Objects and Classes. Class Stereotypes and Associations.

Definitions

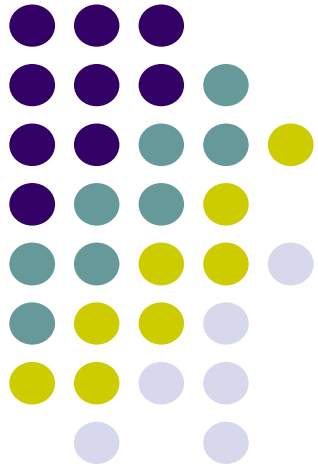
Attributes and Operations

Stereotypes

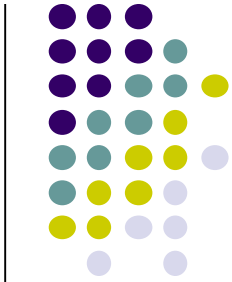
Associations. Multiplicity

Analyses Class Diagrams

Examples



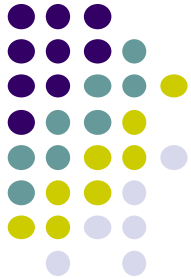
Object-Oriented *Analysis*



- Object-Oriented Analysis (OOA) means ***defining the problem***.
- ***OOA is a process of defining the problem in terms of objects***: real-world objects with which the system must interact, and ***candidate software objects*** used to explore various solution alternatives.
- You can define all of your real-world objects in terms of their classes, attributes, and operations.
- Done before the system design

Source: UML Applied: A .Net Perspective, by Martin Shoemaker (Apress, 2004; ISBN: 1590590872)

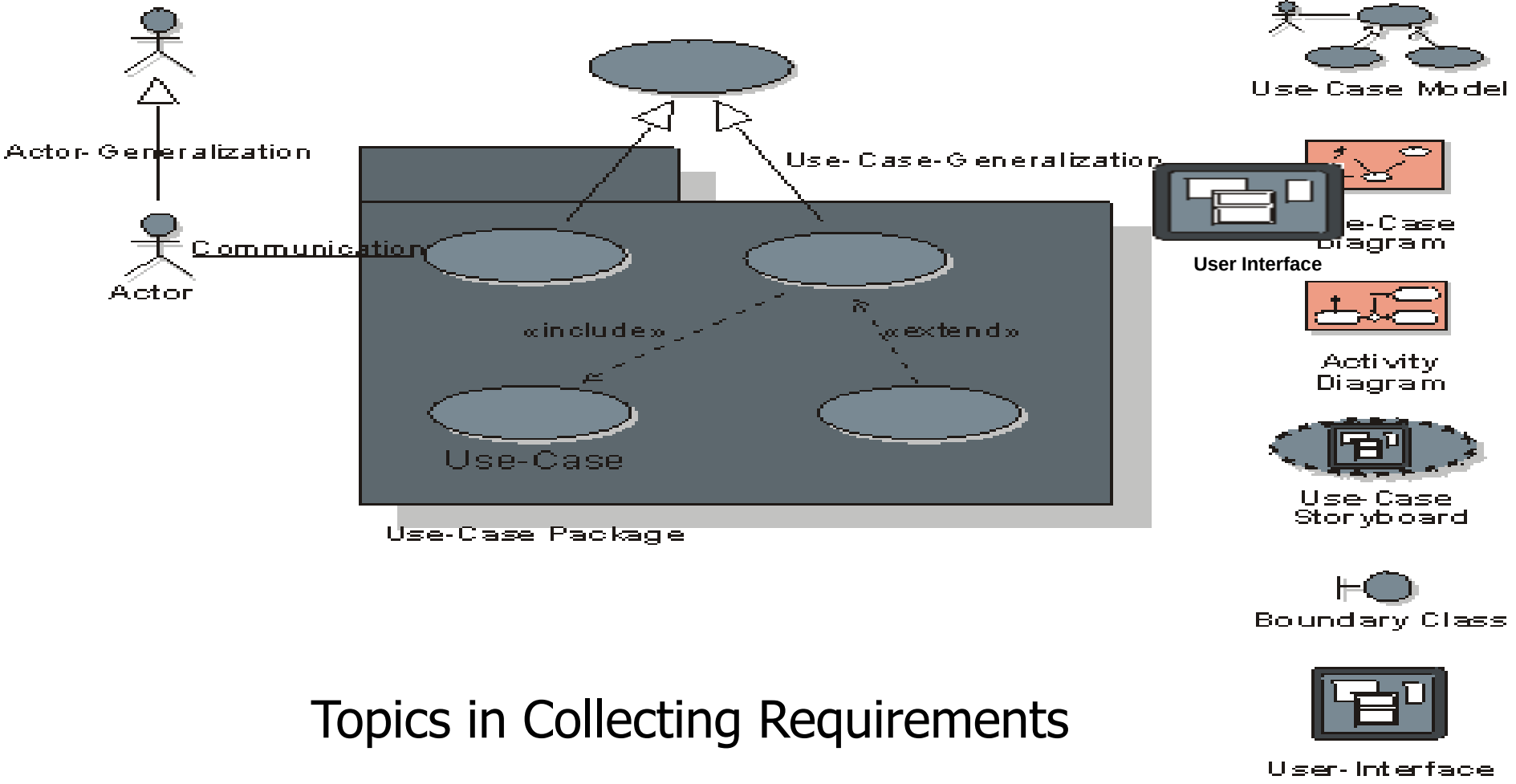
Source: UML Applied: A .Net Perspective, by Martin Shoemaker (Apress, 2004; ISBN: 1590590872)



Object-Oriented *Design*

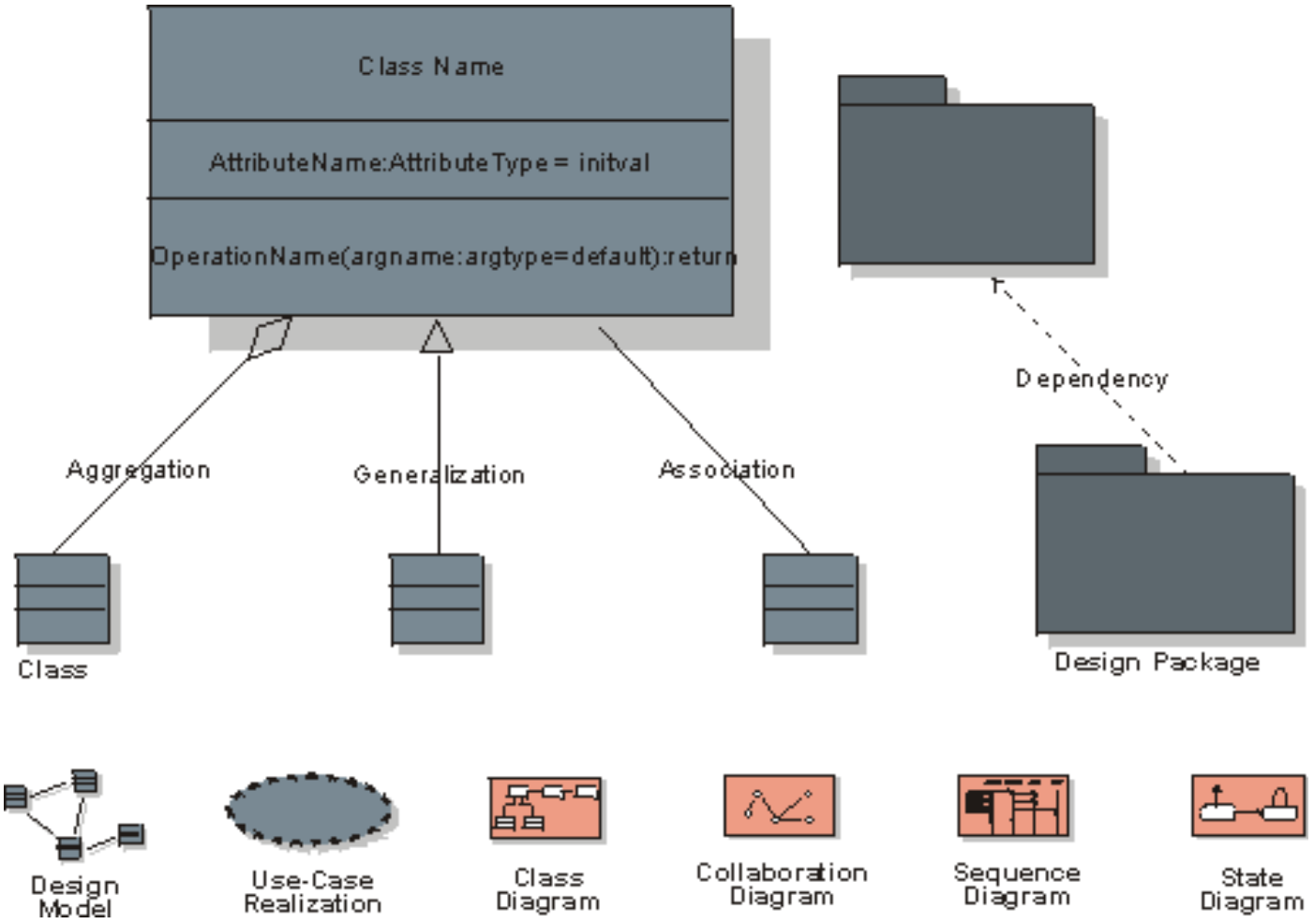
- Object-Oriented Design (OOD) means ***defining the solution***.
- OOD is the process of defining the components, interfaces, objects, classes, attributes, and operations that will satisfy the requirements.
- You typically start with the ***candidate objects*** defined during analysis, and add or change objects as needed ***to refine a solution***.
- Two scales of OOD:
  - architectural design - defining the components
  - component design - defining the classes and interfaces within a component.

From Requirements to Analysis and Design

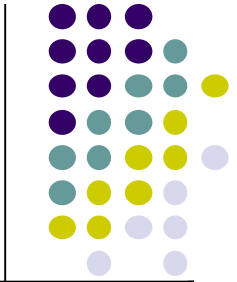


Topics in Collecting Requirements

Topics in Analyses and Design



What are Objects?



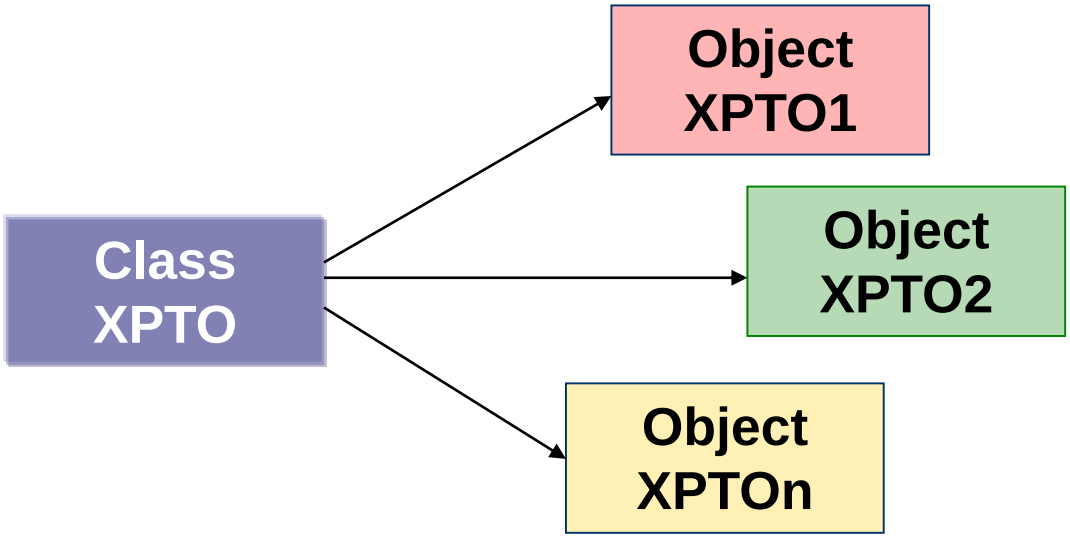
Object:	Representation of an entity, either real-world (concrete like an item) or conceptual (concept as a process). An object is an abstraction with identity, state and behavior.
Identity:	Each object is unique.
State:	A set of the values of object properties at the moment, plus the relationships the object may have with other objects.
Behavior:	Determines how the object responds to request from other objects and is implemented by the set of operations for the objects.

What are Classes?

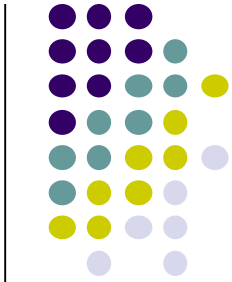
 **Class**	A class is a description of a set (container) of objects with common properties(attributes), behavior (operations), responsibilities, relationships, and semantics
Worker:	Analyst and mainly Designer
Purpose:	Used by designers, implementers, and testes.
Options:	Using any of the «entity», «boundary», and «control» stereotypes is optional.
Reports:	Class report (contains information regarding a specific class within the design model).

Object and Classes

- A class is a template to create objects.
- Each object is an instance of some class.
- The object cannot be instance of more that one class.

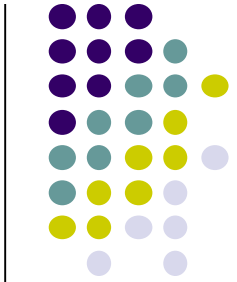


Class Properties



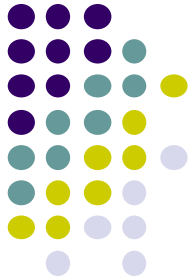
Name	The name of the class.
Brief Description	A brief description of the role and purpose of the class.
Responsi-bilities	The responsibilities defined by the class.

Class Properties – cont.



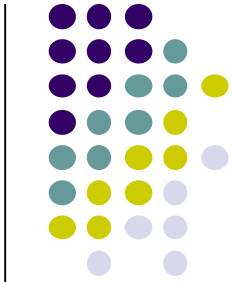
Relation-ships	The relationships, such as generalizations, associations, and aggregations, in which the class participate.
Operations	The operations defined by the class.
Attributes	The attributes defined by the class.

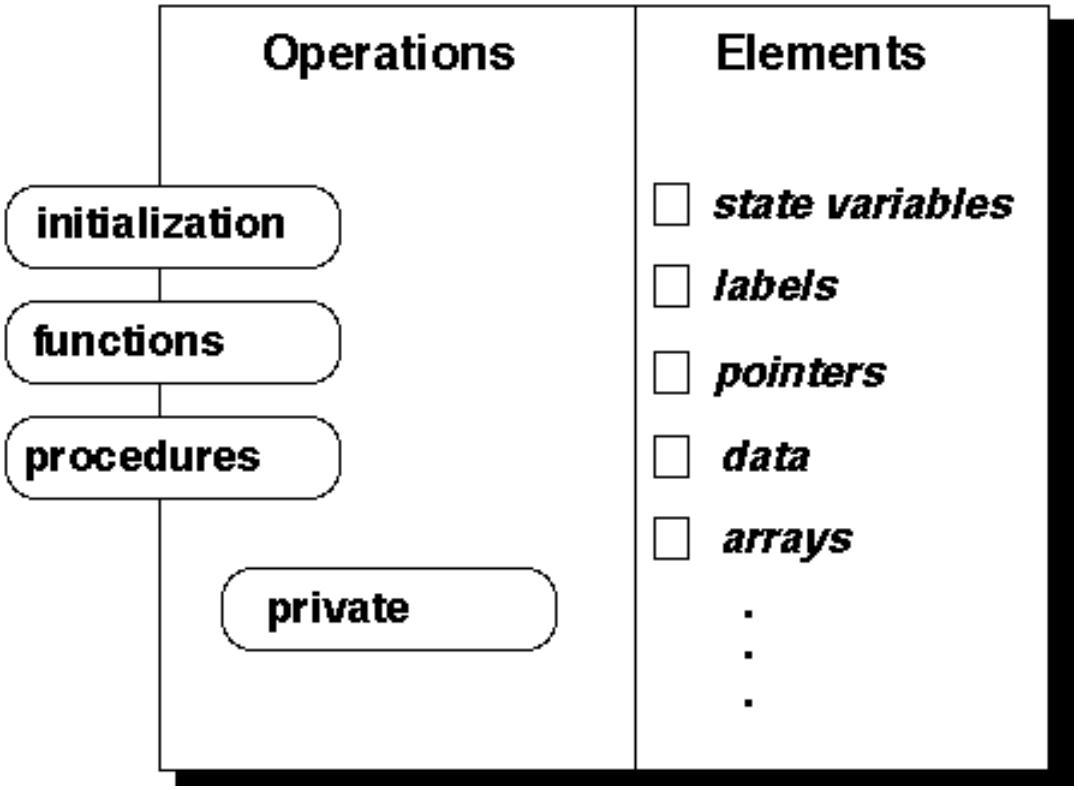
Object-Oriented Programming



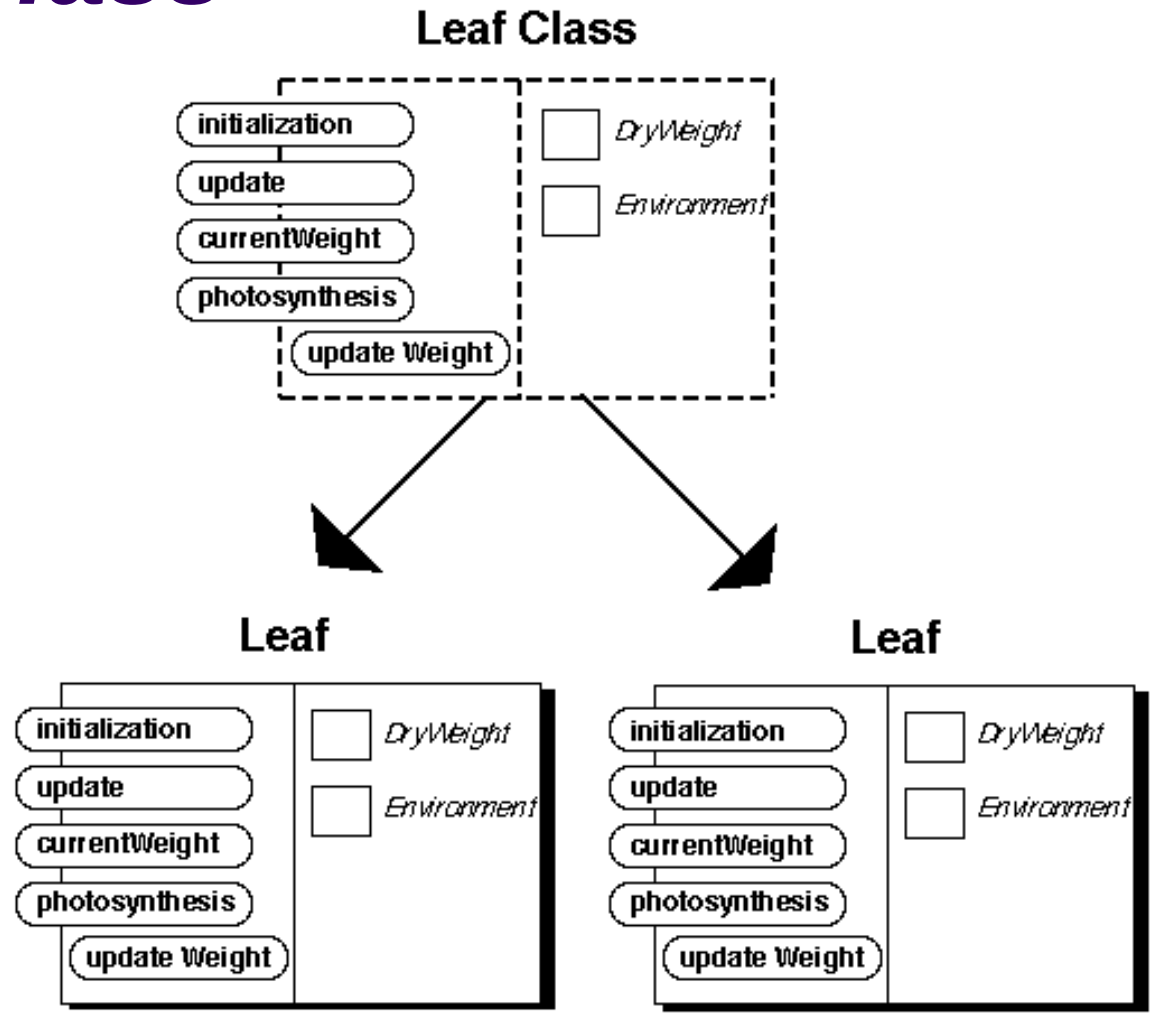
- **Language Level Definition**
  - Class
  - Object
  - Inheritance
- **Conceptual Level Definition**
  - Abstraction
  - Delegation
  - Encapsulation
  - Information Hiding
  - Hierarchy

Language Level Definition of OOP : *Object*

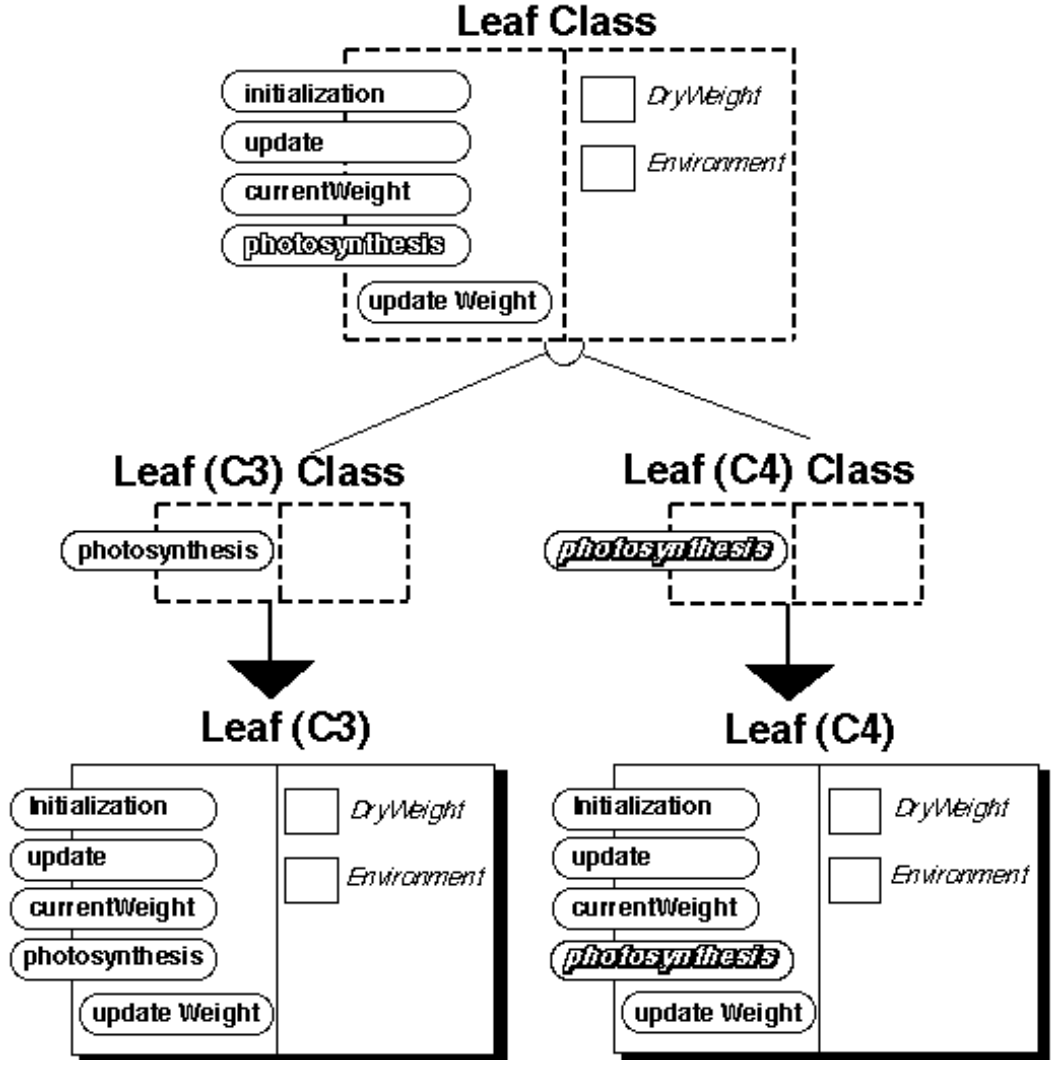




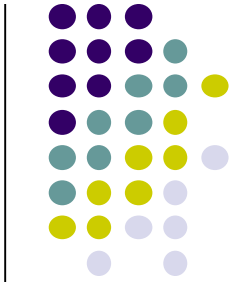
Language Level Definition of OOP : *Class*



Language Level Definition of OOP : *Inheritance*



Sub-typing vs. implementation inheritance



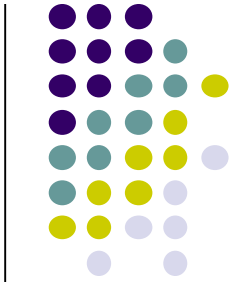
We can distinguish two broad classes of inheritance:

- **Sub-typing**
  - A logical classification of B as a subtype of A. Also called *interface inheritance*. The is-a argument is straightforward. A is *usually* an abstract class (or an interface in Java). All (or almost all) the operations of A are applicable to B. We expect to use Bs in mixed collections of As etc. *We may often inherit some implementation at the same time*.

- **Implementation inheritance**
  - Using the implementation of A for convenience in B. A is often concrete. Only some the operations of the operations of A are applicable to B. *We probably won't use Bs in place of As*.

- Sub-typing is almost always OK, provided the classification is valid. Implementation inheritance is controversial.

Conceptual Level Definition of OOP: *Abstraction*

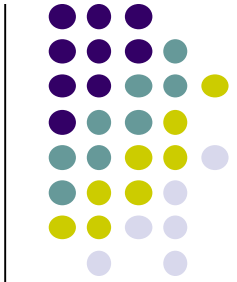


- "Extracting the essential details about an item or group of items, while ignoring the unessential details."
  Edward Berard
- "The process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use."
  Richard Gabriel

Example

- Pattern: Priority queue
- Essential Details:
  - length
  - items in queue
  - operations to add/remove/find item
- Variation: link list vs. array implementation; stack, queue

Conceptual Level Definition of OOP: *Delegation*

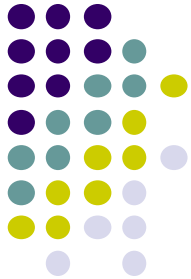


Three ways of reusing classes:

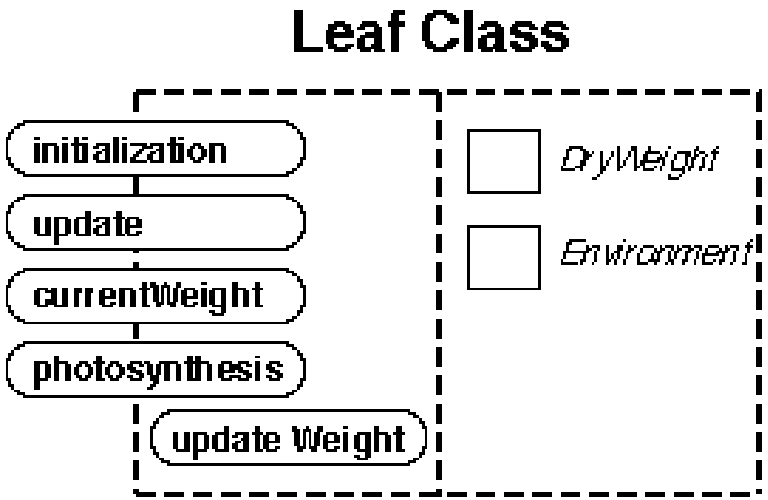
- Generalize A, usually by *parameterization*, so that it does the what you want as well as what it originally did.
- *Create a new class B which **delegates** operations to A, usually by having an object of class A as an instance variable and calling methods on that object. B is then a client of A.*
- Create a new class B which *inherits* from A. B is then a *subclass* of A.

- Open discussion – *pro's and con's?*

Conceptual Level Definition of OOP: *Encapsulation, Information Hiding*



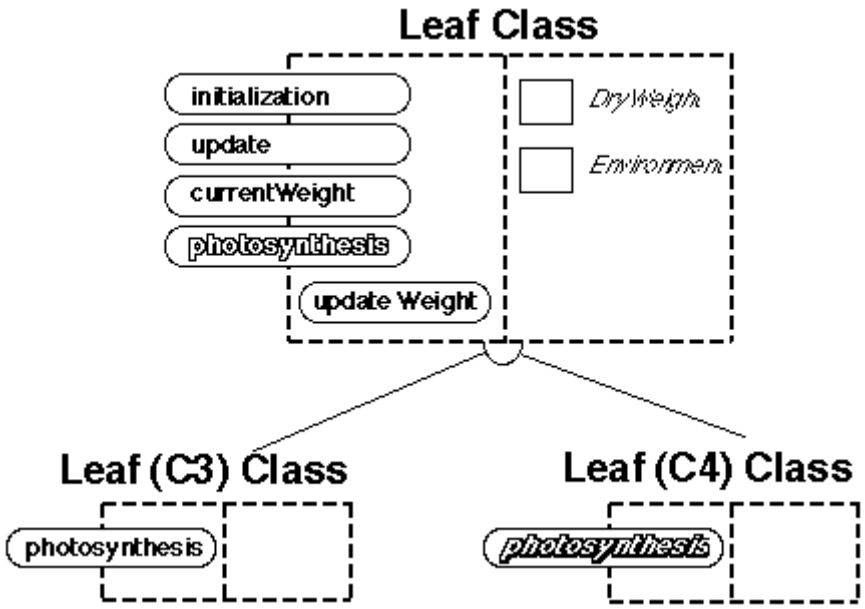
- ***Encapsulation*** - Enclosing all parts of an abstraction within a *class container*
- ***Information Hiding*** - Hiding parts of the abstraction within an *object*
- **Example**



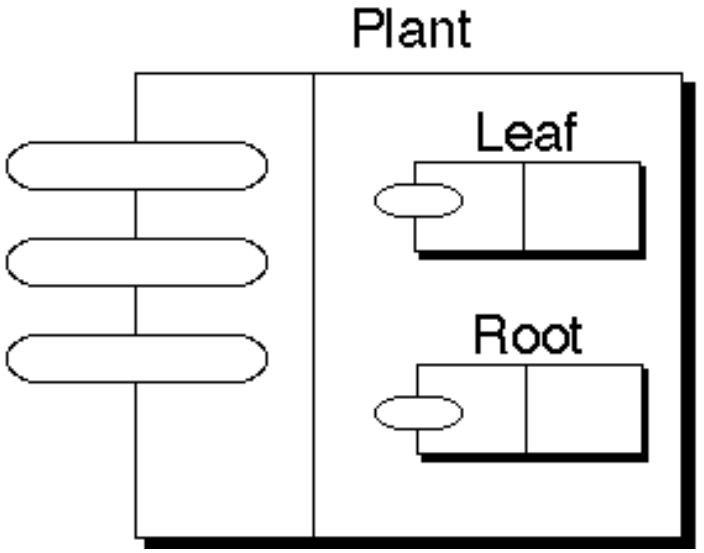
Conceptual Level Definition of OOP: *Hierarchy*

- *Abstractions arranged in order of rank or level*

Class Hierarchy



Object Hierarchy



From Requirements through Analyses to Design [Bruegge&Dutoit, 2004]

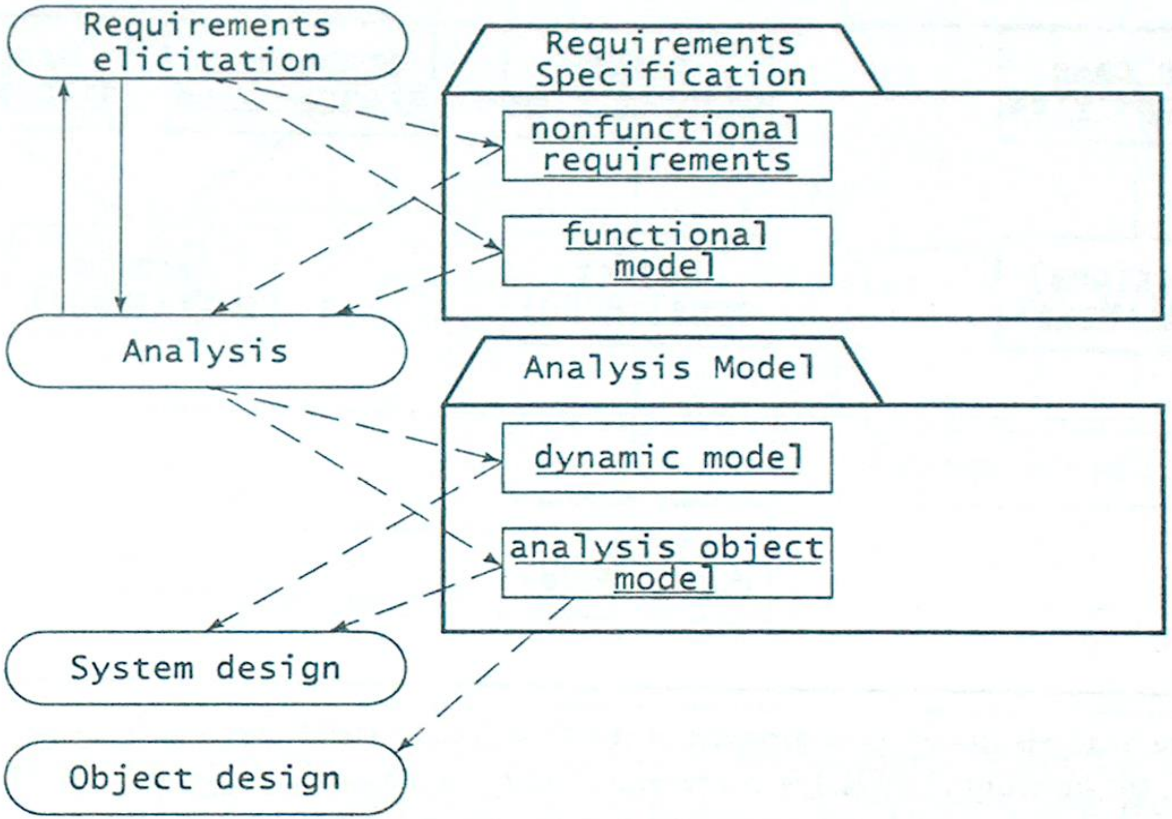
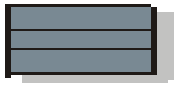


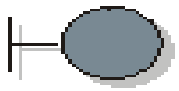
Figure 5-2 Products of requirements elicitation and analysis (UML activity diagram).

The Analysis Classes and Their Stereotypes

 **Analysis Class**	Represent an early conceptual model for 'things in the system which have responsibilities and behavior'. They eventually evolve into ***classes*** and ***subsystems*** in the Design Model.

- Analysis classes may be *stereotyped* as one of the following:
  - **Boundary** classes
  - **Control** classes
  - **Entity** classes
- Stereotyping results in a robust object model because changes to the model tend to affect only a specific area. Changes in the user interface, for example, will affect only boundary classes, etc.

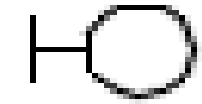
Boundary Classes

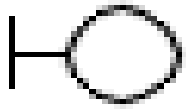
 **Boundary Class**	A **boundary class** models the interaction between the system and its surroundings (one or more actors).

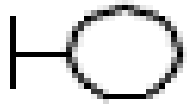
Boundary classes capture the UI requirements:

- coordinating the actor's behavior with the "internals" of the system (primary windows);
- receiving input from the actor to the system, e.g., information or requests;
- providing output from the system to the actor, e.g., stored information or derived results.

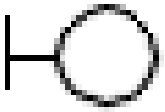
Boundary Classes - Examples

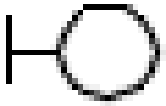


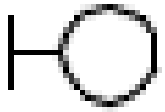


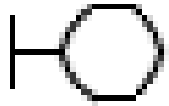


Boundary classes for a mail application



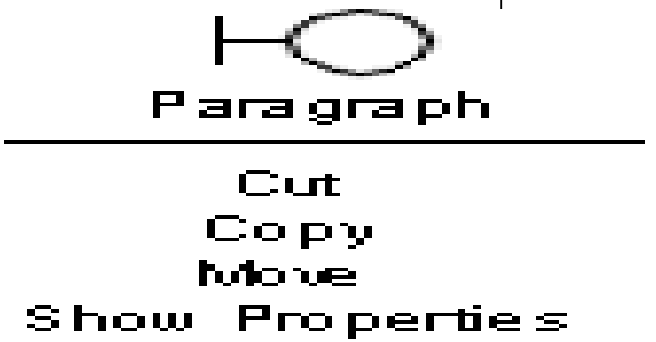


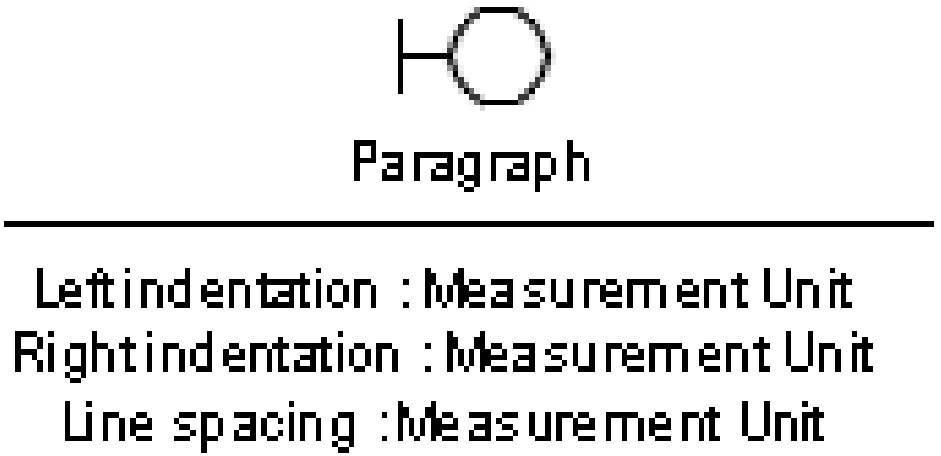




Boundary classes for a document editor

Responsibilities and Attributes





Use case name	ReportEmergency
Participating actors	Initiated by FieldOfficer Communicates with Dispatcher
Flow of events	1. The FieldOfficer activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the FieldOfficer. 3. The FieldOfficer fills out the form by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form. 4. FRIEND receives the form and notifies the Dispatcher. 5. The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the report. 6. FRIEND displays the acknowledgment and the selected response to the FieldOfficer.
Entry condition	• The FieldOfficer is logged into FRIEND.
Exit condition	• The FieldOfficer has received an acknowledgment and the selected response from the Dispatcher, OR • The FieldOfficer has received an explanation indicating why the transaction could not be processed.
Quality requirements	• The FieldOfficer's report is acknowledged within 30 seconds. • The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.

Figure 2-14 An example of a use case, ReportEmergency.

Boundary objects for *ReportEmergency* use case

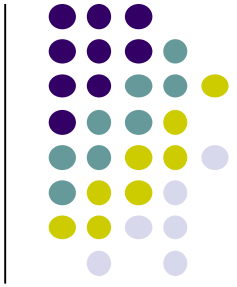
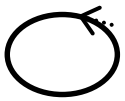
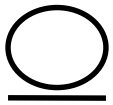


Table 5-3 Boundary objects for the ReportEmergency use case.

AcknowledgmentNotice	Notice used for displaying the Dispatcher's acknowledgment to the FieldOfficer.
DispatcherStation	Computer used by the Dispatcher.
ReportEmergencyButton	Button used by a FieldOfficer to initiate the ReportEmergency use case.
EmergencyReportForm	Form used for the input of the ReportEmergency. This form is presented to the FieldOfficer on the FieldOfficerStation when the "Report Emergency" function is selected. The EmergencyReportForm contains fields for specifying all attributes of an emergency report and a button (or other control) for submitting the completed form.
FieldOfficerStation	Mobile computer used by the FieldOfficer.
IncidentForm	Form used for the creation of Incidents. This form is presented to the Dispatcher on the DispatcherStation when the EmergencyReport is received. The Dispatcher also uses this form to allocate resources and to acknowledge the FieldOfficer's report.

Control and Entity Classes

 **Control Class**	Used to model control behavior specific to one or a few use cases. Control classes encapsulate use-case specific behavior.
 **Entity Class**	Used to model information and associated behavior that must be stored. Entity objects hold and update information about some phenomenon, such as an event, a person, etc.

Entity objects for *ReportEmergency* use case

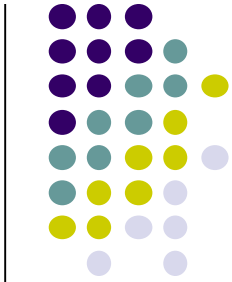
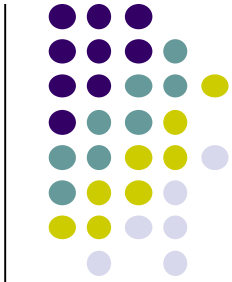


Table 5-2 Entity objects for the ReportEmergency use case.

Dispatcher	Police officer who manages Incidents. A Dispatcher opens, documents, and closes Incidents in response to Emergency Reports and other communication with FieldOfficers. Dispatchers are identified by badge numbers.
EmergencyReport	Initial report about an Incident from a FieldOfficer to a Dispatcher. An EmergencyReport usually triggers the creation of an Incident by the Dispatcher. An EmergencyReport is composed of an emergency level, a type (fire, road accident, other), a location, and a description.
FieldOfficer	Police or fire officer on duty. A FieldOfficer can be allocated to, at most, one Incident at a time. FieldOfficers are identified by badge numbers.
Incident	Situation requiring attention from a FieldOfficer. An Incident may be reported in the system by a FieldOfficer or anybody else external to the system. An Incident is composed of a description, a response, a status (open, closed, documented), a location, and a number of FieldOfficers.

Control and Entity Classes - Examples



 **Control Class**	It is your turn – give examples here!

 **Entity Class**	Examples?

Examples [Bruegge&Dutoit, 2004]

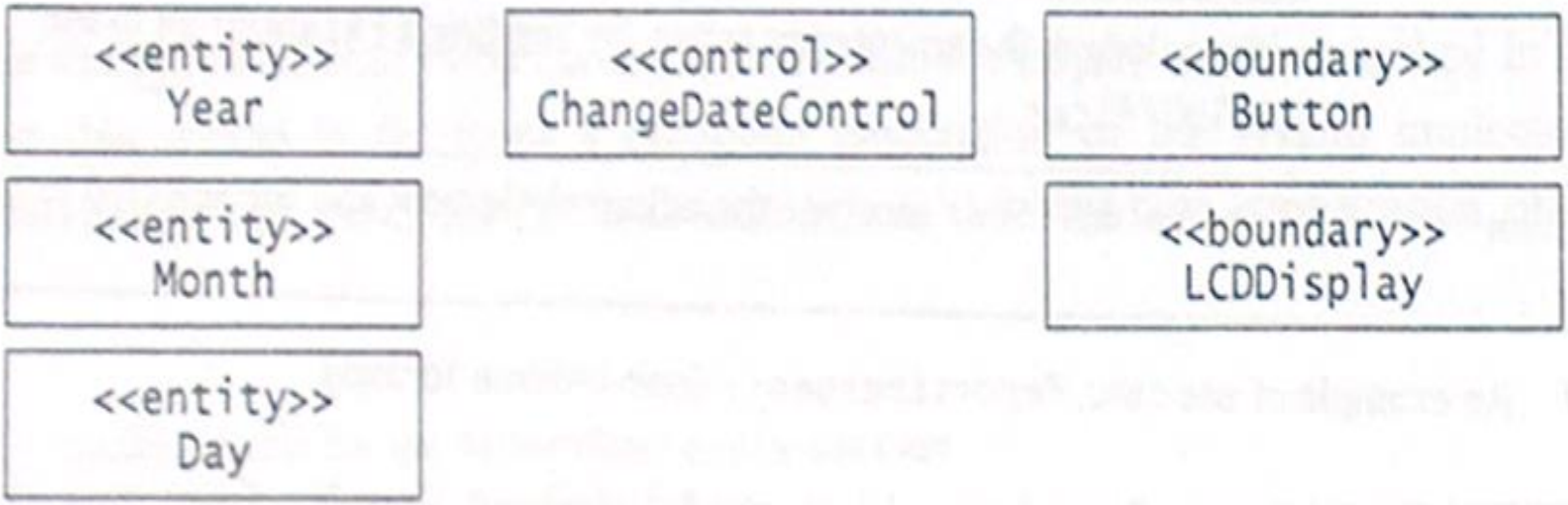
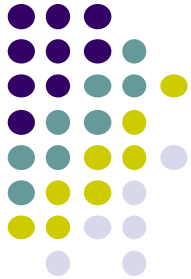


Figure 5-5 Analysis classes for the 28watch example.

Analyses Activities (Use cases -> Objects)



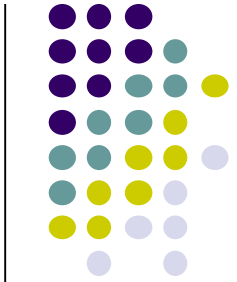
- Find entity objects
- Find boundary objects
- Find control objects
- Map use cases to objects with sequence diagrams
- Model interaction among objects with CRC (Class-Responsibilities-Collaborations) cards
- Find associations and aggregations
- Identify attributes
- Model state-dependent behaviour of individual objects
- Build inheritance relationships
- Do a review of the analyses model

Drawing CRC card diagram

- Class-Responsibility Collaborator (CRC) card visualize classes in card-like presentation.
- Each CRC card contains information like the description of class, its attributes and responsibility.
- Homework: read https://www.visual-paradigm.com/support/documents/vpuserguide/94/1289/6518_drawingcrcca.html

Class Name	
Responsibilities	**Collaborations**
(what the class does)	(related objects)

What are analyses objects and what are not?



- The analyses object model and its dynamic model represent user level concepts *but not actual software classes and components*
- The analyses model should be *safe from system classes and components*
- Analyses classes are still *high-level abstractions* and will be realized in details later
- Analyses classes make a natural transition *from user requirements → through understanding the system → to system design*

Good and bad examples of analyses objects [Bruegge&Dutoit, 2004]

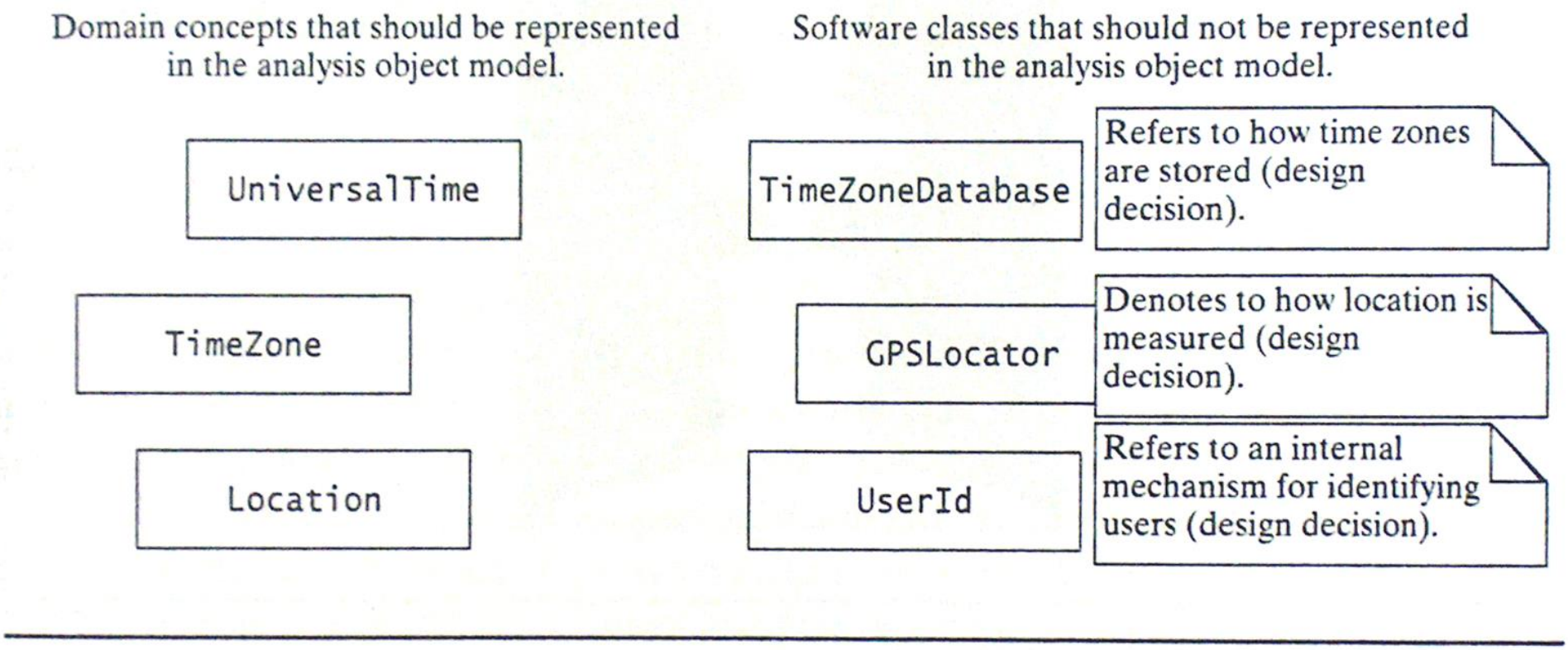
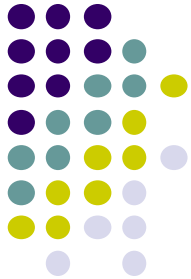


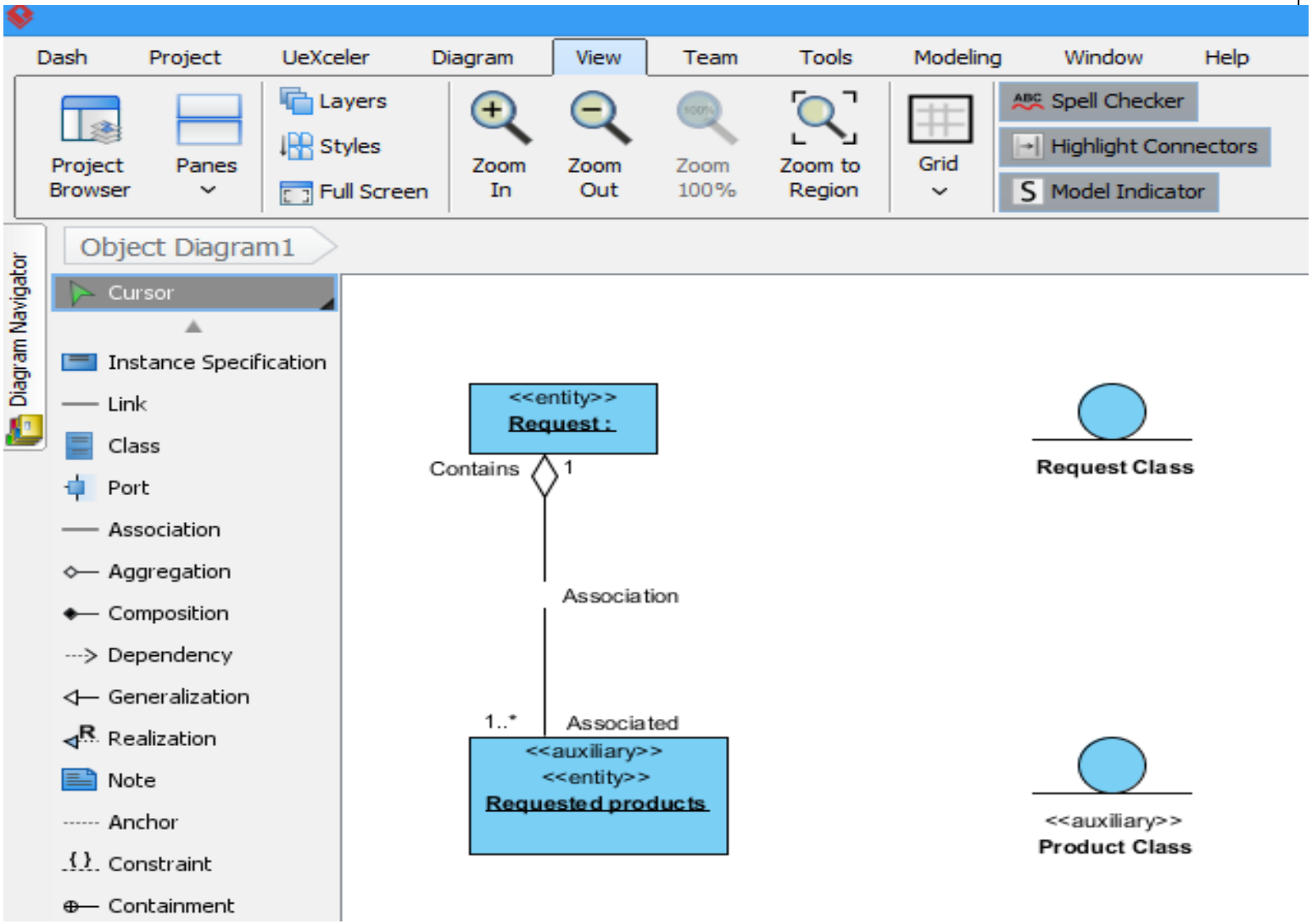
Figure 5-4 Examples and counterexamples of classes in the analysis object model of SatWatch.

UML Object Diagrams 1/3

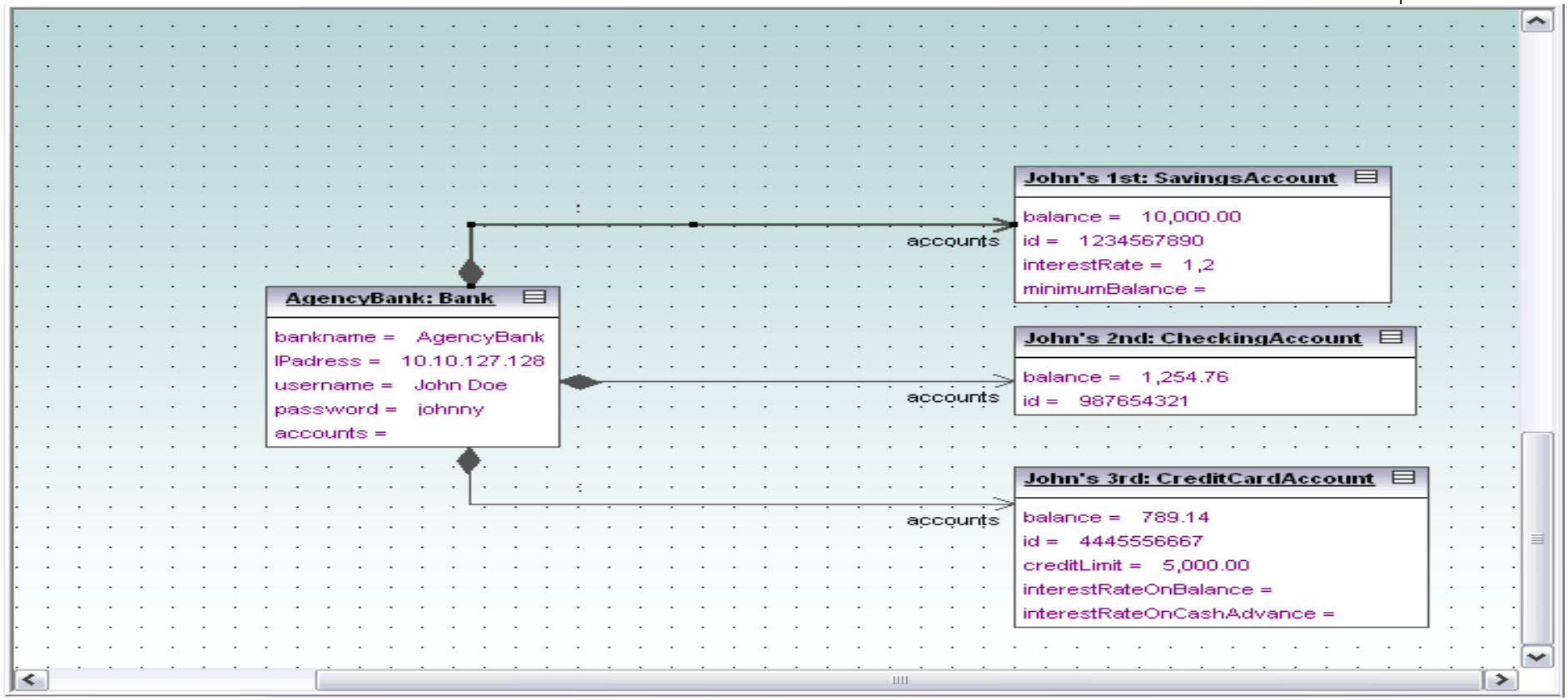


- Object diagram shows *a snapshot of instances of things in class diagrams*.
- UML object diagrams are used to illustrate *an instance of a class at a particular point in time* - a real-life example of a class and its relationships.
- Object diagrams can help clarify classes and inheritance, i.e. *may assist non-programming stakeholders who may find class diagrams too abstract*.

UML Object Diagrams 2/3



UML Object Diagrams 3/3

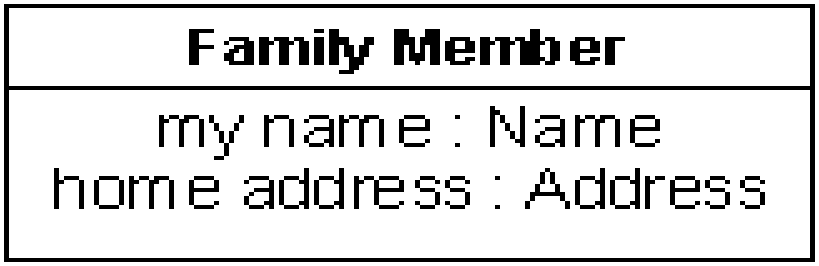


- Homework: https://www.visual-paradigm.com/support/documents/vpuserguide/94/2584/7191_drawingobjec.html

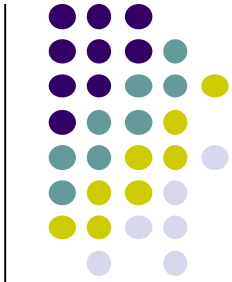
Defining Design Classes

 **Design Class**	A **design class** is a description of a set of objects that share the same responsibilities, relationships, operations, attributes, and semantics.

Example:



The Attribute Notation

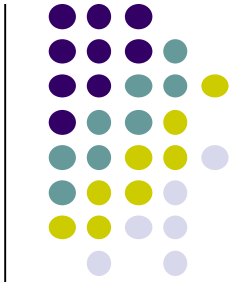


- *visibility / name* : *type multiplicity = default* {*property strings and constraints*}

- visibility ::= {+|-|#|~}

- multiplicity ::= [*lower*..*upper*]

Visibility of Attributes



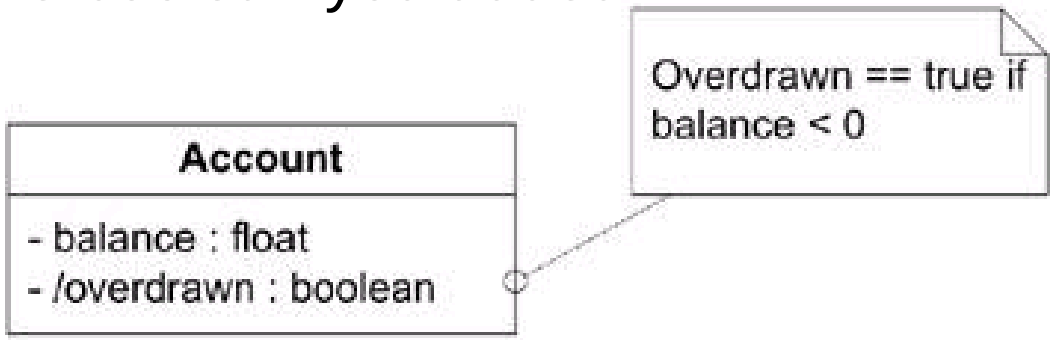
Attribute visibility can be: **visibility ::= { + | - | # | ~ }**

- **Public**: the attribute is visible both inside and outside the package containing the class.
- **Private**: the attribute is only visible to the class itself and to **friends** of the class
- **Protected**: the attribute is visible only to the class itself, to its subclasses, or to **friends** of the class (language dependent)
- **Package:** only classes within the same package as the container can see and use the classes.

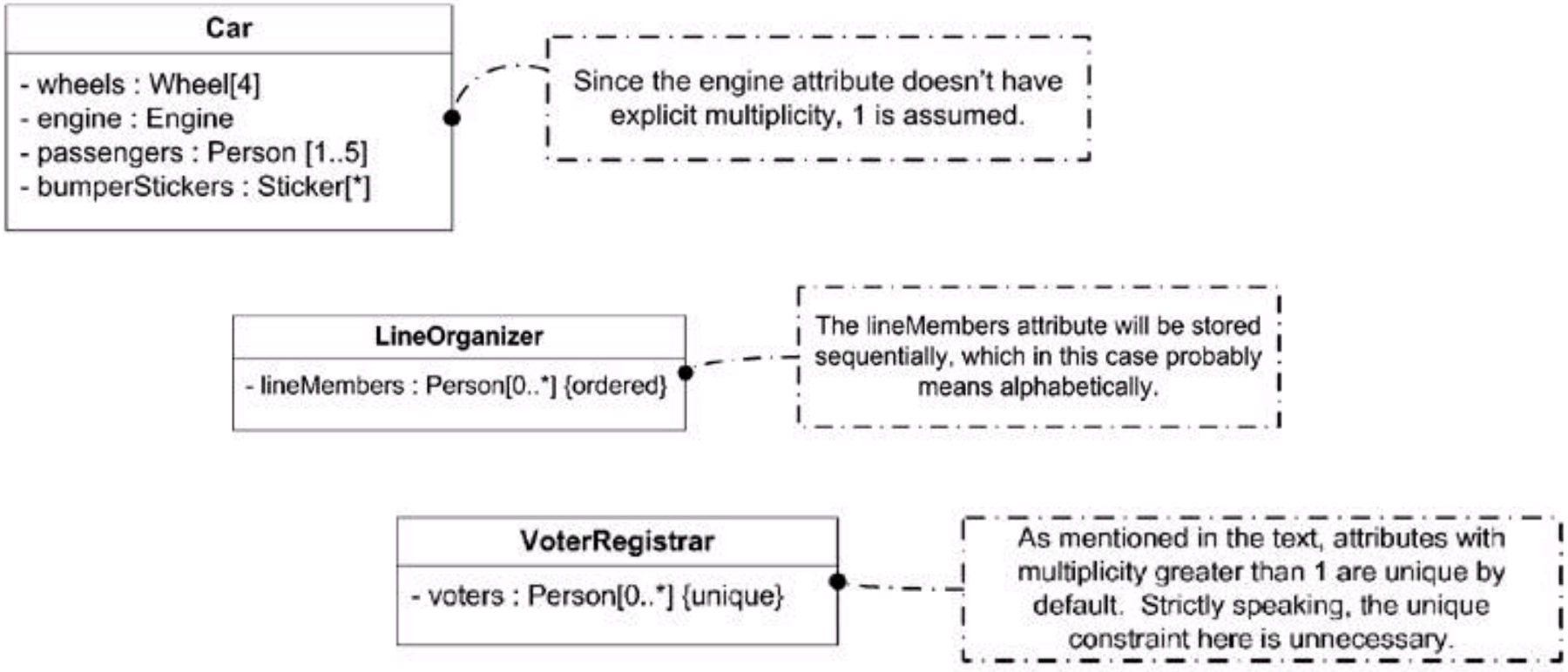
Derived Attributes

(source: O'Reilly "UML 2.0 in a Nutshell")

The derived notation, which is the leading forward slash (/), can be used as an indicator to the implementer that *the attribute may not be strictly necessary*. For example, let's say you modeled a bank account with a simple class named Account. This class stores the current balance as a floating-point number named balance. To keep track of whether this account is overdrawn, you add a boolean named overdrawn. Whether the account is overdrawn is really based on whether the balance is positive, not the boolean you added.



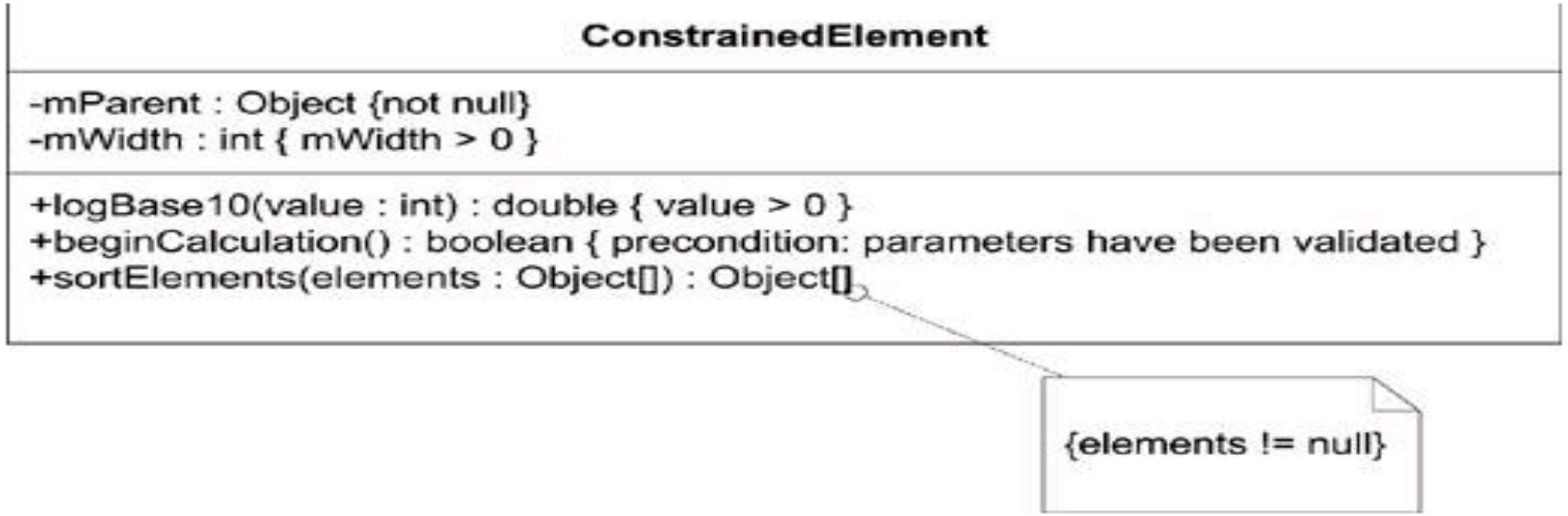
Multiplicity, Ordering, Uniqueness
(source: O'Reilly "UML 2.0 in a Nutshell")



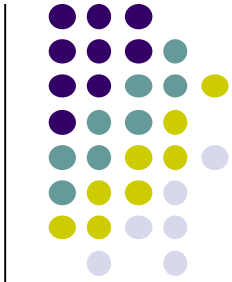
Constraints

(source: O'Reilly "UML 2.0 in a Nutshell")

Constraints represent restrictions placed on an element. They may be ***natural language*** or use a formal grammar such as the ***OCL***; however, *they must evaluate to a Boolean expression*. You typically show constraints between curly braces ({…}) after the element they restrict, though they may be placed in a note and linked to the element using a dashed line.



Operations



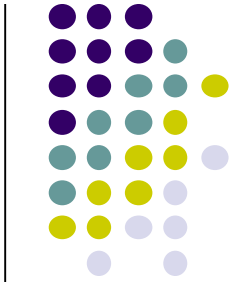
We place operations in a separate compartment with the following syntax:

- *visibility name (parameters) : return-type {properties}*

where parameters are written as:

- *direction parameter_name : type [multiplicity] = default_value { properties }*

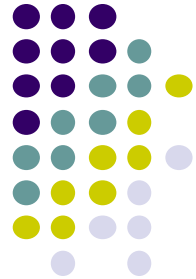
More about op's - Direction



Direction (optional) - indicates how a parameter is used by an operation. It is one of **in**, **inout**, **out**, or **return**:

- **in** states that the parameter is passed to the operation by the caller;
- **inout** states that the parameter is passed by the caller and is then *possibly modified* by the operation and passed back out.
- **out** states that the parameter isn't set by the caller but is modified by the operation and is passed back out.
- **return** indicates that *the value set by the caller is passed back out as a return value*.

More about op's - Properties

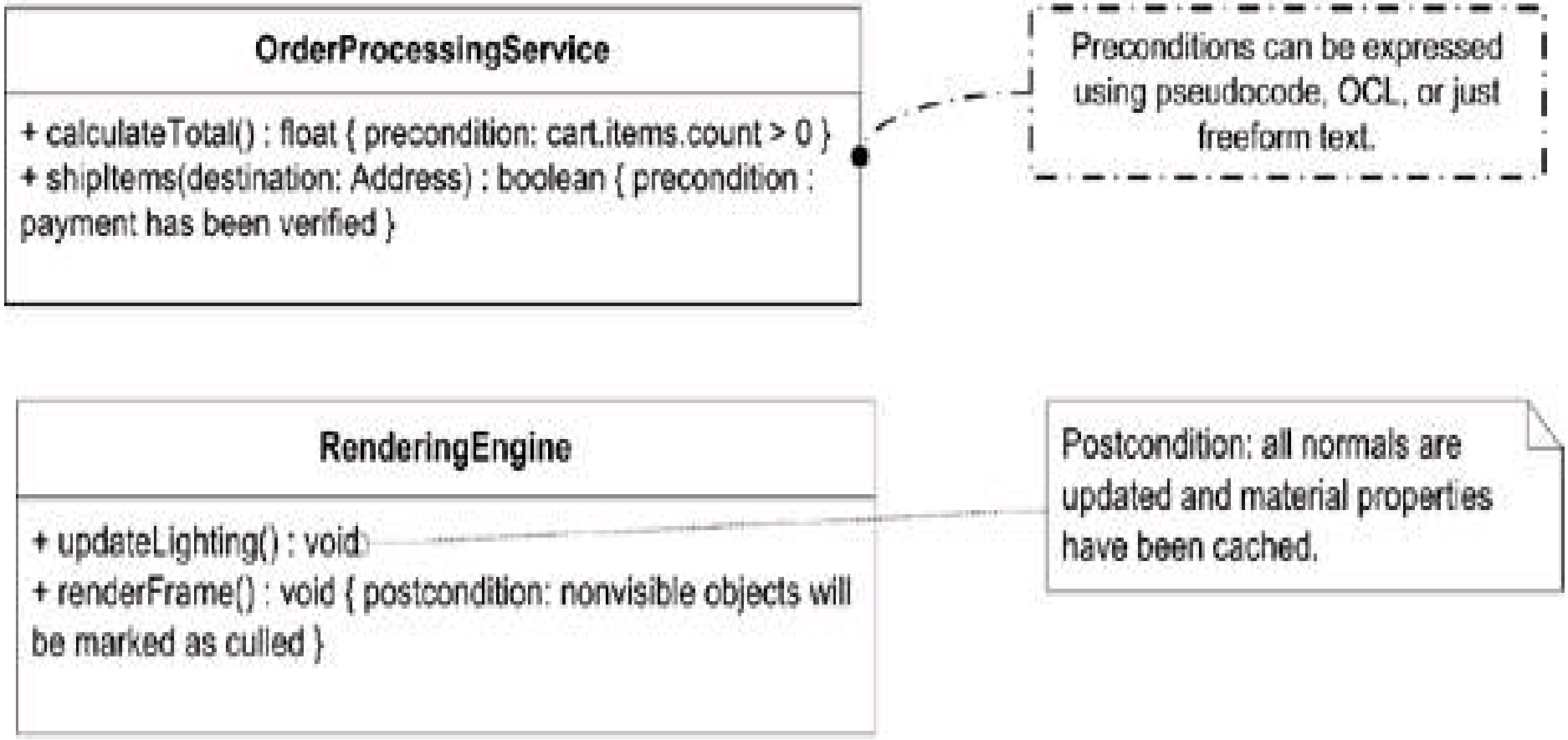


- *Properties* - specifie any parameter-related properties and is specified between curly braces.

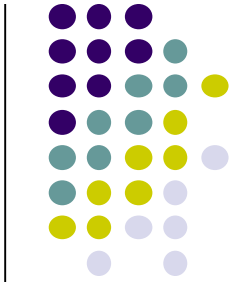
- These are typically defined within the context of a specific model, with a few exceptions:
  - **ordered**,
  - **readOnly**, and
  - **unique**.

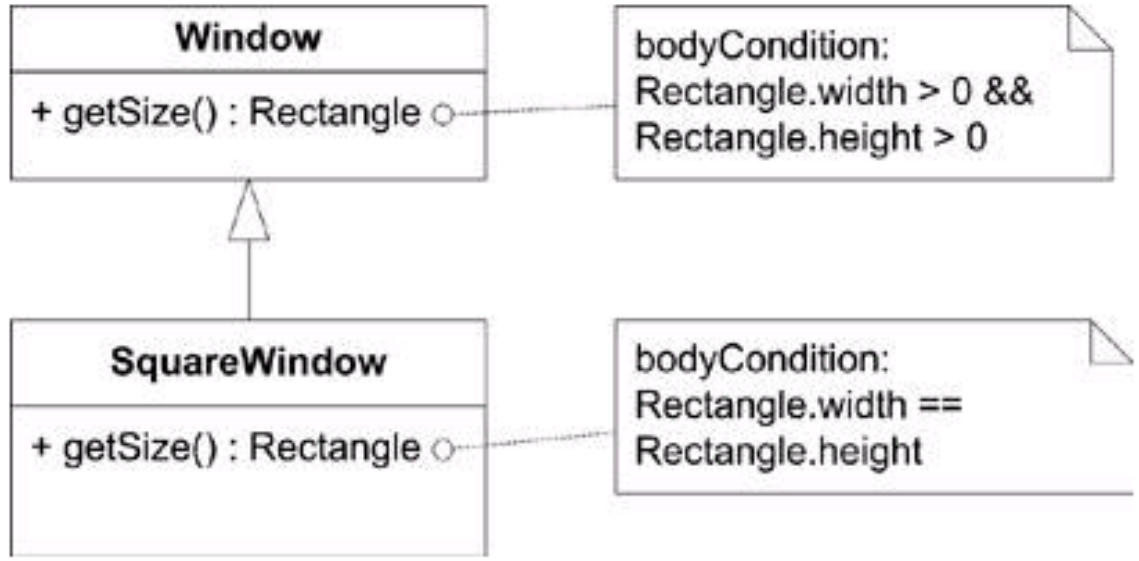
Pre- and Post-condition Constraints

(source: O'Reilly "UML 2.0 in a Nutshell")



Body Conditions & Query Operations





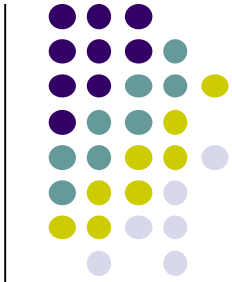
Associations

Association	An **association** models a bi-directional semantic connection among instances.

Associations represent *structural relationships* between objects of different classes; it connects instances of two or more classes together for some duration (as opposed to a dependency relationship, which represents a temporary association between two instances).

An association between the **Cash Dispenser** and the **Cash Drawer**, *named* **supplies Value**.

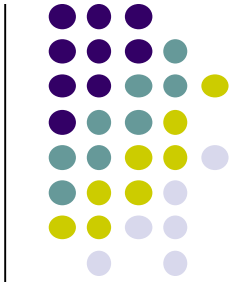
Role and Multiplicity of Associations



- Each end of an association is a **role** specifying the face that a class plays in the association**.**

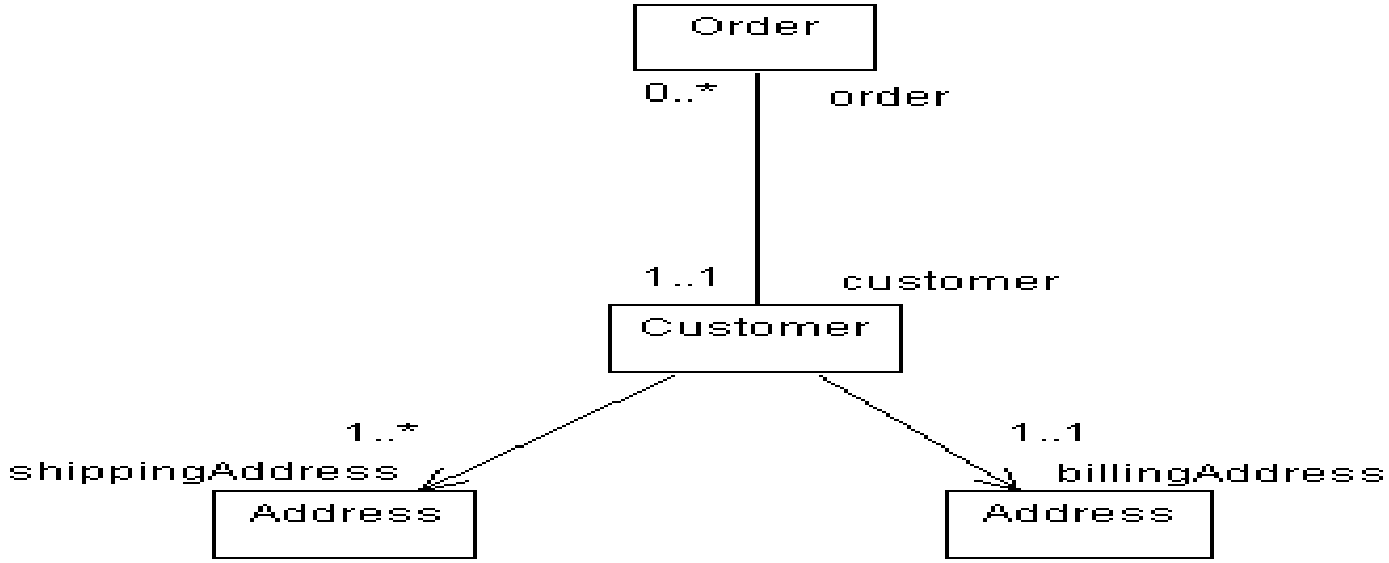
- For each role you can specify the **multiplicity** of its class, how many objects of the class can be associated with one object of the other class:
  - 1 Exactly one (as 1..1)
  - 0..1 Zero or one
  - 1..* One or more
  - 5..7 Specific Range (5, 6, 7)
  - 1..3,7 Combination (1, 2, 3, 7)

Navigability of Associations



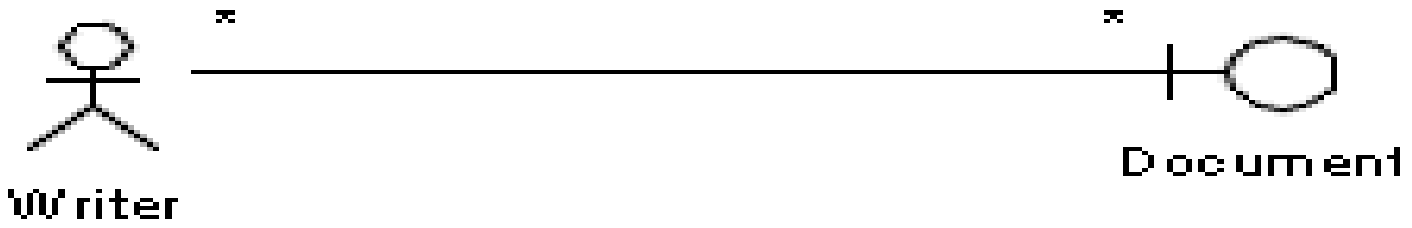
- The **navigability** property on a role indicates that it is possible to navigate from a associating class to the target class using the association. Implemented by object references.

Example

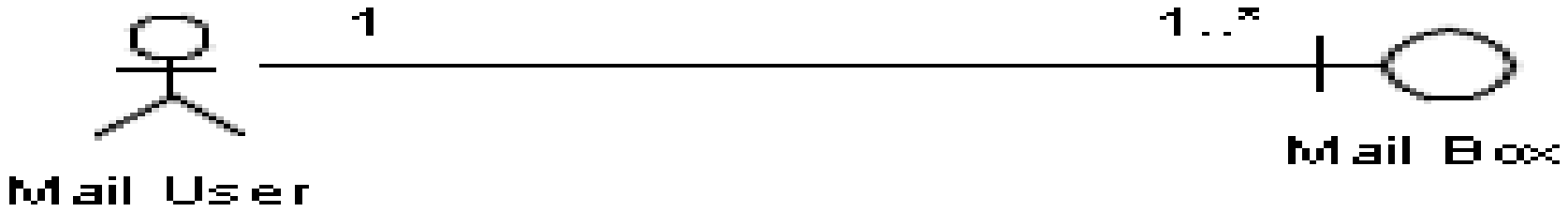


- A Customer can have two different kinds of Addresses: an address to which bills are sent, and a number of addresses to which orders may be sent. As a result, we have **two** associations between Customer and Address.
- An Order must have an associated Customer (1..1), but a Customer may not have any Orders (0..* at the Order end).
- **Customer** must know its **Addresses**, but the **Addresses** have no knowledge of which **Customers.**

Relating Actors and Boundary Classes



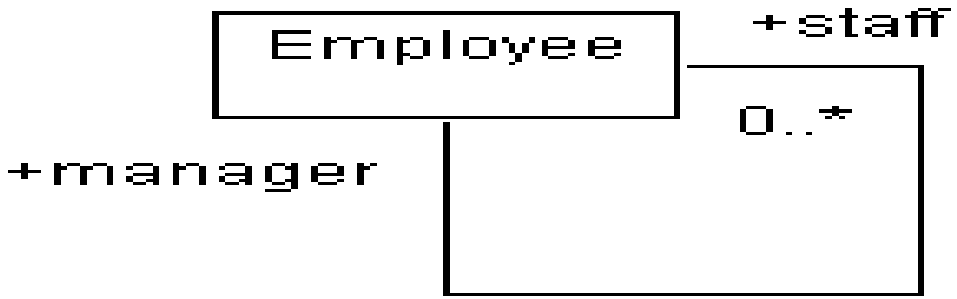
For the document editor, we have identified a Writer actor that interacts with Documents.



For the mail application, we have identified a Mail User actor that interacts with Mail Boxes.

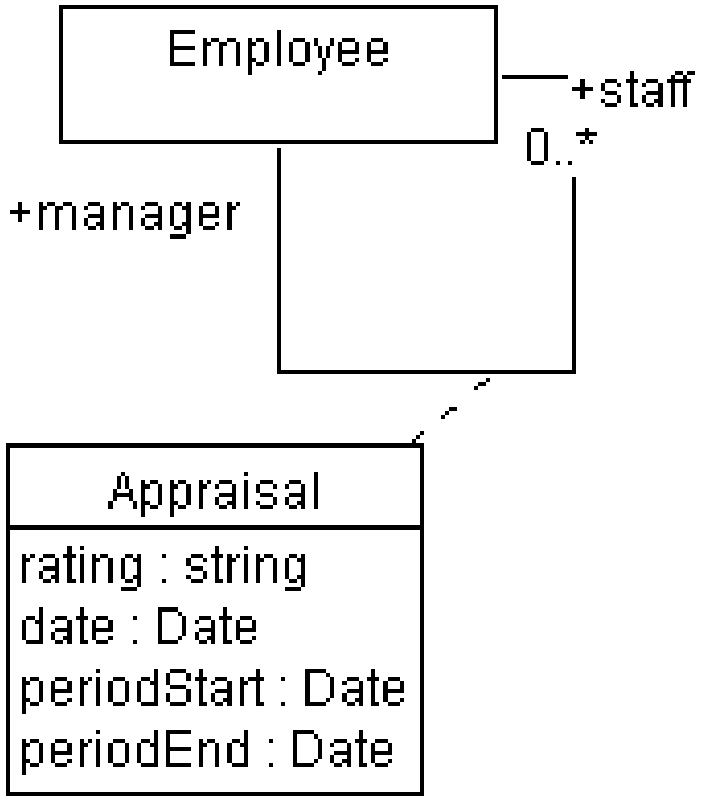
Self-Associations

A **Self-Association** means that one instance of the class has associations to other instances of the same class. Here, role names are essential to distinguish the purpose for the association.



•The Employee association is navigable in both directions since employees would know their manager, and a manager knows her staff.

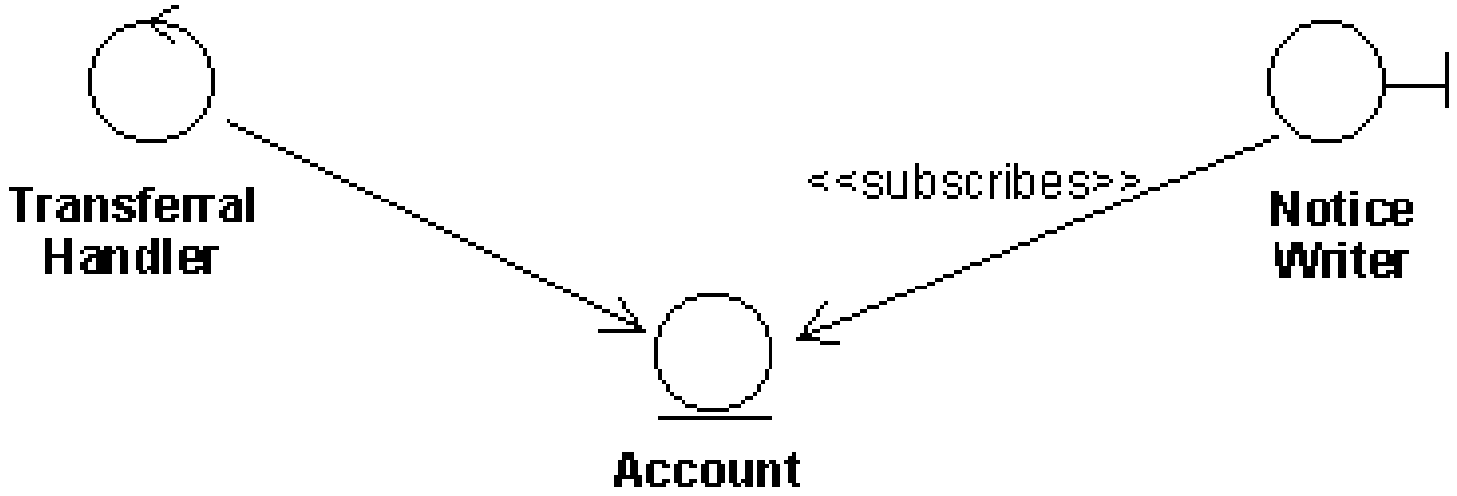
Self-Associations (cont.)



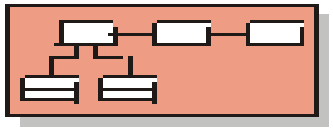
- An **association class** (Appraisal) is an association that also has class properties (shown by a dashed line). Its attributes, operations, and associations apply to the original association itself.

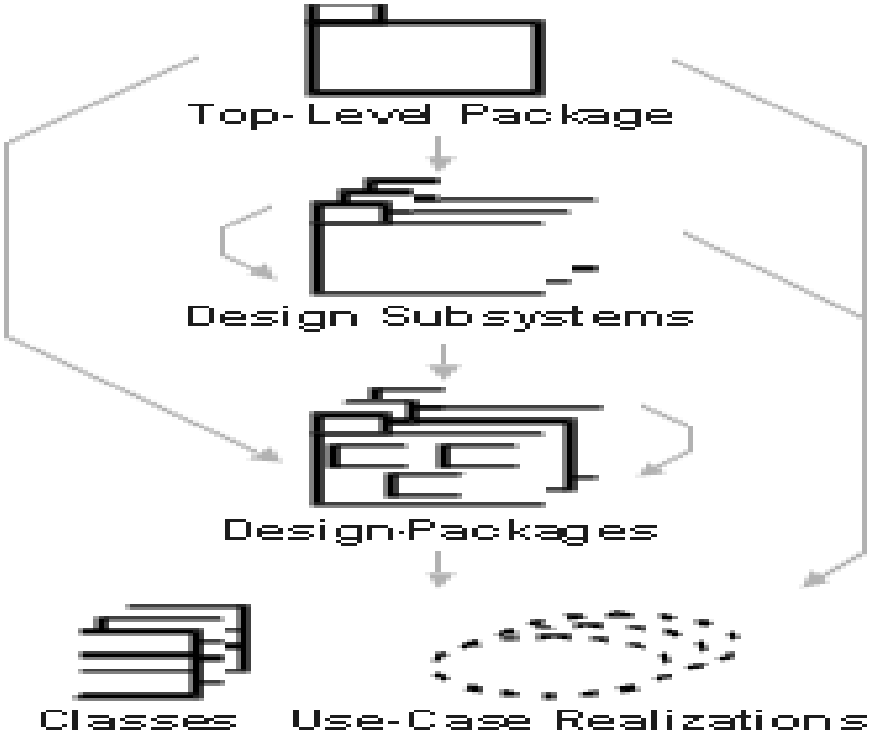
The Subscribe Association

The subscribes-association associates an object of any type with an entity object. The associating object will be informed when a particular event takes place in the associated entity object (e.g., Notice Writer is informed when Account balance is less than 0).



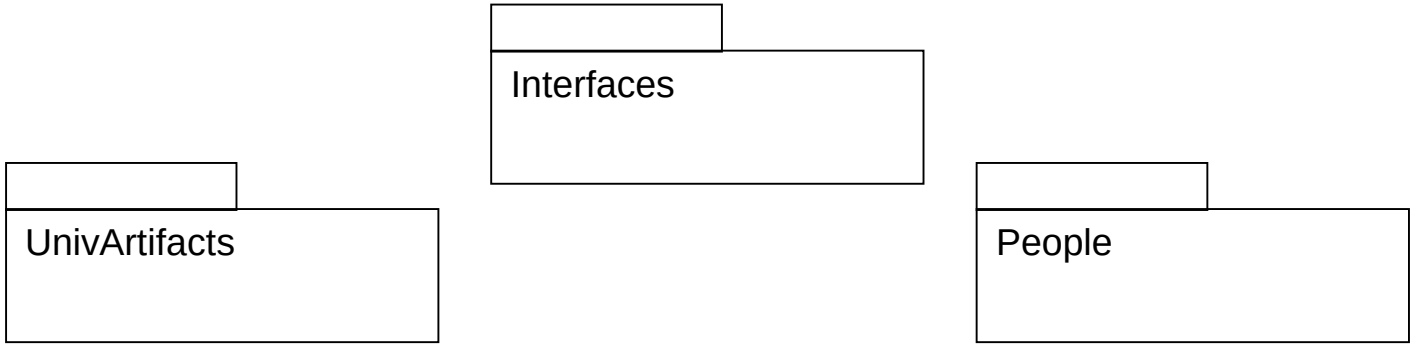
Class Diagrams and Packages

 **Class Diagram**	A **class diagram** shows a collection of declarative (static) model elements, such as classes, packages, and their contents and relationships.



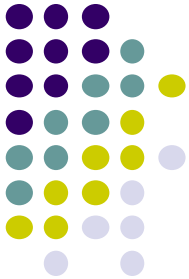
The design model is a hierarchy of packages (**design subsystems** and **design packages**), with "leaves" that are **classes** or **use-case realizations**.

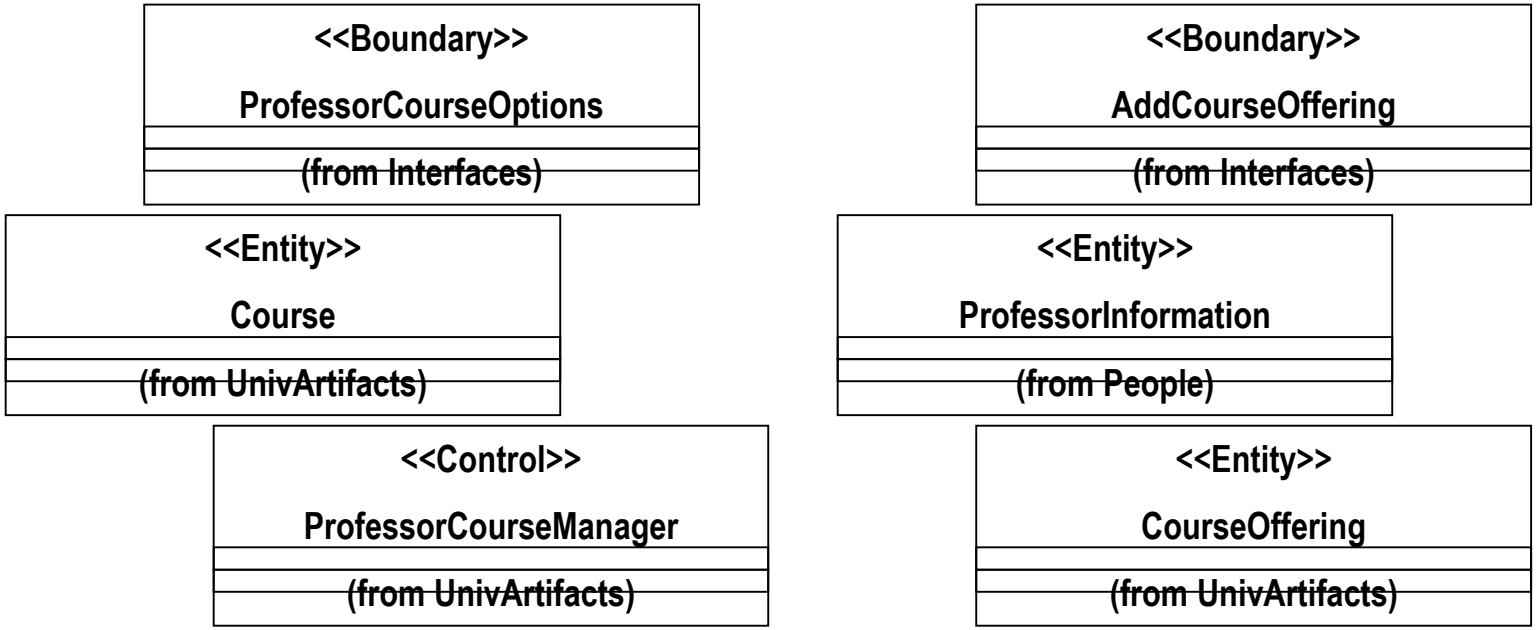
Example: The University Course Registration (UCR) Case Study



Class Diagram: Logical View/Main

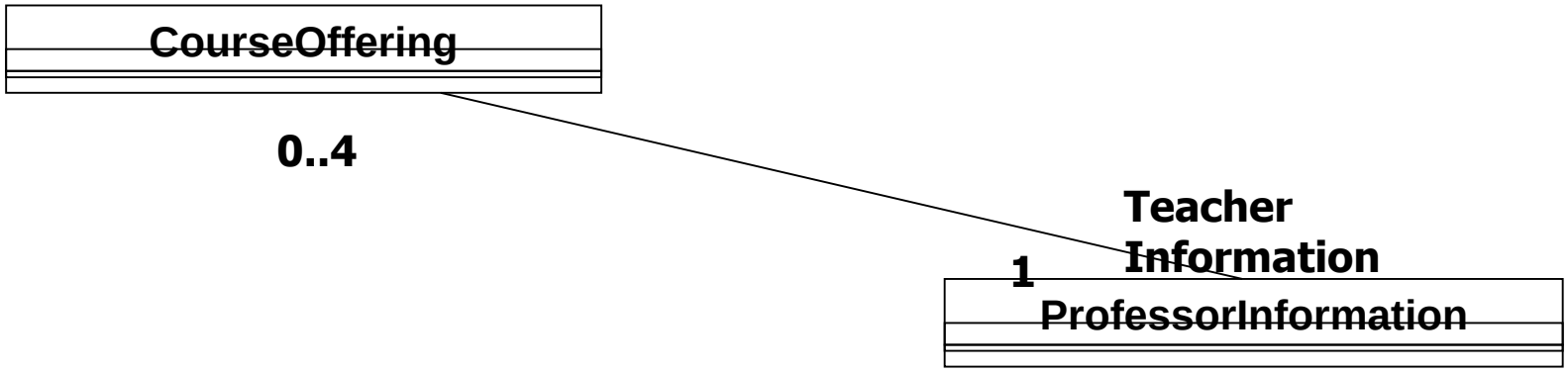
The University Course Registration (UCR) Case Study (cont.)





Class Diagram with Stereotype Display

UCR (cont.)



Multiplicity indications of the association between *CourseOffering* and *ProfessorInformation*