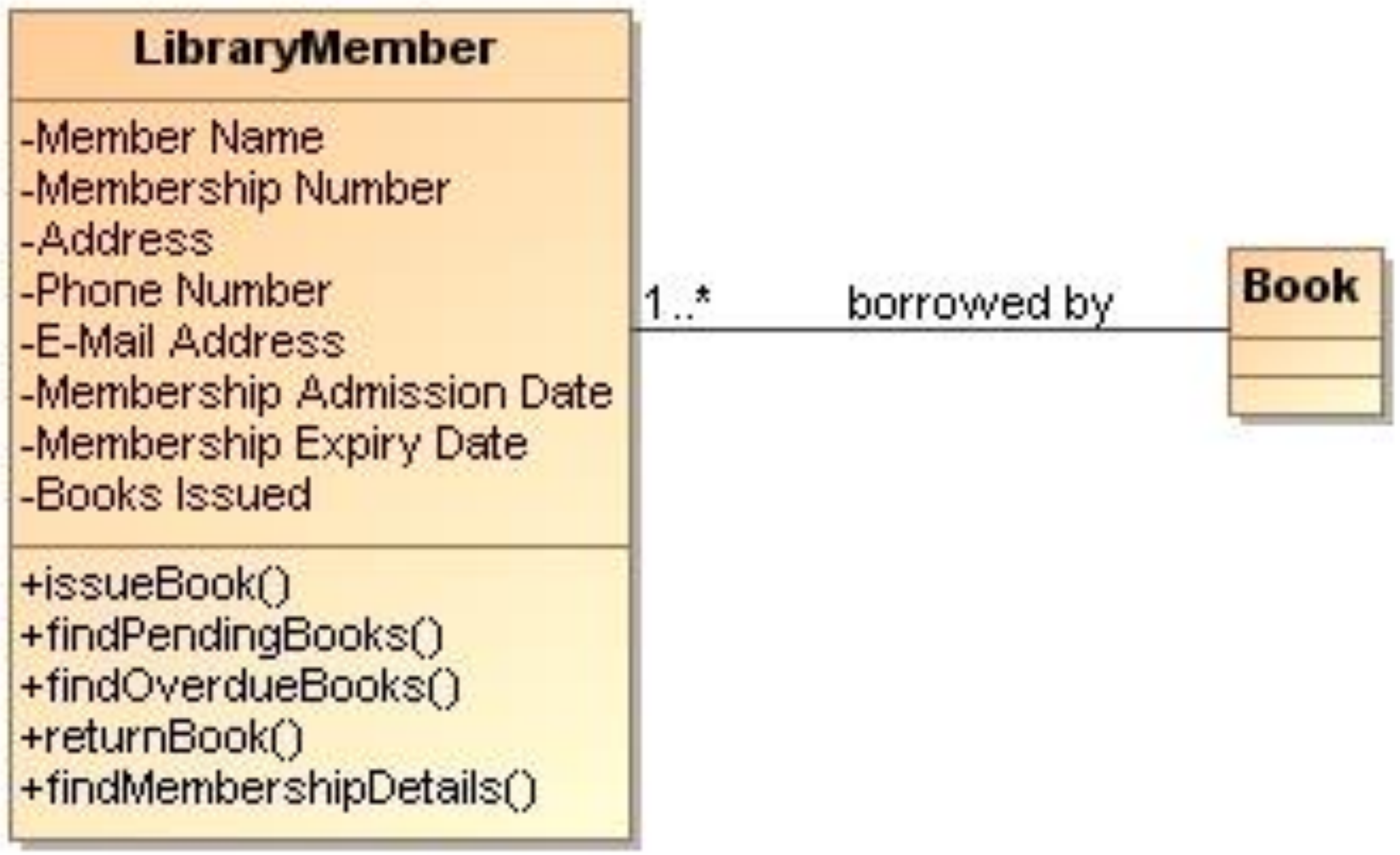
Role names

- It is a name the uniquely identifies one end of an associations roles often appear as nouns in problem descriptions.
- Use of role names are optional.

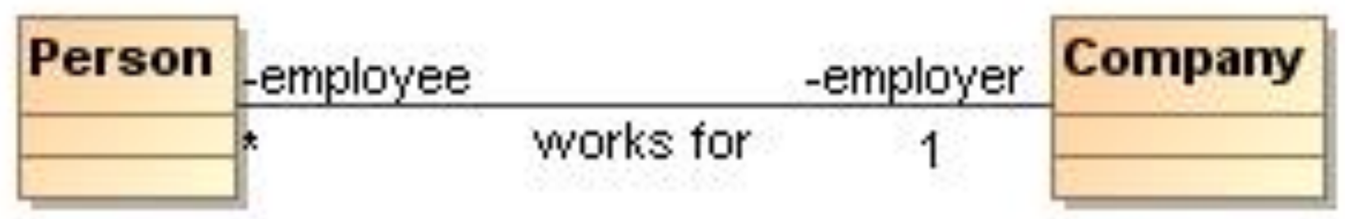
Association Relationship



Association Relationship

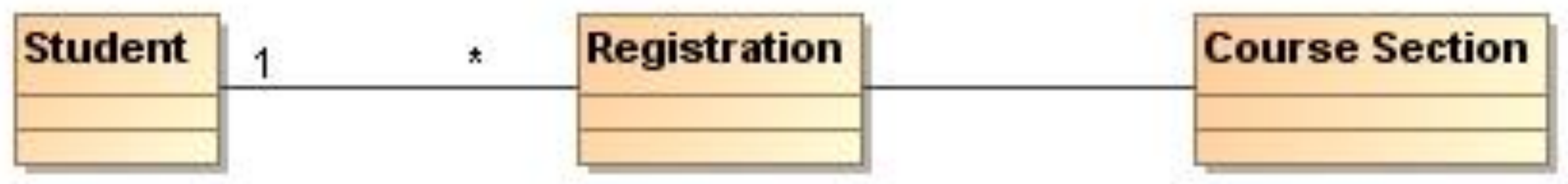


Association Relationship with Role name



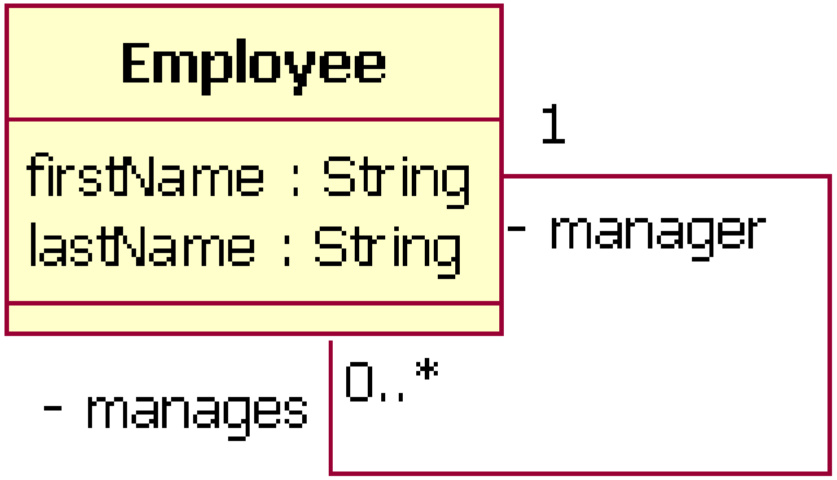
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes



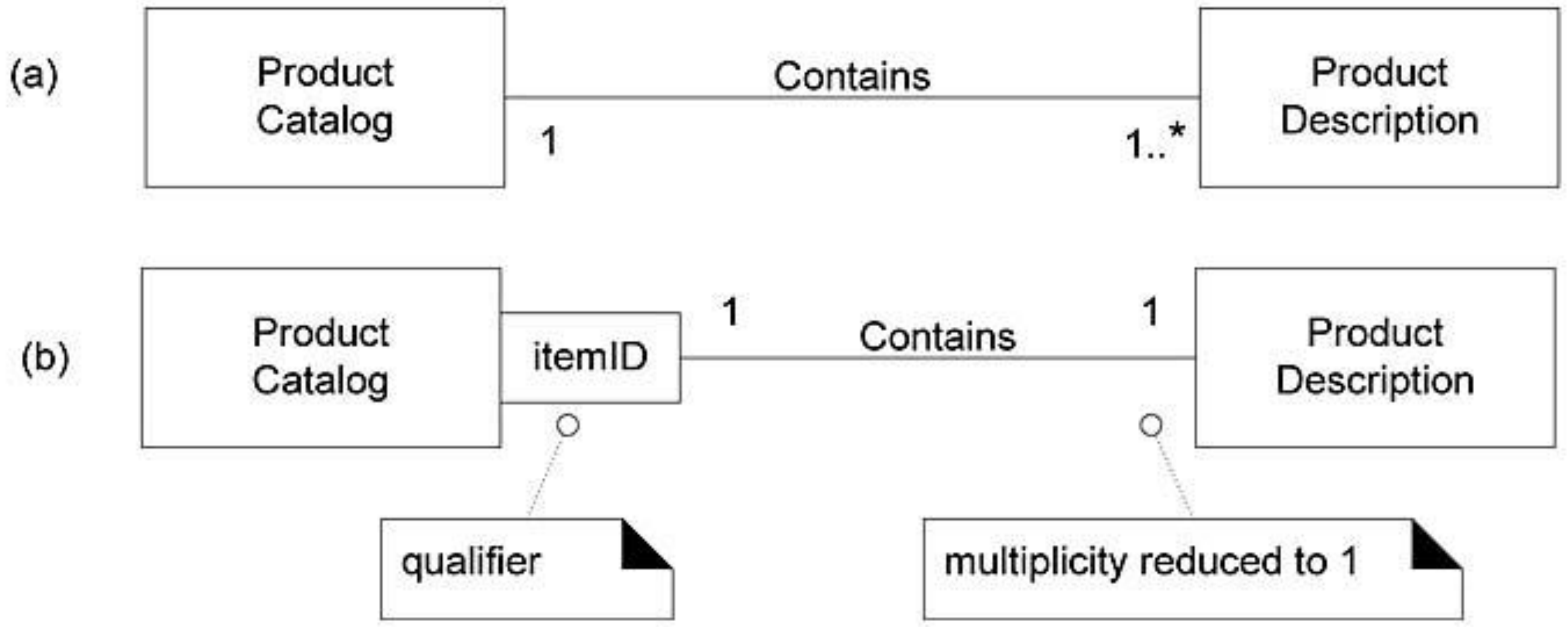
Recursive/ Reflexive associations

- It is possible for an association to connect a class to itself



Qualified associations

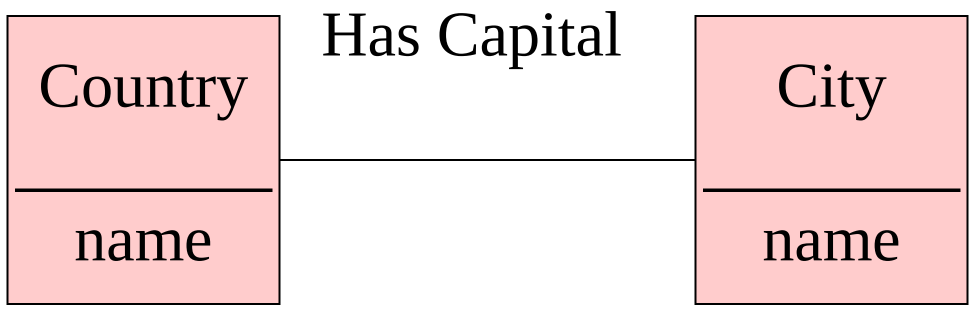
- A qualified association has a qualifier that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key.

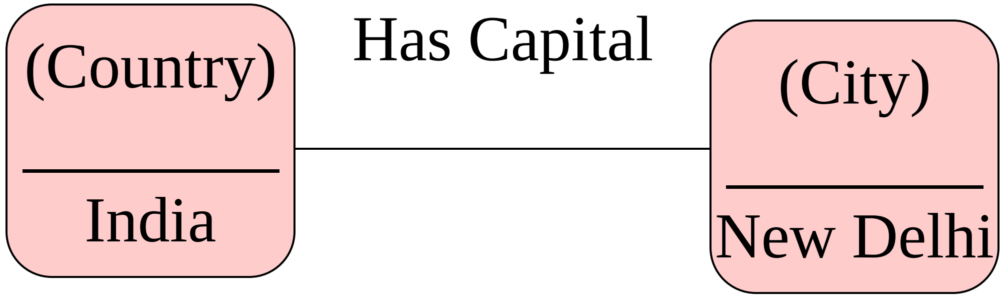


Links

- A link is a physical or conceptual connection between object instances.
- A link is a instance of an association.

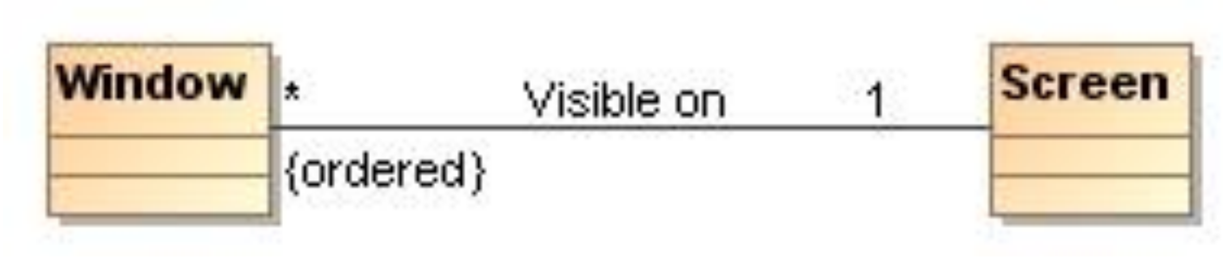
Links and Associations





Ordering

- Usually objects on the many side of an association have no explicit order.
- It can be regarded as set.
- Sometimes, the objects are explicitly ordered.



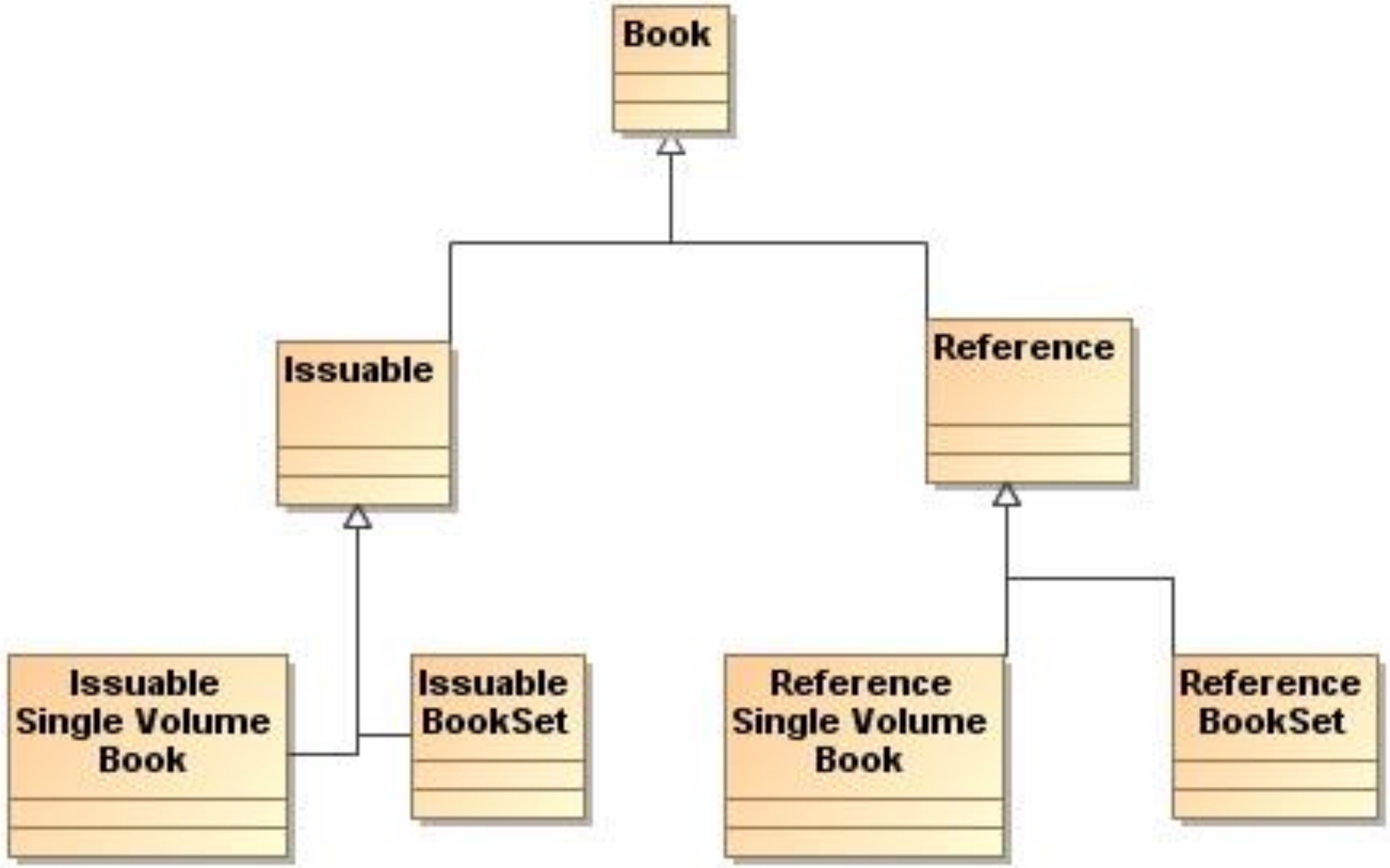
Qualification

- A qualified association relates two object classes and a qualifier.
- The qualifier is a special attribute that reduces the effective multiplicity of an association.

Generalization and Inheritance

- These are powerful abstractions for sharing similarities among classes while preserving their differences.
- Generalization is the relationship between a class and one or more refined versions of it.
- The class being refined is called the superclass and each refined version is called a subclass.

Inheritance Relationship

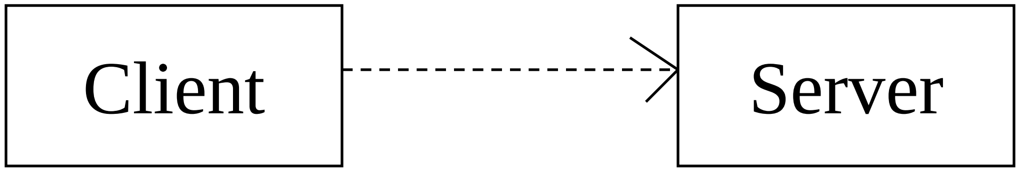


Module

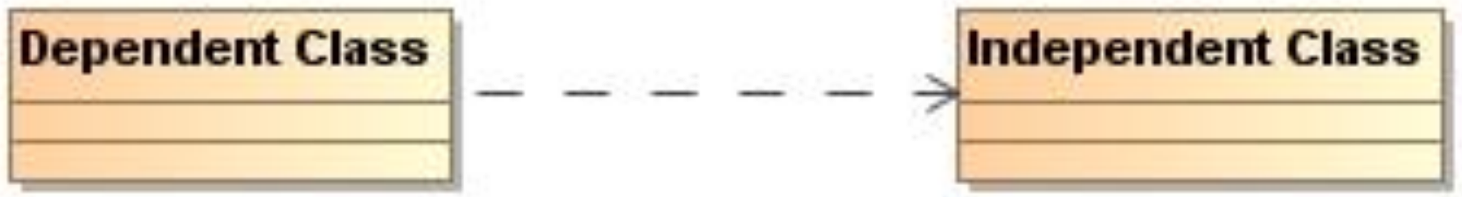
- A module is a logical construct for grouping classes, associations and generalizations.

Class Dependency

- Dependency is semantic connection between dependent and independent model elements.
- This association is unidirectional and is shown with dotted arrowhead line.
- In the following example it shows the dependency relationship between client and server.
- The client avails services provided by server so it should have semantic knowledge of server.
- The server need not know about client.

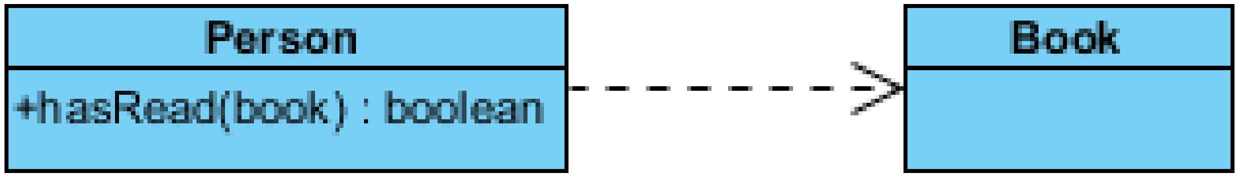


Class Dependency



Dependency

- An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.
  - A special type of association.
  - Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
  - Class1 depends on Class2
  - The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.

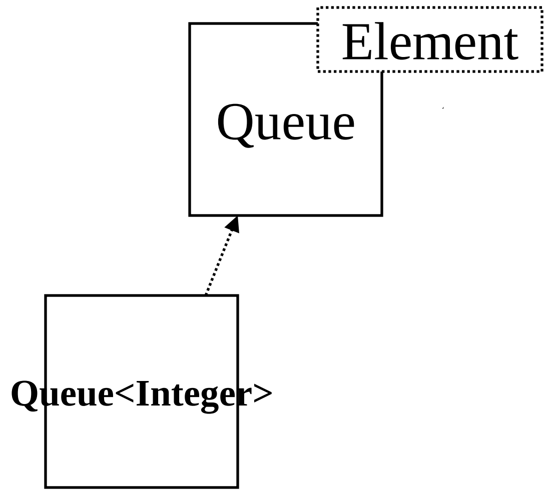


Instantiation

This relationship is defined between parameterized class and actual class.

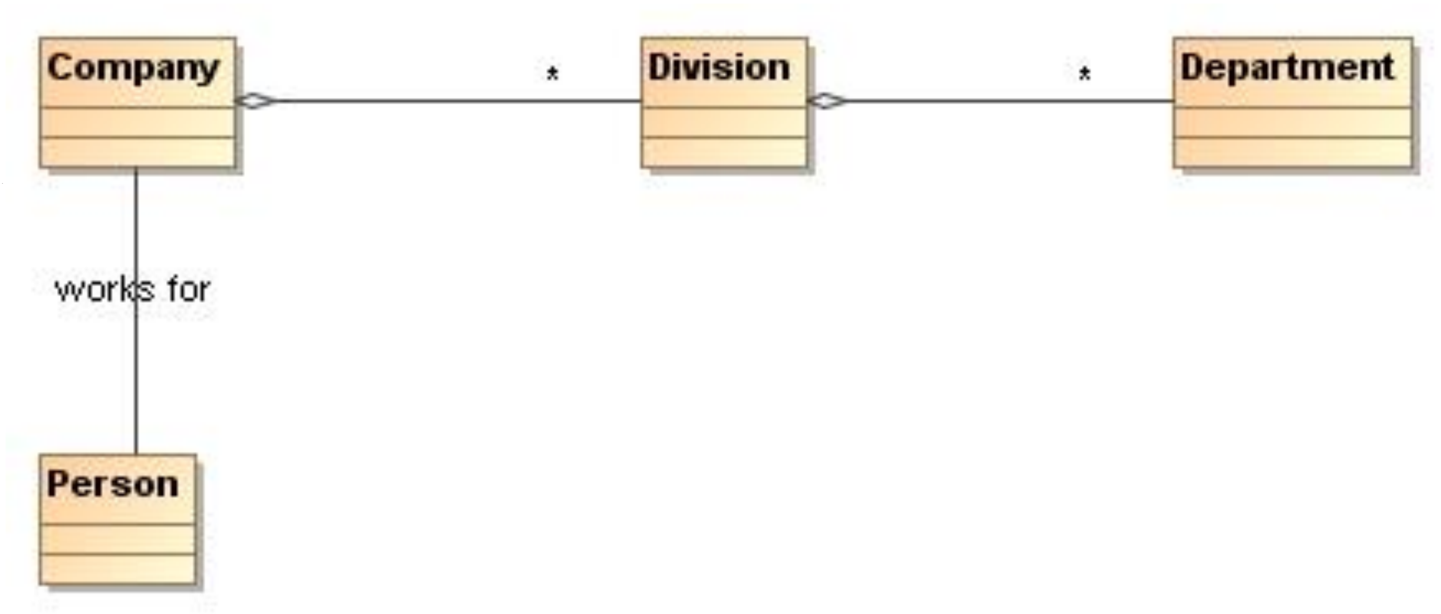
- Parameterized class is also referred as generic class.
- A parameterized class can't have instances unless we first instantiated it

Example:



Aggregation Vs Association

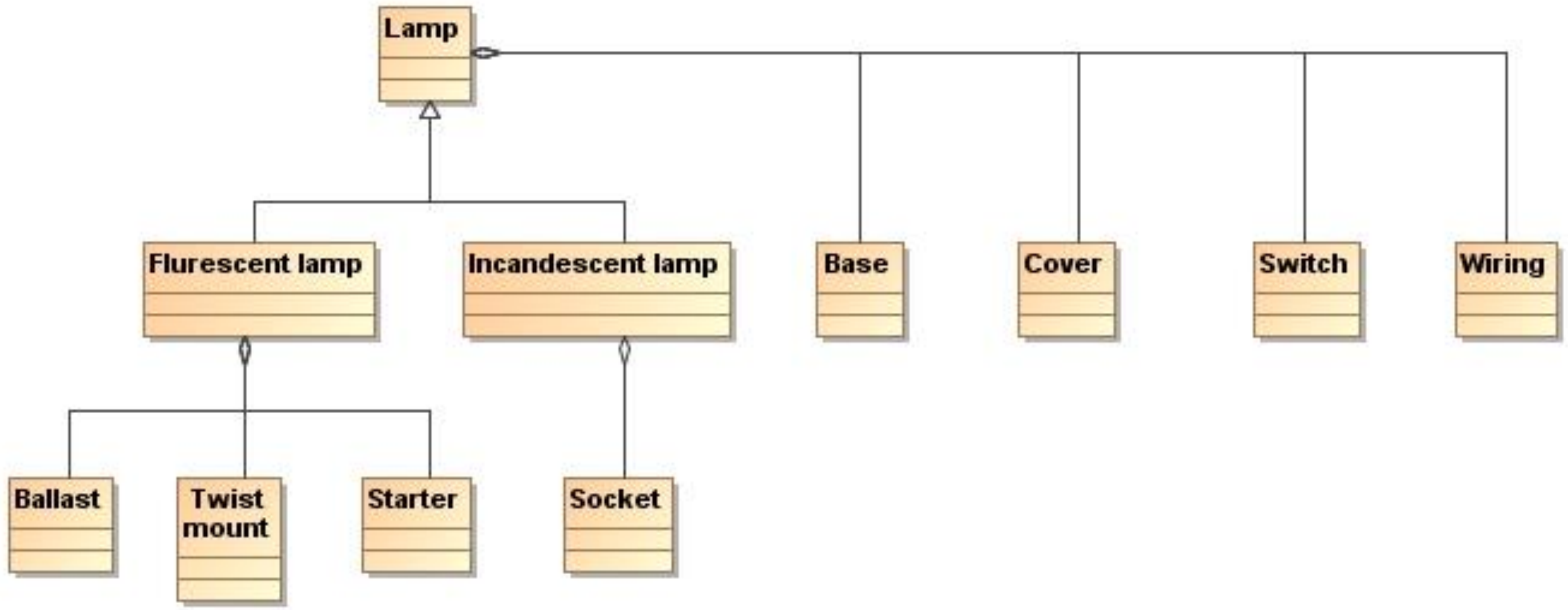
- Aggregation is a special form of association
- If two objects are tightly related by a part whole relationship, it is an aggregation.
- If the two objects are usually considered as independent, even though they may often be linked, it is an association.



Aggregation Vs Generalization

- Aggregation relates instances.
- Among two distinct objects one of them is a part of the other.
- Generalization relates classes and is a way of structuring the description of a single object.
- Both subclass and superclass refer to properties of a single object.

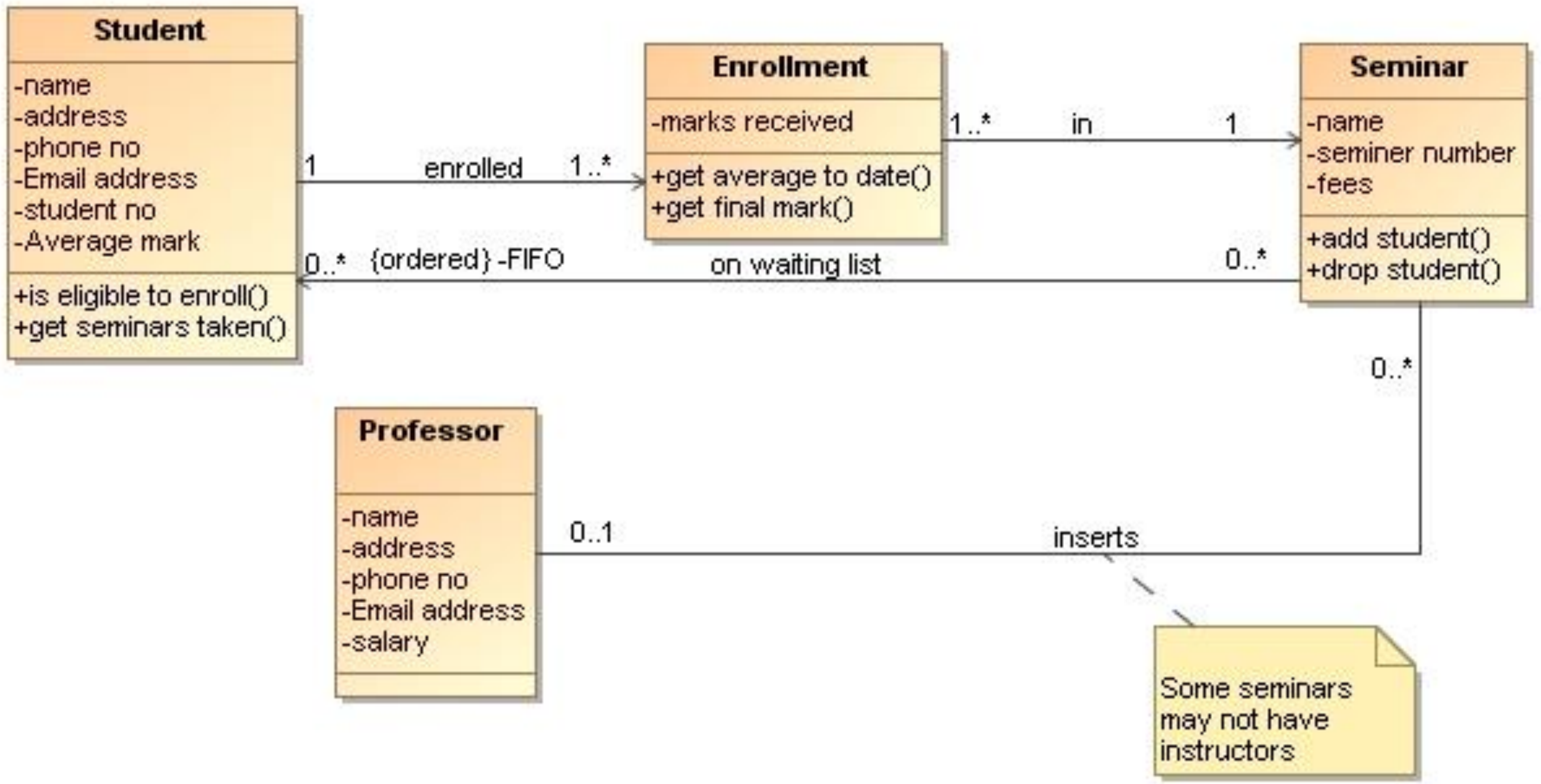
Aggregation Vs Generalization



Types of aggregation

- Fixed: Has a fixed structure. The number and types of subparts are predefined.
- Variable: Has a finite number of levels, but the number of parts vary.
- Recursive :It contains directly or indirectly an instance of the kind of aggregate. The number of potential level is unlimited.

A Class Diagram

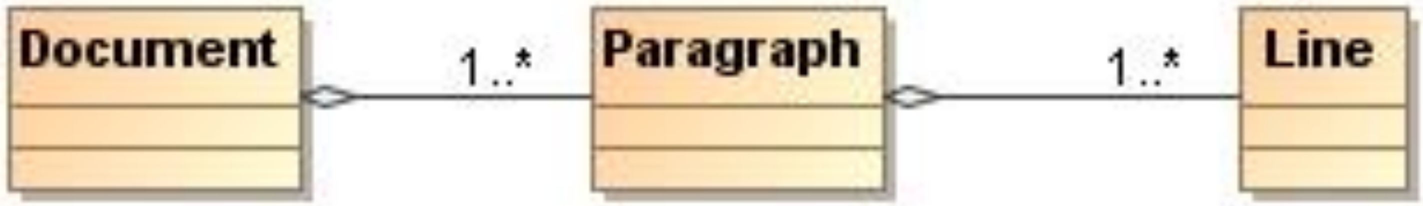


Aggregation Relationship

This is a special type of association

- Represents whole-part relationship
- Represented by a diamond symbol at the composite end
- Cannot be reflexive(i.e. recursive) (?)
- Not symmetric
- It can be transitive
- The association with label “contains” or “is part of” is an aggregation
- It represents “has a” relationship
- It is used when one object logically or physically contains other
- The container is called as aggregate
- The components of aggregate can be shared with others

Aggregation Relationship



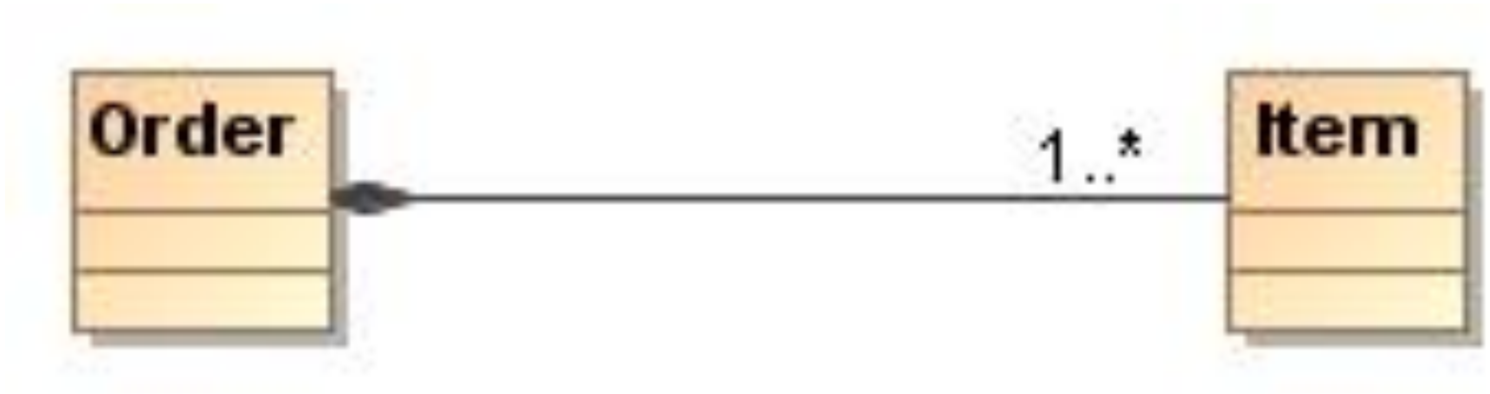
Composition Relationship

This is a strong form of aggregation

- It expresses the stronger coupling between the classes
- The owner is explicitly responsible for creation and deletion of the part
- Any deletion of whole is considered to cascade its part
- The composition has a filled diamond at its end

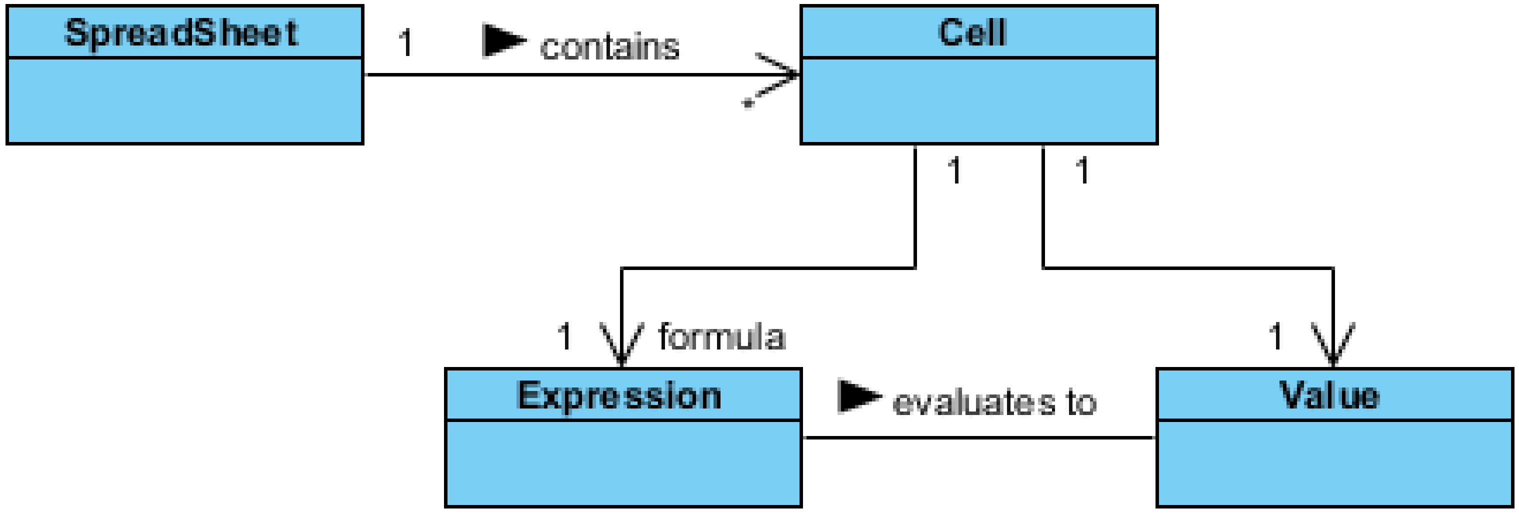
Composition Relationship

- Life of item is same as the order

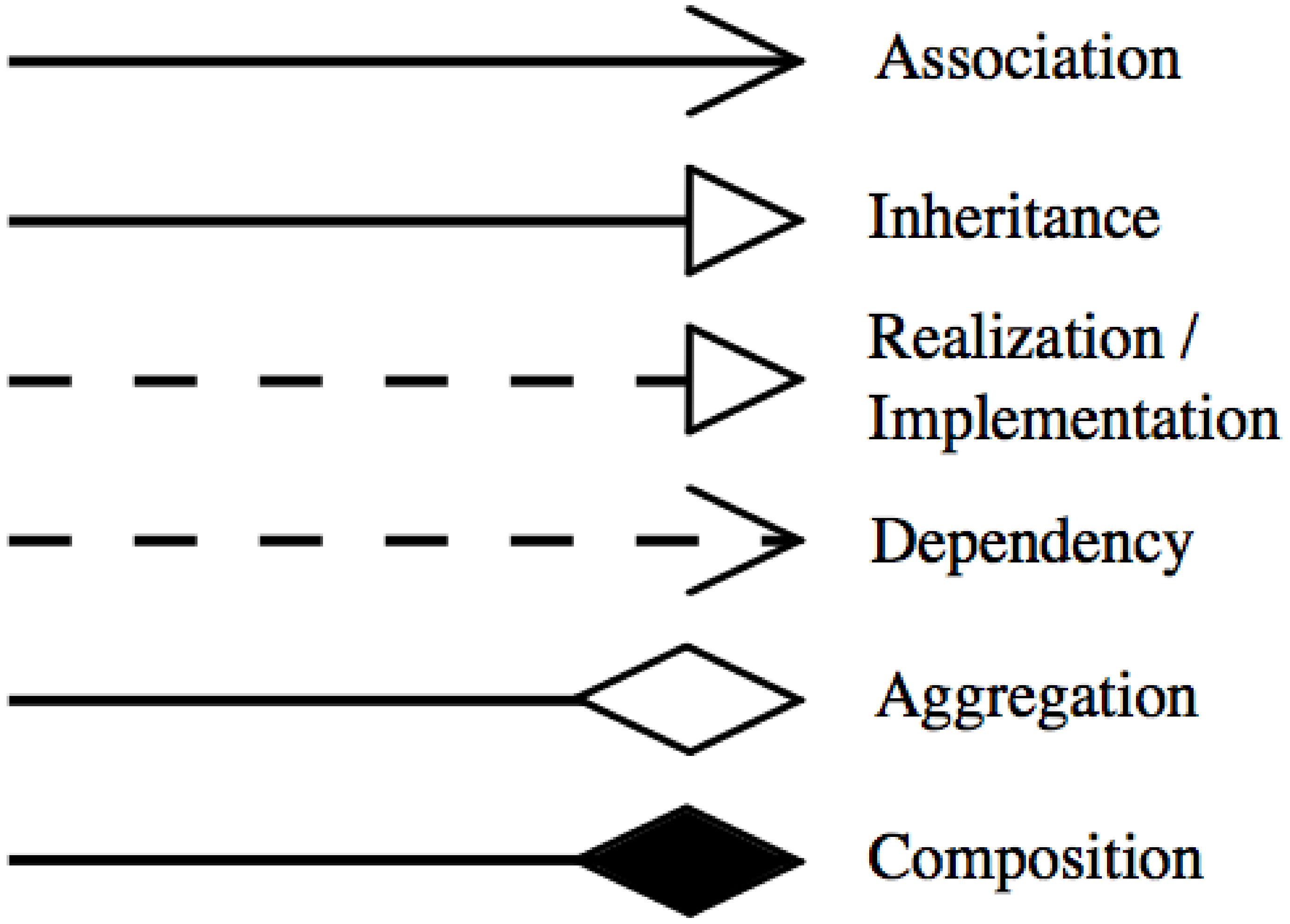


Relationship Names

- They often have a **small arrowhead to show the direction** in which direction to read the relationship, e.g., expressions evaluate to values, but values do not evaluate to expressions.



Relationships



Constraint

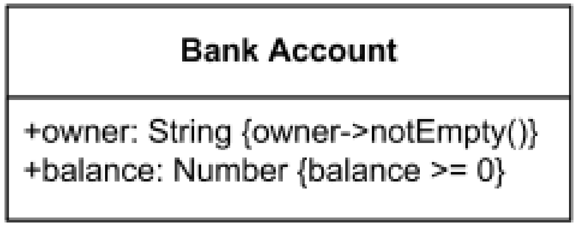
- A **constraint** is a packageable element representing **condition**, **restriction** or **assertion** related to some of the **semantics** of an element (that owns the constraint).
- Some kinds of constraints are **predefined** in UML, and **constraint** element allows to introduce **user defined** conditions and restrictions.
- Constraint is a **condition** (a Boolean expression) which must evaluate to a Boolean value - true or false. Constraint must be satisfied (i.e. evaluated to **true**) by a correct design of the system.

Constraint

- In general there are many possible kinds of **owners** for a Constraint. Owning element must have access to the constrained elements to verify constraint. The **owner** of the constraint will determine when the constraint is evaluated. For example, **Operation** can have pre-condition and/or a post-condition constraints.
- Constraints are commonly unnamed (anonymous), though constraint may have an optional name.
- **OCL** (Object Constraint Language) is a constraint language predefined in UML but UML specification does not restrict languages which could be used to describe constraints. If some tool is used to draw UML diagrams, it could be any constraint language supported by that tool.

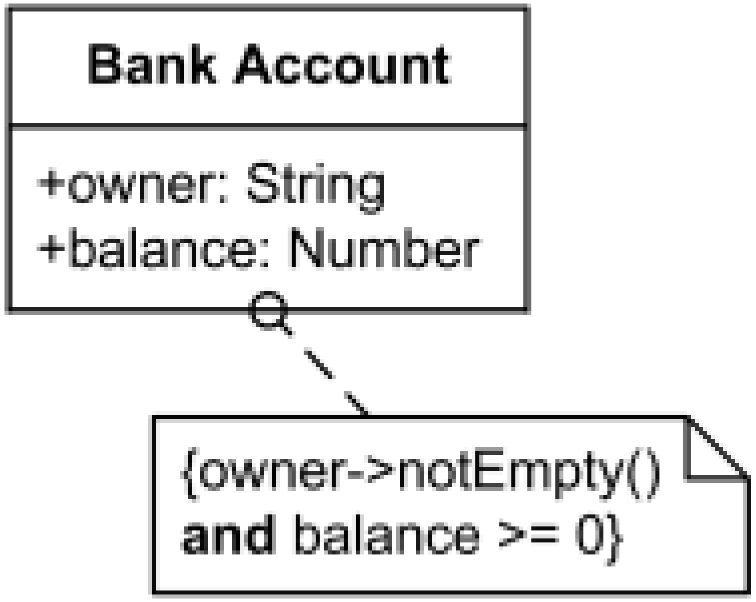
Constraint

- A constraint is shown as a text string **in curly braces** : <constraint> ::= ‘{‘ [<name> ‘:’] <Boolean-expression> ‘ }’

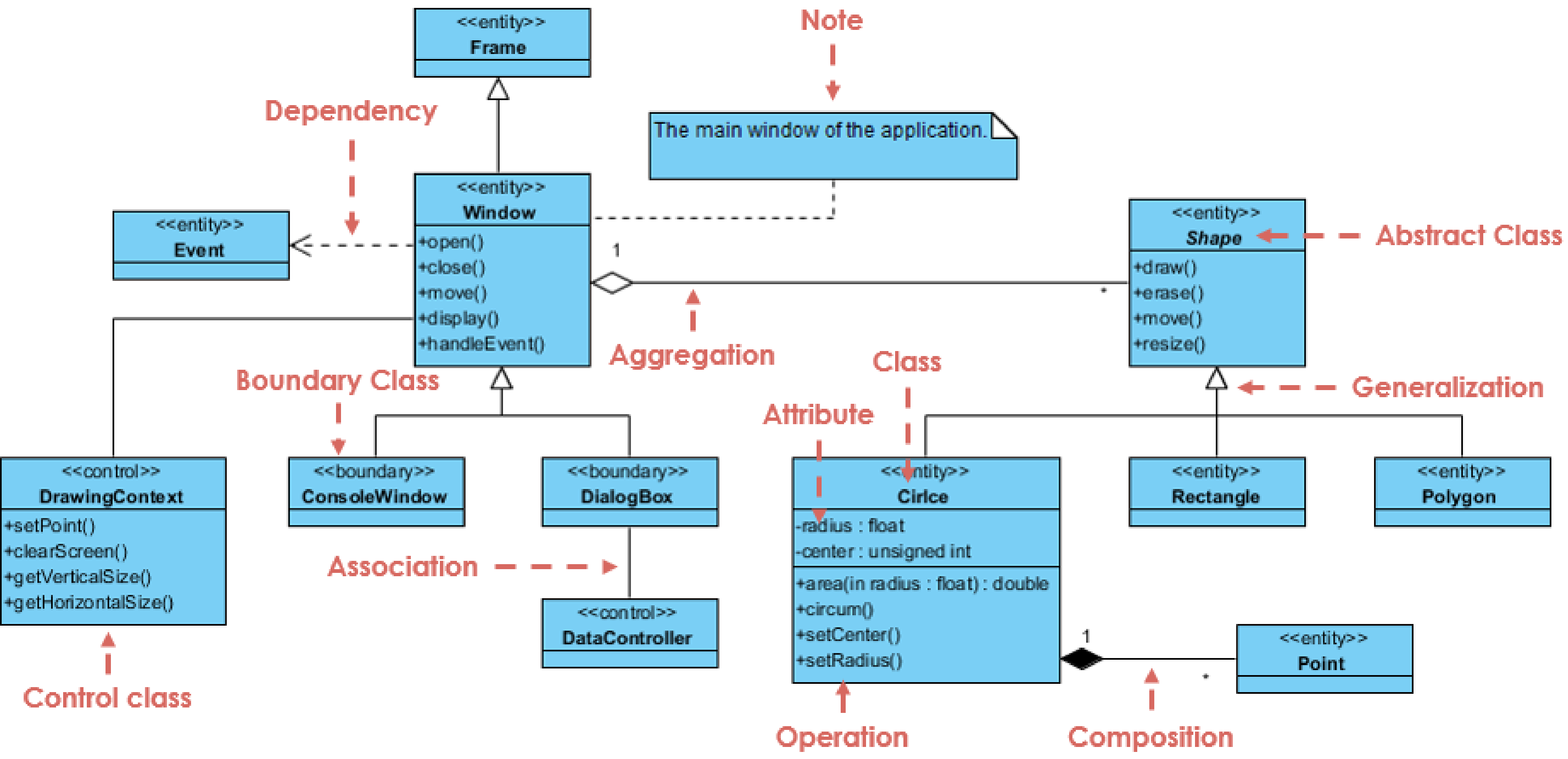

Bank Account
+owner: String {owner->notEmpty()}
+balance: Number {balance >= 0}

Constraint

- The constraint string may be placed in a **note** symbol and attached to each of the symbols for the constrained elements by a dashed line.



Class Diagram



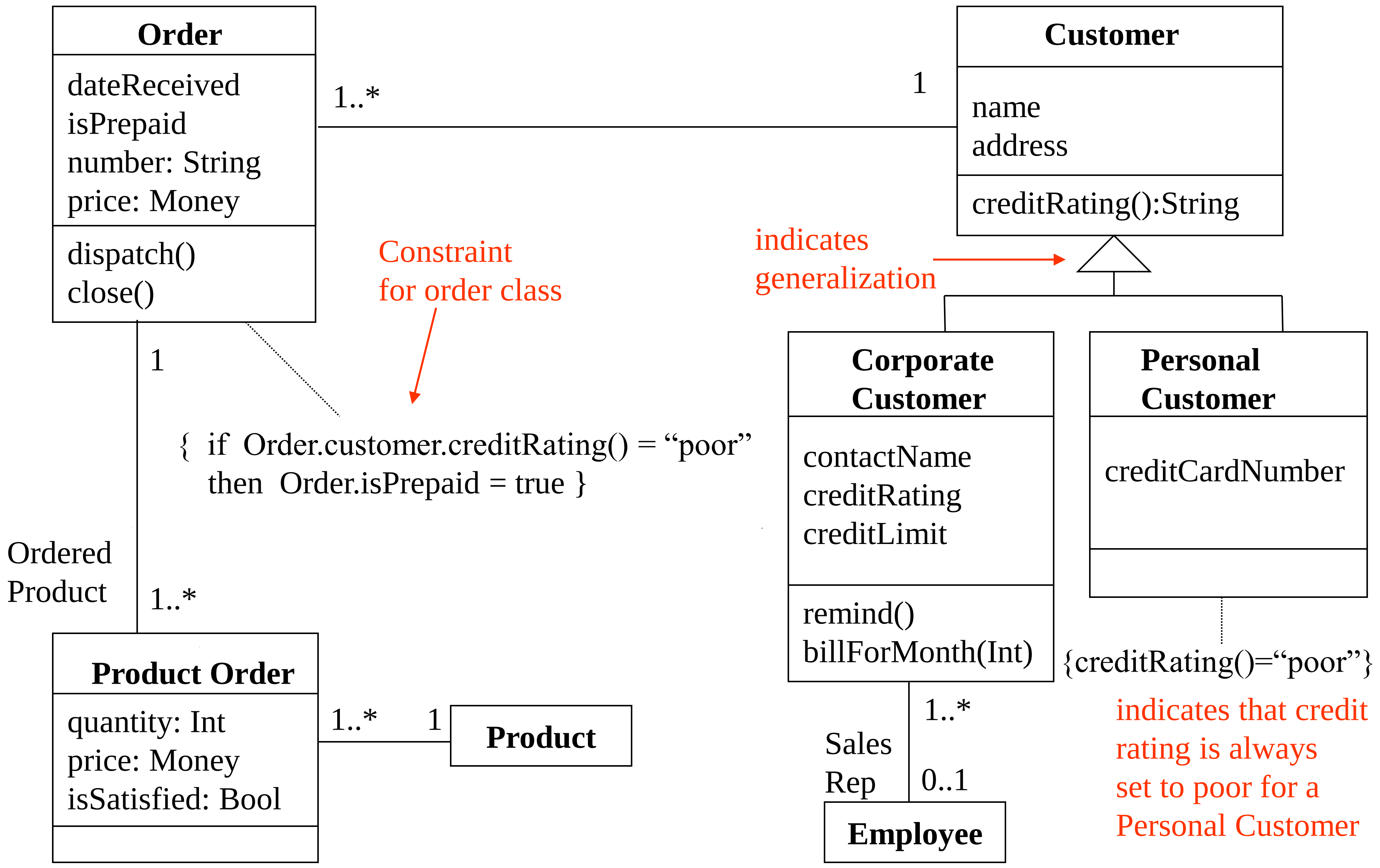
Class Diagram

- Shape is an abstract class. It is shown in Italics.
- Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. In other words, a Circle is-a Shape. This is a generalization / inheritance relationship.
- There is an association between DialogBox and DataController.
- Shape is part-of Window. This is an aggregation relationship. Shape can exist without Window.
- Point is part-of Circle. This is a composition relationship. Point cannot exist without a Circle.
- Window is dependent on Event. However, Event is not dependent on Window.

Class Diagram

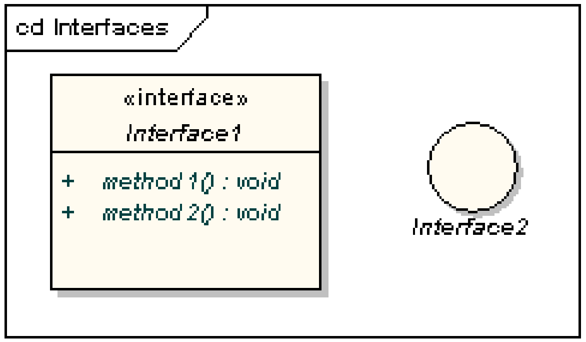
- The attributes of Circle are radius and center. This is an entity class.
- The method names of Circle are area(), circum(), setCenter() and setRadius().
- The parameter radius in Circle is an in parameter of type float.
- The method area() of class Circle returns a value of type double.
- The attributes and method names of Rectangle are hidden. Some other classes in the diagram also have their attributes and method names hidden.

Class Diagram



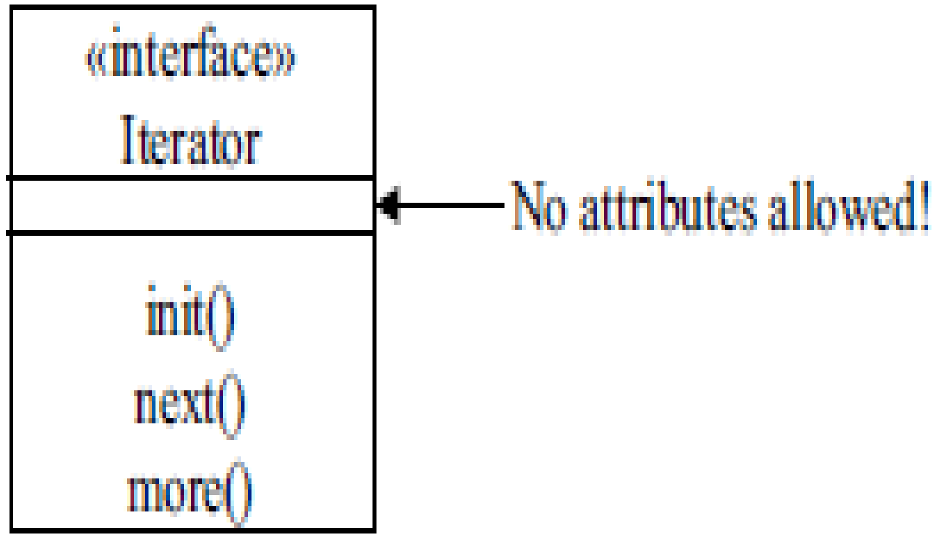
Interfaces

- An interface is a collection of operations (not data) that specifies a particular service of a class or a component
  - For instance, lists, queues, stacks, and trees typically provide an Iterator interface that allows other classes to cycle through their elements
- An interface is similar to a class but with a number of restrictions. All interface operations are public and abstract, and do not provide any default implementation. All interface attributes must be constants. However, while a class may only inherit from a single super-class, it may implement multiple interfaces.



Interfaces UML Notation

- The most simple notation for an interface is a labeled circle
- However, a full class diagram can be used to specify the particular operations associated with an interface

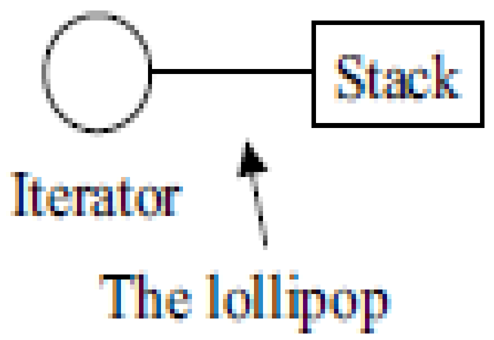


How interfaces are used

- You cannot instantiate an instance of an interface, instead other classes (and thus their objects) choose to implement certain interfaces.

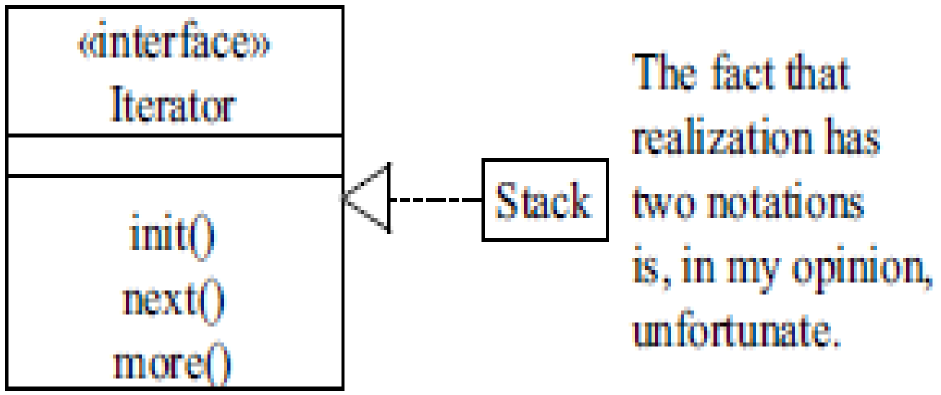
Interfaces UML Notation

- To indicate that a class implements a particular interface, use the “lollipop” notation.
- This is also called “realization”.

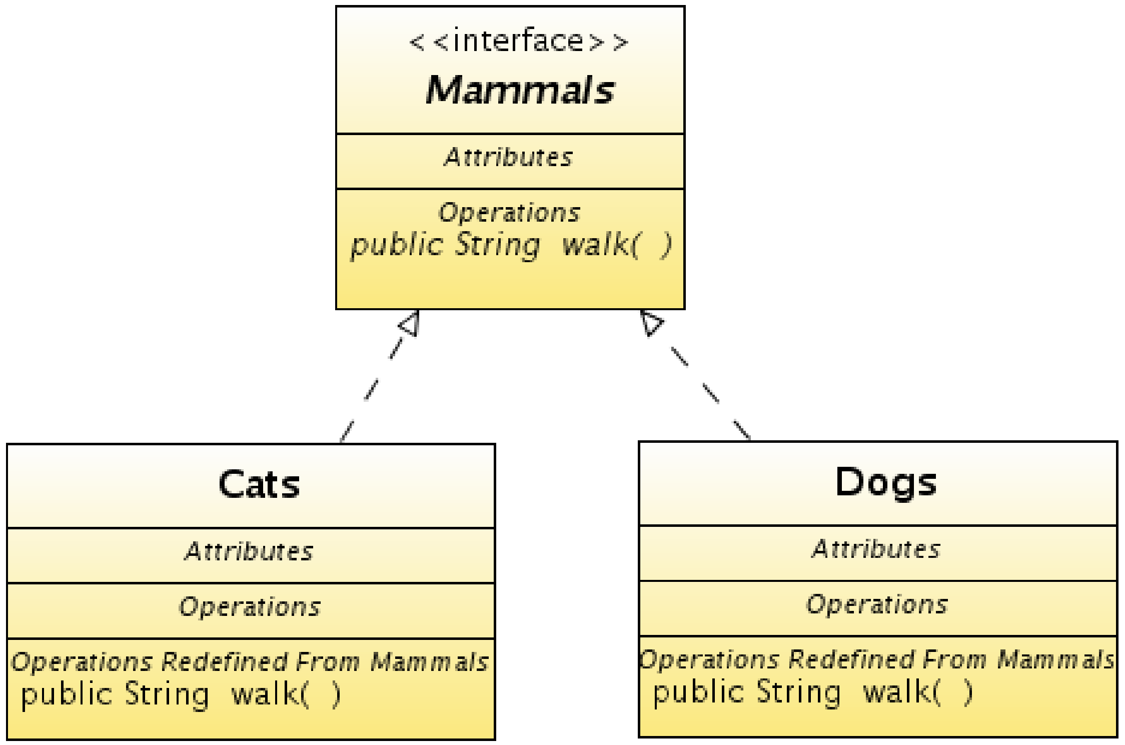


Interfaces UML Notation

- When drawing an interface using a class diagram, realization is shown using the following notation.


«interface»
Iterator
init()
next()
more()
Stack
The fact that realization has two notations is, in my opinion, unfortunate.

Realizations

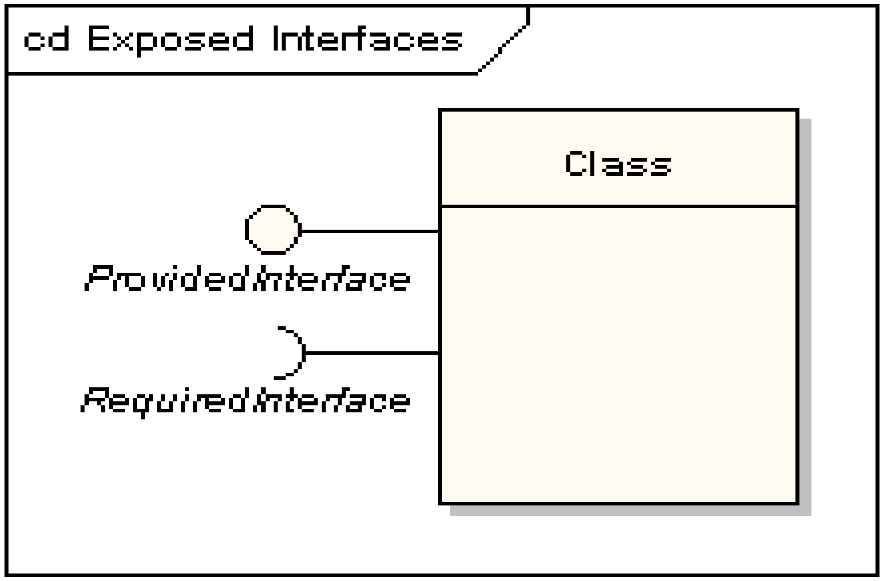


Types of interfaces

- The circle notation does not show the interface operations.
- When interfaces are shown as being owned by classes, they are referred to as exposed interfaces.
- An exposed interface can be defined as either provided or required.
- A provided interface is an affirmation that the containing classifier supplies the operations defined by the named interface element and is defined by drawing a realization link between the class and the interface.
- A required interface is a statement that the classifier is able to communicate with some other classifier which provides operations defined by the named interface element and is defined by drawing a dependency link between the class and the interface.

Types of interfaces

- A provided interface is shown as a "ball on a stick" attached to the edge of a classifier element. A required interface is shown as a "cup on a stick" attached to the edge of a classifier element.



Why an extension mechanism?

- Although UML is very well-defined, there are situations in which it needs to be customized to specific problem domains

- UML extension mechanisms are used to extend UML by:
  - adding new model elements,
  - creating new properties,
  - and specifying new semantics

- There are three extension mechanisms:
  - stereotypes, tagged values, constraints and notes

Stereotypes

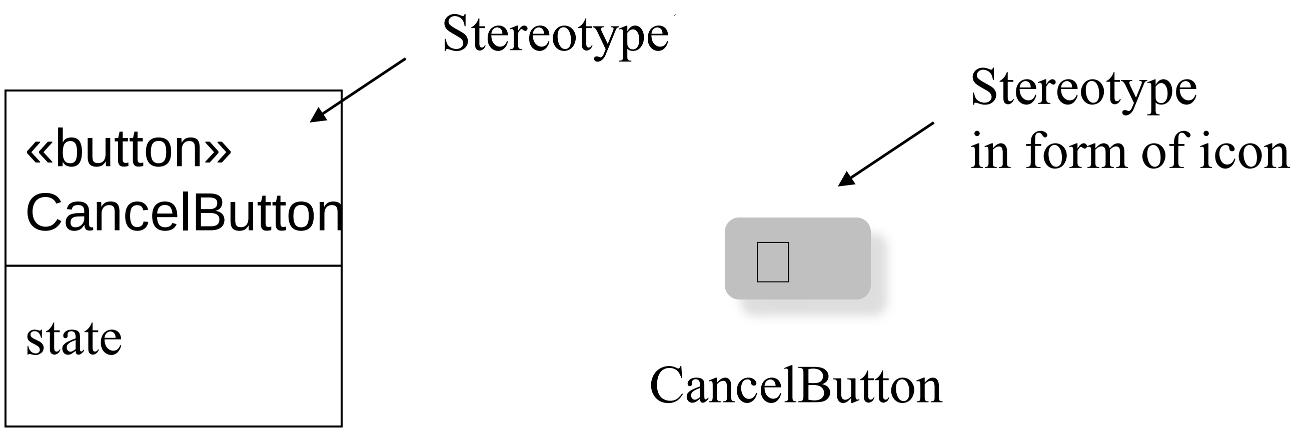
- Stereotypes are used to extend UML to create new model elements that can be used in specific domains

- E.g. when modeling an elevator control system, we may need to represent some classes, states etc. as
  - «hardware»
  - «software»

- Stereotypes should always be applied in a consistent way

Stereotypes (cont.)

- Ways of representing a stereotype:
  - Place the name of the stereotype above the name of an existing UML element (if any)
    - The name of the stereotype needs to be between «» (e.g. «node»)
    - Don’t use double ‘<‘ or ‘>’ symbols, there are special characters called open and close guillemets
  - Create new icons



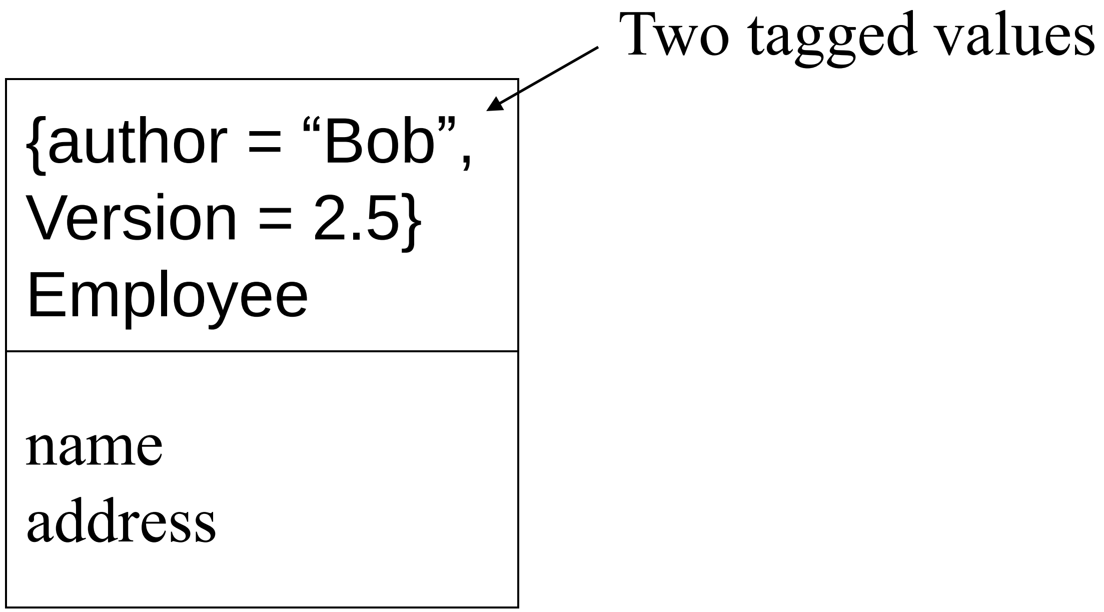
Tagged Values

- Tagged values
  - Define additional properties for any kind of model elements
  - Can be defined for existing model elements and for stereotypes
  - Are shown as a tag-value pair where the tag represent the property and the value represent the value of the property

- Tagged values can be useful for adding properties about
  - code generation
  - version control
  - configuration management
  - Authorship, etc.

Tagged Values (cont.)

- A tagged value is shown as a string that is enclosed by brackets {} and which consists of:
  - the tag, a separator (the symbol =), and a value



The UML Metamodel

- A metamodel is a model representing the structure and semantics of a particular set of models
- A UML model is an instance of the UML metamodel
- The UML metamodel
  - Describes the UML model elements
  - Is defined using a subset of UML
  - Is organized in the form of packages

The UML Metamodel (cont.)

- UML metamodel is defined according to the following concepts:
  - Abstract Syntax: The metamodel of UML is described using UML class diagrams
  - Well-formedness rules: Well-formedness rules are used to express constraints on the model elements
    - -E.g. a class cannot have two names
  - Semantics: describes using the natural language the semantics of the model elements

UML Profiles

- UML Profiles provide an extension mechanism for building UML models for particular domains
  - e.g. real-time systems, web development, etc…

- A profile consists of a package that contains one or more related extension mechanisms (such as stereotypes, tagged values and constraints)
  - that are applied to UML model elements

- Profiles do not extend the UML metamodel. They are also called *the UML light-weight extension mechanism*

UML Profiles (cont.)

- A UML profile is a specification that does one or more of the following:
  - Identifies a subset of the UML metamodel (which may be the entire UML metamodel)
  - Specifies stereotypes and/or tagged values
  - Specifies well-formedness rules beyond those that already exist
  - Specifies semantics expressed in natural language

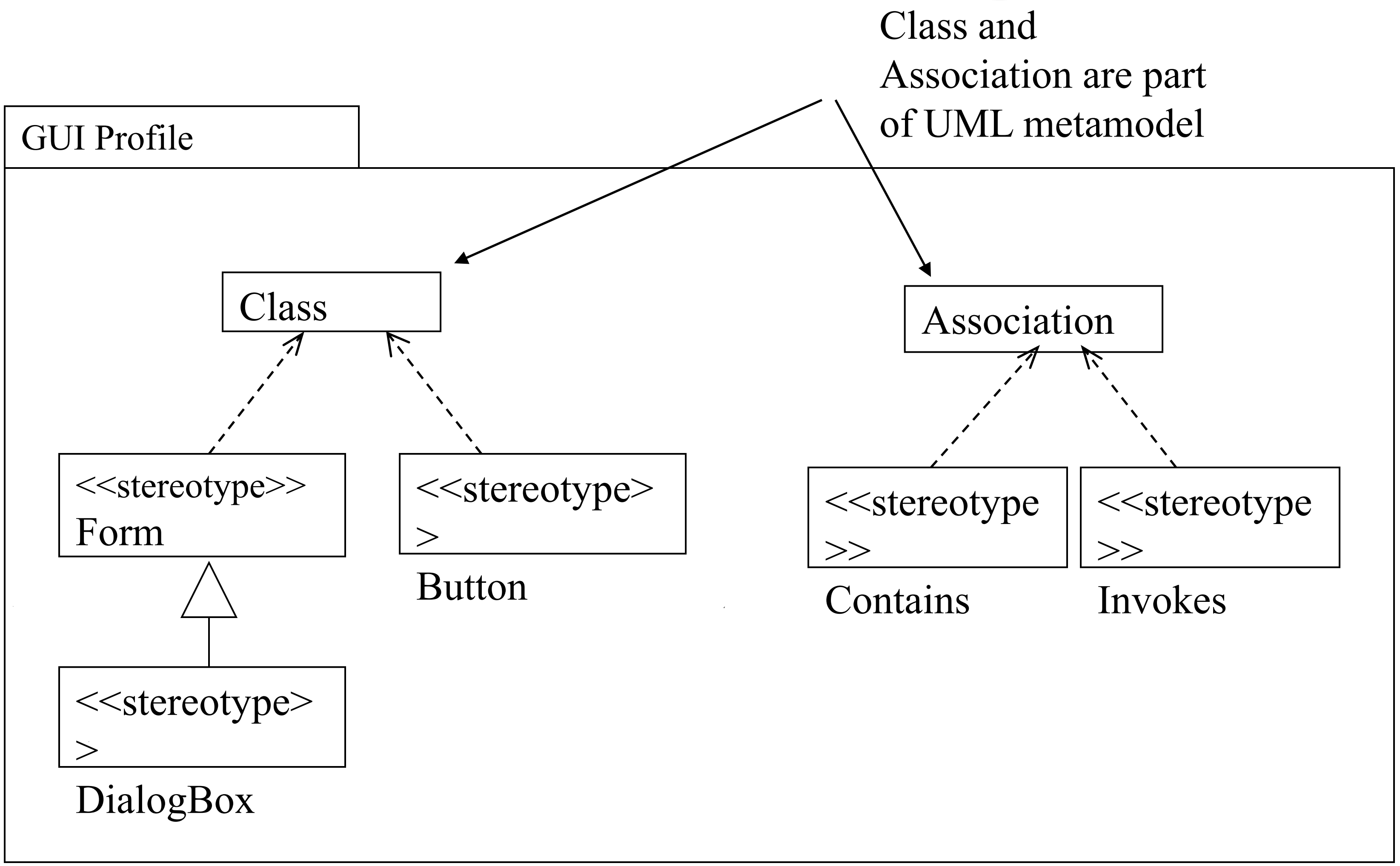
Example of a profile
inspired by the research report of Cabot et al. (2003)

- We would like to create a UML profile for representing basic GUI components.

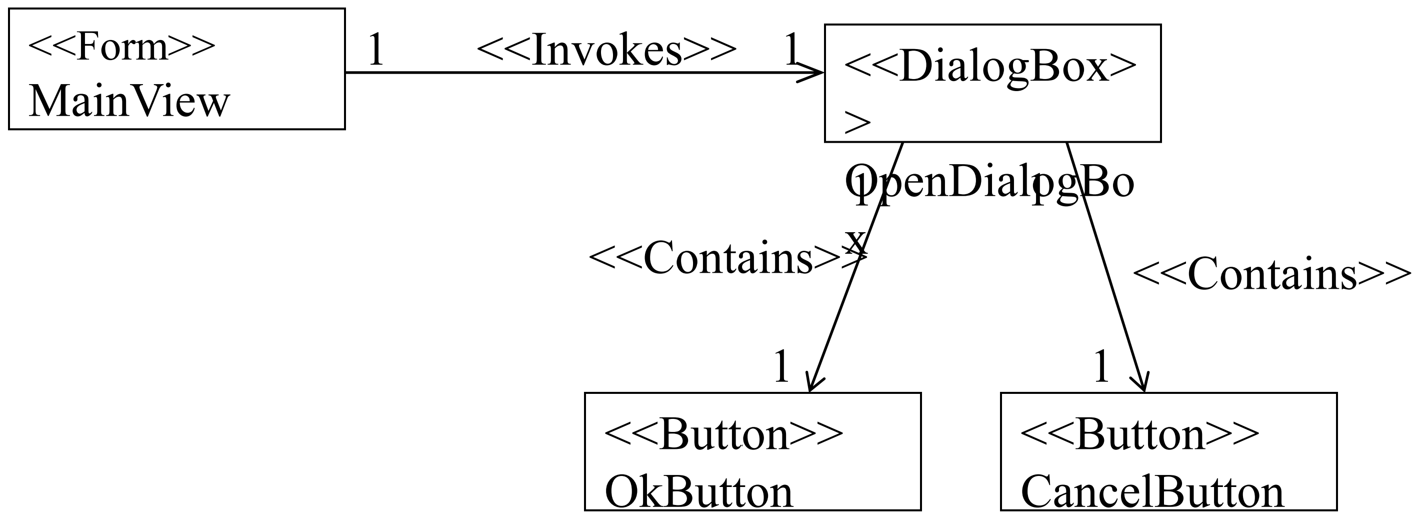
- We suppose that our GUI contains the following components:
  - Forms (which can also be dialog boxes)
  - Buttons

- Constraints: (in practice, we need to be more precise)
  - A form can invoke a dialog box
  - A form as well as a dialog box can contain buttons

The GUI profile package



Instance Diagram of the GUI Profile



Dealing with Complex System - Multiple or Single Class Diagram?

- Inevitably, if you are modeling a large system or a large business area, there will be numerous entities you must consider. Should we use multiple or a single class diagram for modeling the problem? The answer is:
  - Instead of modeling every entity and its relationships on a single class diagram, it is better to use multiple class diagrams.
  - Dividing a system into multiple class diagrams makes the system easier to understand, especially if each diagram is a graphical representation of a specific part of the system.

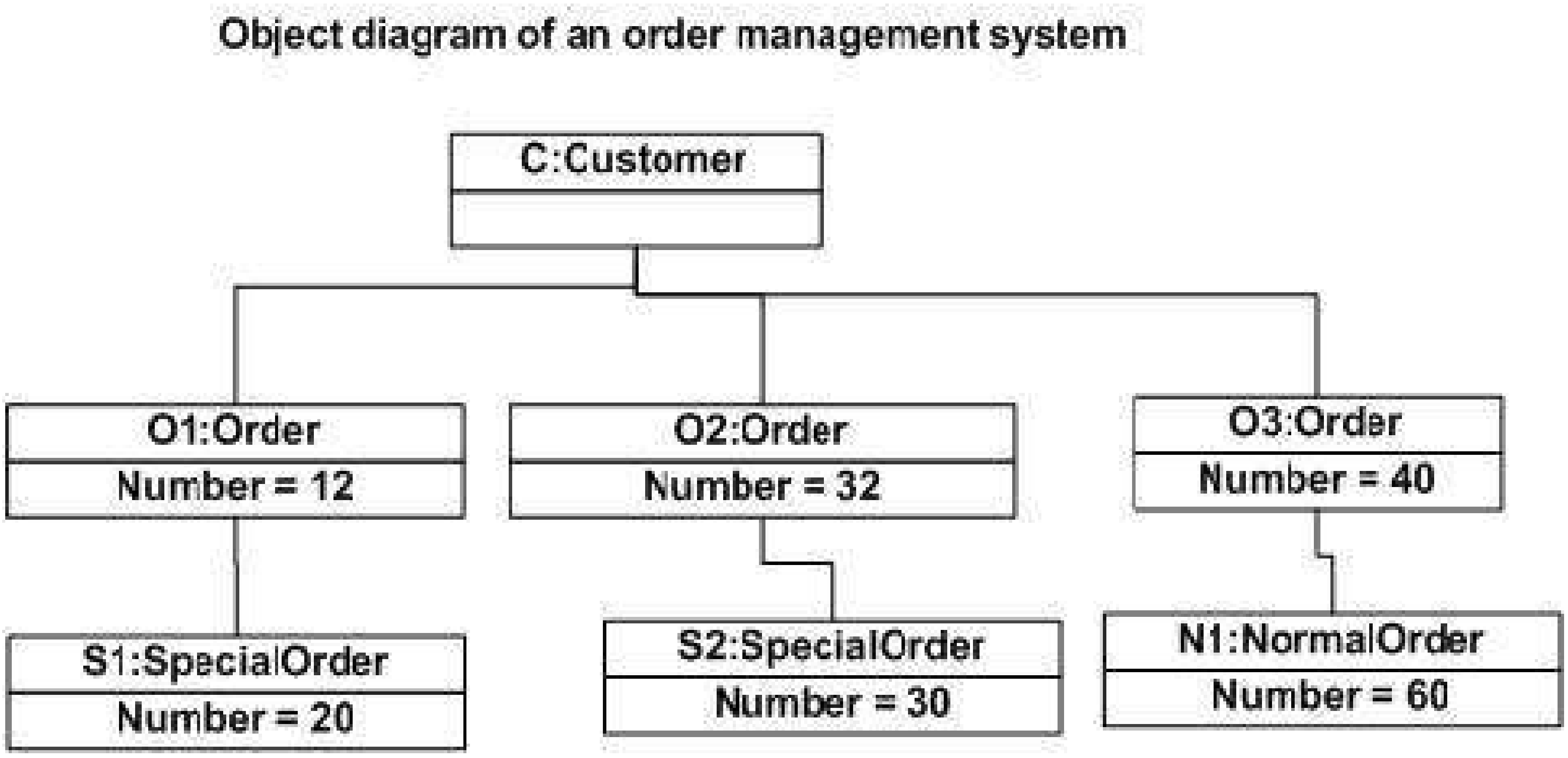
Object Diagram

- Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
- Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
- Object diagrams are used to render a set of objects and their relationships as an instance.

Object Diagram - Purpose

- The purposes of object diagrams are similar to class diagrams.
- The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature.
- It means the object diagram is closer to the actual system behavior. The purpose is to capture the static view of a system at a particular moment.
- The purpose of the object diagram can be summarized as −
  - Forward and reverse engineering.
  - Object relationships of a system
  - Static view of an interaction.
  - Understand object behavior and their relationship from practical perspective

Object Diagram



This image cannot currently be displayed.

Thank you

Questions?