Structural Modeling

Class and Object Diagrams
CRC Cards.

Lecture : 9
Date : 1-11-2005

Yannis Tzitzikas
University of Crete, Fall 2005

Outline

- Structural modeling
- CRC Cards
- Class Diagrams
  - Classes
  - Attributes
  - Operations
  - Associations
  - Generalization
  - Constraints
- Object Diagrams

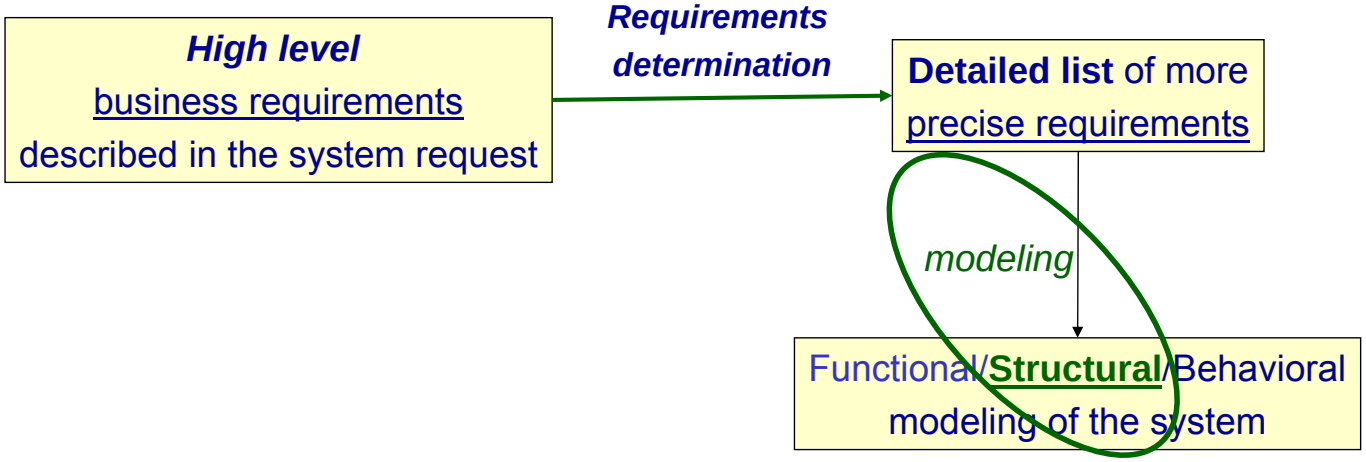
What is Structural Modeling?

Its objective is to describe:

- the structure of the data that supports the business processes

Why to do Structural Modeling ?

- Reduces the “semantic gap” between the real world and the world of software
- Establishes a common vocabulary for analysts and users
- Represent things, ideas, and concepts of importance in the application domain



How we model the structure in OO Analysis and Design?

Usually we employ 3 types of models:

- ***CRC Cards***
  - capture the essential elements of a class
- ***Class Diagrams***
  - allow the description of the types of objects in the system and the various kinds of static relationships that exist among them
- ***Object Diagrams***
  - show example configurations of objects (instances rather than classes)

Remarks:

- We can define class diagrams from several perspectives.

CRC Cards

(Class-Responsibility-Collaboration Cards)

CRC Cards: Objectives

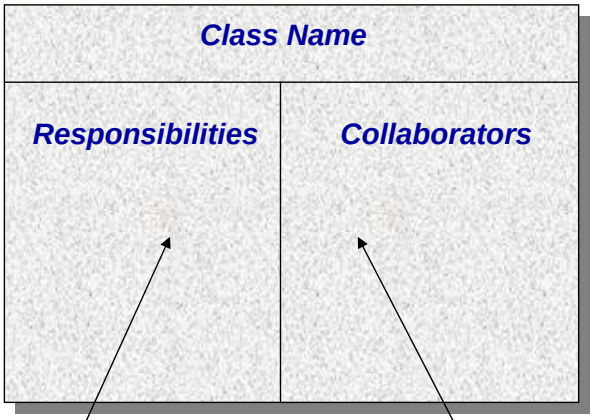
- It is an informal approach to object oriented modeling
- It is used for group brain-storming
- Proposed by Ward Cunnigham in the late 1980s

CRC cards help us:

- to identify and define the classes
- define and understand how they will collaborate

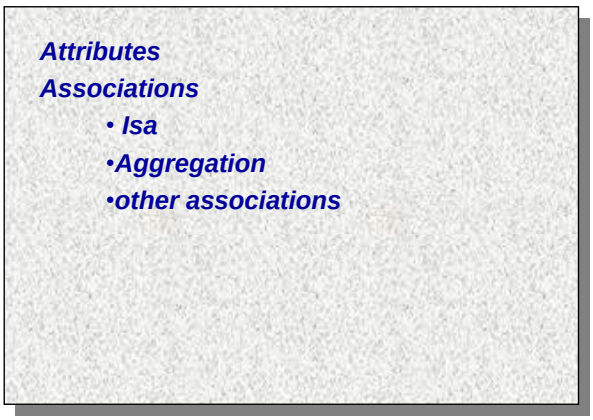
What is a CRC card?

front side:



Size:10 x 15 cm

back side



Responsibilities:

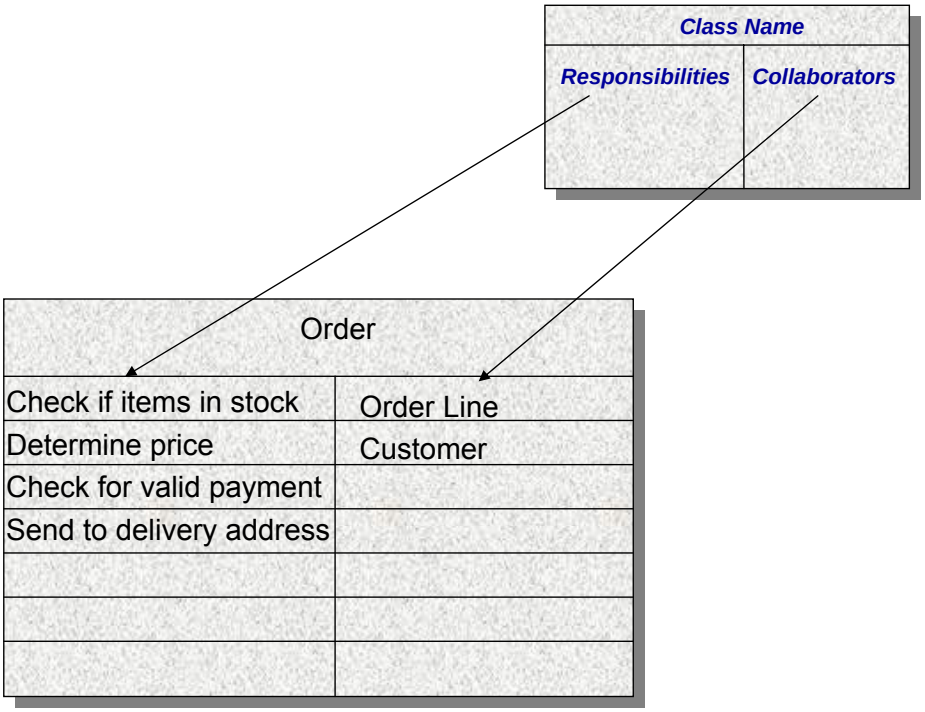
- of ***Knowing***
- of ***Doing***

Collaborators:

- Objects ***working together*** to service a request
  - i.e. UML associations

Consider the front as the public information, and the back as the encapsulated, implementation details

Example of a CRC Card



Another Example

	Responsiblity	Collaborators
Teacher	Teaches Lessons Evaluates Students	Secretary Student Principal
Student	Learns Lessons	Teacher Principal
Principal	Administers Funds Diciplines Students Hires Staff	Teacher Secretary Student
Nurse	Gives First Aid Gives Vacinations	Students Teachers
Secretary	Answers Phone Prints Handouts	Teacher Principal
Janitor	Cleans Building Fixes Equipment	Teacher Secretary
Cook	Prepares Meals	Janitor

Grade school example

Taken from Cunnigham (Tektronix)

What we can do with CRC cards?

- CRC cards represent the **static** (structural) view of the system's classes
- The **dynamic** description can be informally described by "***role-playing***"
  - Other techniques for describing the dynamic behavior:
    - Sequence Diagrams
      - (they will be covered in the lecture about Behavioral modeling)

Working with CRC cards

...“Card Playing”:

- The team (<7 persons) sits around a table
  - domain experts, analysts, oo developers
- they start by identifying a number of classes of the problem domain
- they create one card for each class
  - the responsibilities should not be too many (they should fit in the card)
- they can then start role-playing the scenarios of the Use Cases
  - each person can role-play one ore more cards
  - they pick up on the air the classes that are active
  - they move them to show the exchange of messages
- If something doesn't seem right, they change accordingly the cards (by changing their contents, or by creating/destroying cards)

How we start?

- A good **starting point** for CRC analysis is Use Cases.
- Start by trying to identify the classes in the problem domain.
  - Use the requirements document, identify the classes that are obvious in the subset of the problem that is going to explored in this session.
    - Find all of the **nouns** and **verbs** in the problem statement.
    - The nouns are a good key to what class are in the system, and the verbs show what the responsibilities are going to be.
    - Use this information for the basis of a brainstorming session and identify all the class that you see. Record them and filter the results after the brainstorming

Example: Identifying the classes by analyzing the text

candidate classes

Problem statement.

- *This application will support the operations of a technical library for an R&D organization. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. Users will enter their company ids in order to use the system; and they will enter material ID numbers when checking out and returning items.*
- *Each borrower can be lent up to five items. Each type of library item can be lent for a different period of time (books 4 weeks, journals 2 weeks, videos 1 week). If returned after their due date, the library user's organization will be charged a fine, based on the type of item(books $1/day, journals $3/day, videos $5/day).*

Example:Identifying the responsibilities

candidate responsibilities

Problem statement.

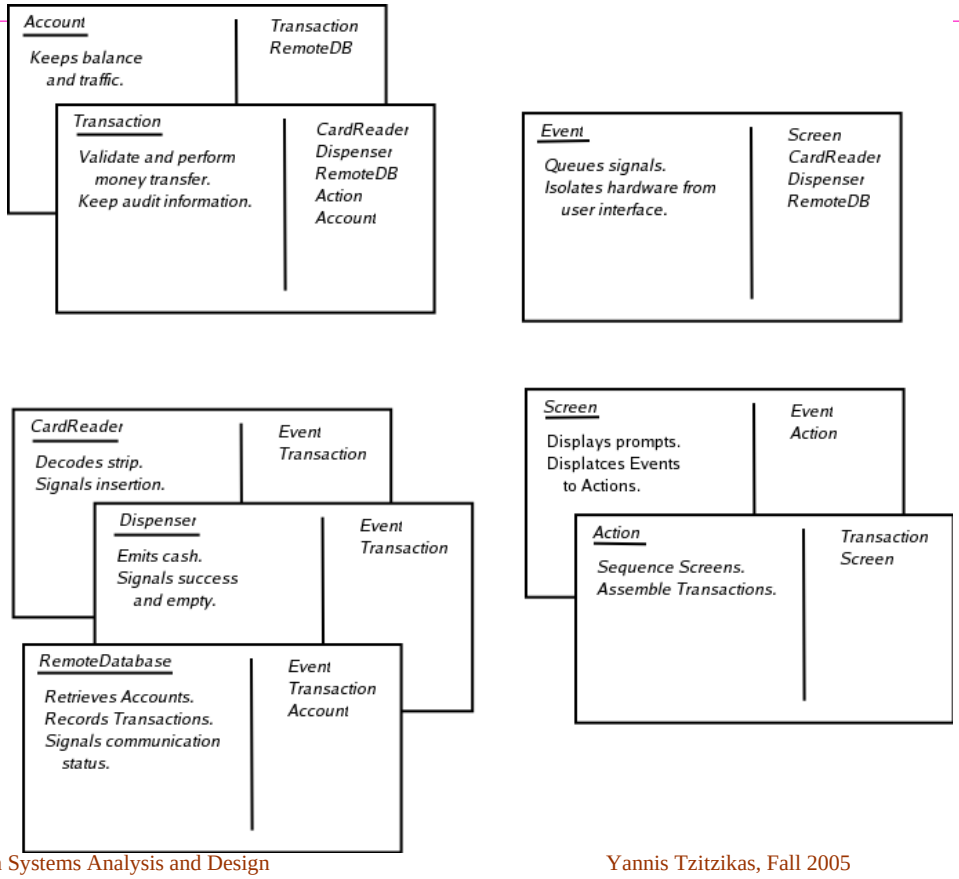
- *This application will support the operations of a technical library for an R&D organization. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. Users will enter their company ids in order to use the system; and they will enter material ID numbers when checking out and returning items.*
- *Each borrower can be lent up to five items. Each type of library item can be lent for a different period of time (books 4 weeks, journals 2 weeks, videos 1 week). If returned after their due date, the library user's organization will be charged a fine, based on the type of item(books $1/day, journals $3/day, videos $5/day).*

Scenario execution

*The scenarios of Use Cases can be used as a kind of script for the **role-playing method** of checking the CRC cards.*

- Start with scenarios that are part of the systems normal operation first.
- Then consider the exceptional scenarios (e.g. error recover)
- For each scenario decide which class is responsible. The owner of the class then picks up his card
  - When a card is in the air it is an object and can do things.
  - The own announces that he needs to fulfill his responsibility.
  - The responsibility is refined in to smaller tasks if possible. These smaller tasks can be fulfilled by the same object or by interacting with other objects. If no other appropriate class exist, maybe you need to make one. This is the fundamental procedure of the scenario execution.

CRC cards describing an ATM



The benefits of using CRC cars

- CRC cards allow … animated discussion among the team
  - the participants can experience how the system will work
- with CRC cards it is easy and fast to explore various alternatives (sequence diagrams can be slow to draw)
- CRC cards are portable (no computers are required)
- CRC cards are a useful tool for teaching people the object-oriented paradigm.

A set of steps for Structural Modeling

1. Create CRC cards by analyzing the text of the Use Cases
2. Brainstorm additional candidate classes
3. Role-play each use-case using the CRC cards.
4. Create the class diagram based on the CRC cards.
5. Review the structural model for missing and/or unnecessary classes, attributes, operations, and relationships.

Class Diagrams

What is a Class?

A **class** describes a *group* of objects with
- similar properties (attributes),
- common behaviour (operations),
- common associations to other objects.

How we find classes?
- Use common sense
- Listen to domain experts
- CRC analysis

We define classes for several parts of the system

Typical examples:
- *Application domain classes*
- *user interface classes*
- *data structure classes*
- *file structure classes*
- *operating environment classes*
- *document classes*
- *multimedia classes*
- *...*

Attributes and Operations

- Attributes
  - Units of information relevant to the description of the class
  - Only attributes important to the task should be included
- Operations
  - Actions that instances/objects can take
  - Focus on relevant problem-specific operations (at this point)
- Relationships
  - Generalization
    - Enables inheritance of attributes and operations
  - Aggregation
    - Relates parts to wholes
  - Association
    - Miscellaneous relationships between classes

Class Diagram

- **Class Diagram** = A description of the types of objects in the system and the various kinds of static relationships that exist among them
- Two principal kinds of static relationships
  - **associations** (a Person can own a Car)
  - **subtypes** (a student is a kind of person)
- they also show the attributes and operations of a class and the constraints that apply to the way objects are connected

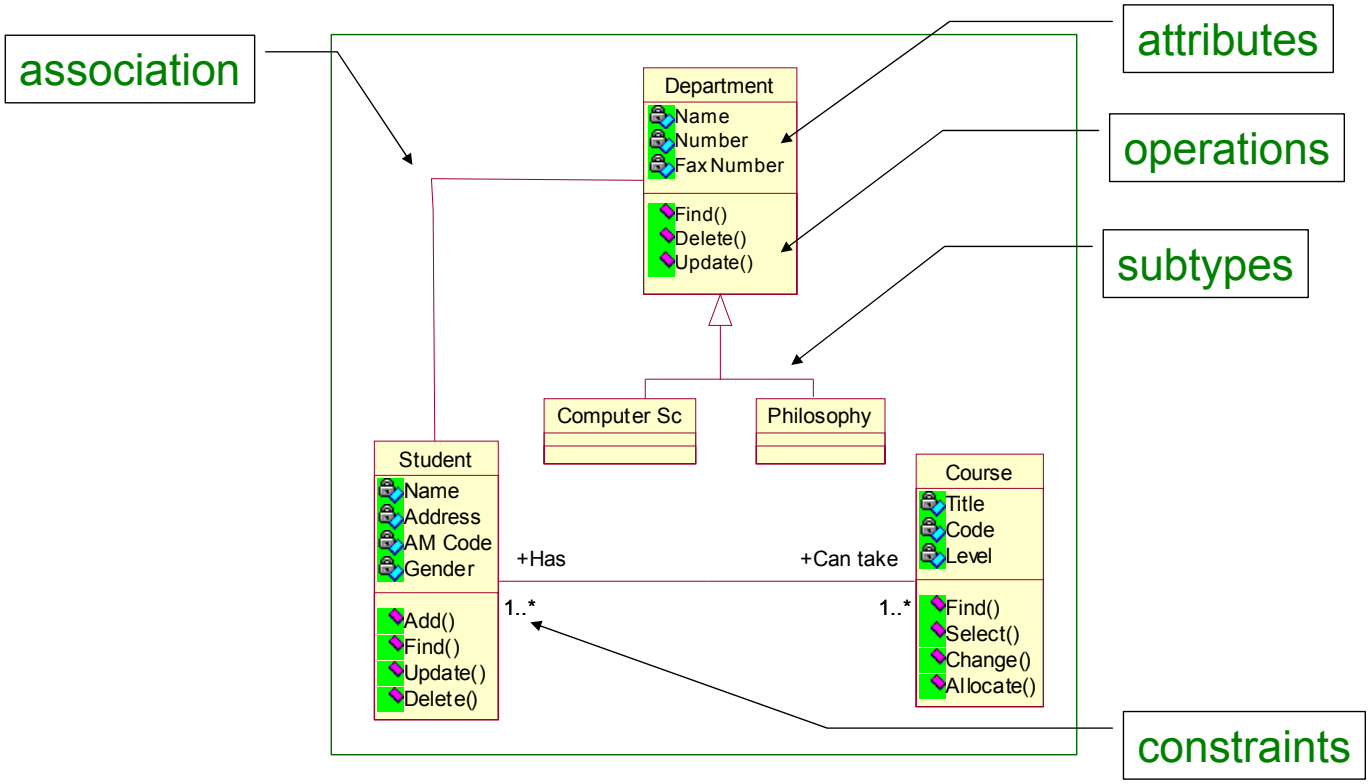
The 3 Perspectives of a Class Diagram

- There are 3 perspectives for the design of a class diagram (of a conceptual model in general)
  - ***Conceptual***
    - Independent of implementation. This is often called **domain model.**
  - ***Specification***
    - Based on **interfaces of the software**, not the implementation
  - ***Implementation***
    - Here we model the **implementation classes**. This is the most often used perspective
- Perspectives are not part of the formal UML
- By *tagging* classes with a stereotype, we can provide an indication of the perspective

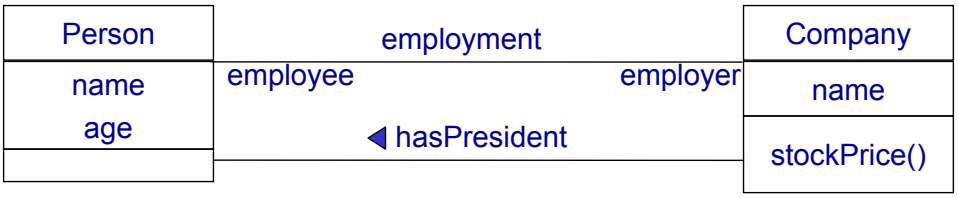
Keywords (UML v2) and Stereotypes (UML v1)

- It is the core extension mechanism of the UML
- If you need a modeling construct that isn't in the UML but is similar to something that is, you treat your construct as a **stereotype** (UML v.1), or **keyword** (UML v.2) of the UML construct.
- Denoted by **<<name>>** (or sometimes **{name}**)
- E.g. interface
  - A UML interface is a class with only public operations with no method bodies nor attributes (like in Java, CORBA)
  - Denoted by <<interface>>
- We could define stereotypes of classes, associations, generalization.
  - We would consider them as subtypes of the meta-model types Class, Association, Generalization

Example of a class diagram



Associations (Perspective: *Conceptual*)



- Represent binary relationships between instances of classes
- Each end can be assigned a name called role name
- Multiplicity constraints
  - how many objects may participate in a given relationship
  - multiplicity indicates lower & upper bounds

    ***** ≡ **0..*** ≡ **0.. ∞** // no constraint

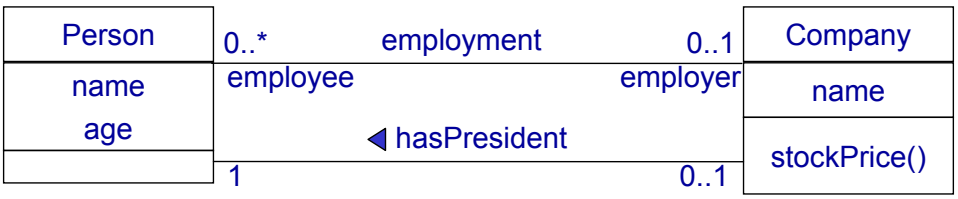
    1 ≡ **1..1** // mandatory and single-valued association

    0..1 // single-valued association
  - other more general multiplicity constraints

    1..11 (for soccer teams)

    3..4 (wheels of a car)

Associations (Perspective: *Conceptual*)



- Represent binary relationships between instances of classes
- Each end can be assigned a name called role name
- Multiplicity constraints
  - how many objects may participate in a given relationship
  - multiplicity indicates lower & upper bounds

    ***** ≡ **0..*** ≡ **0.. ∞** // no constraint

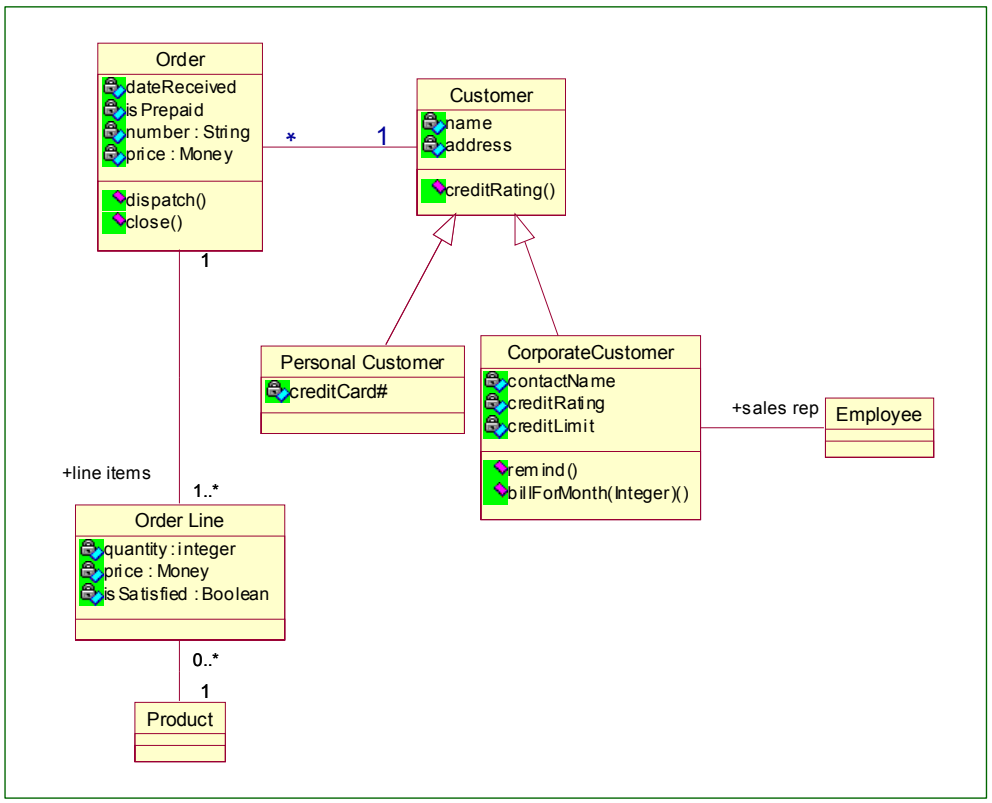
    1 ≡ **1..1** // mandatory and single-valued association

    0..1 // single-valued association
  - other more general multiplicity constraints

    1..11 (for soccer teams)

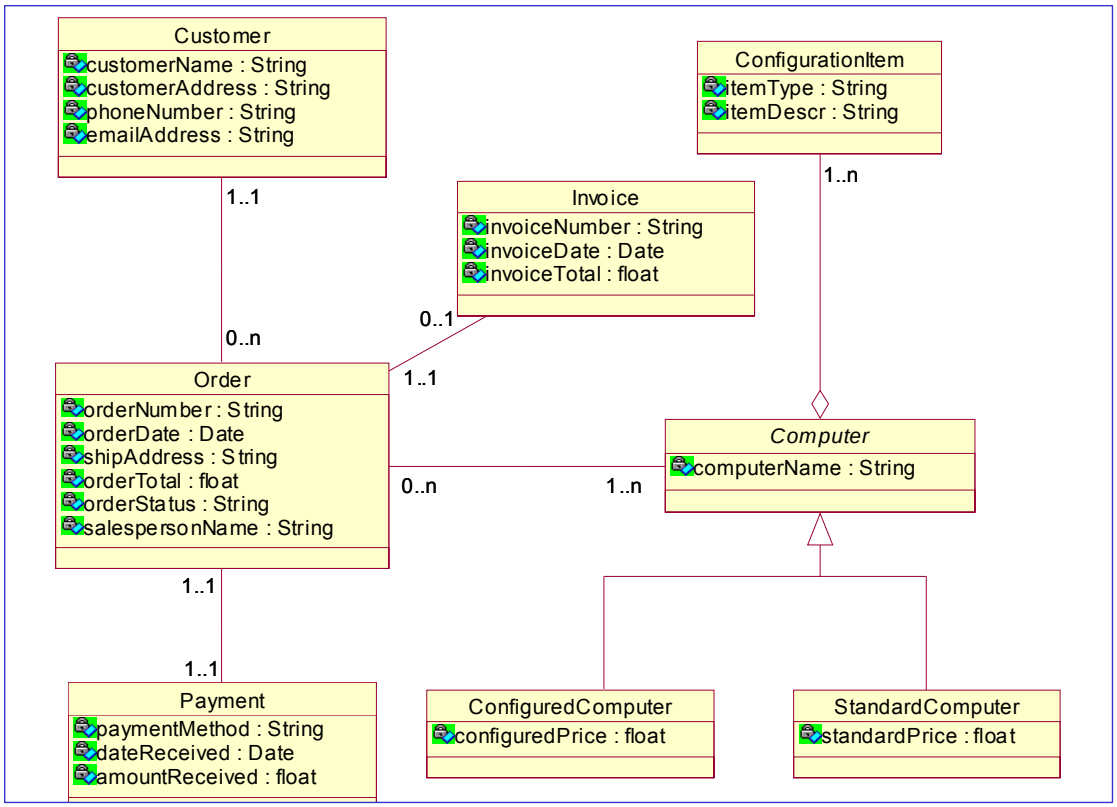
    3..4 (wheels of a car)

Another example



- Do it on class?

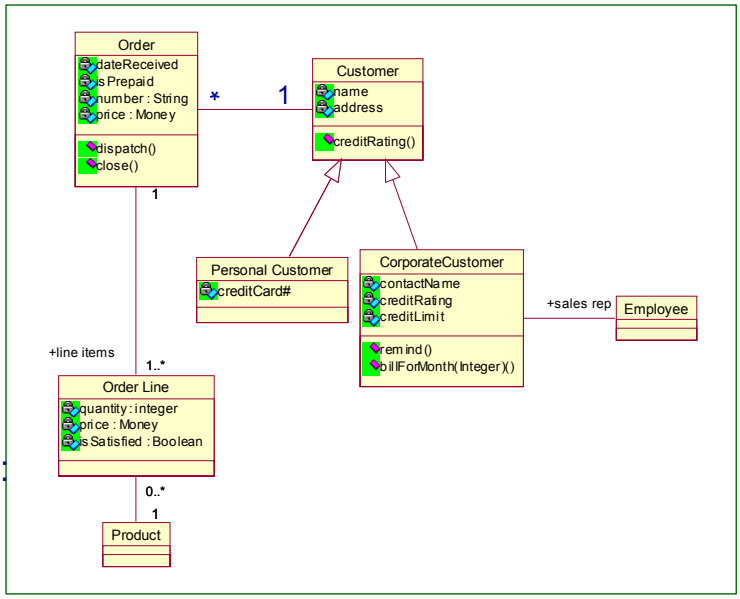
Associations
examples of multiplicity constraints



- Students should give the mult contraints

Associations (Perspective: *Specification*)

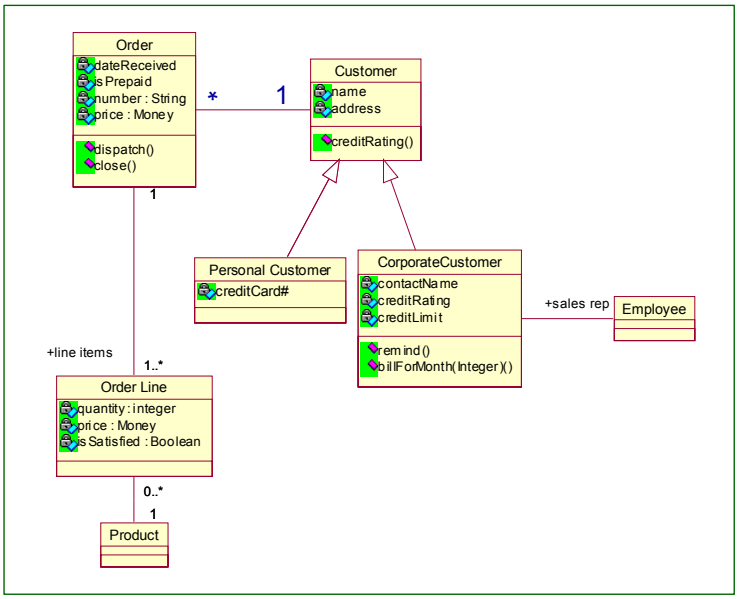
- Here associations represent responsibilities (read & update)
- from this diagram we may say that:
  - there are methods associated with customer that return the orders of a given customer has made
  - the reverse for Order (return the customer)
- we cannot infer implementation details:
  - I.e. if Order class contains a pointer to Customer, or if it calls a method of customer



Associations (Perspective: *Specification*)

- If this were an specification model we could infer the following interface for an Order class

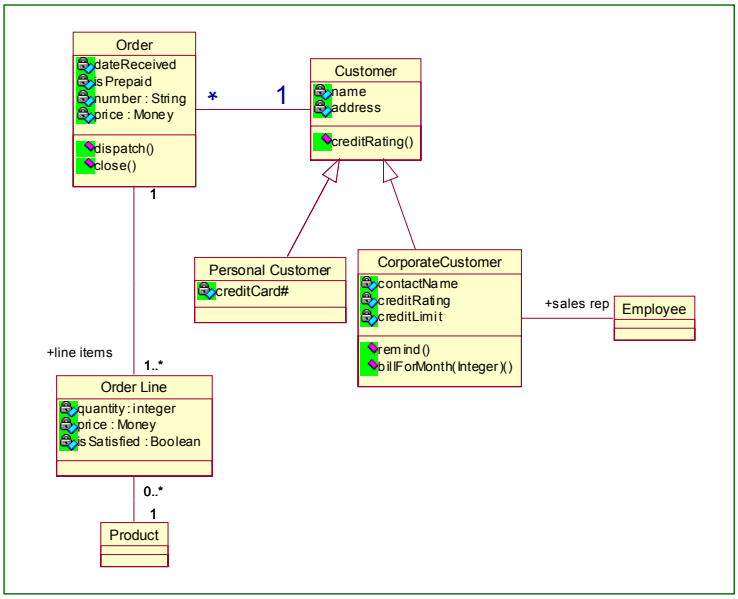
```
class Order {
    public Customer  getCustomer();
    public Set       getOrderLines();
```



Associations (Perspective: *Implementation*)

- If this were an implementation model we could infer:

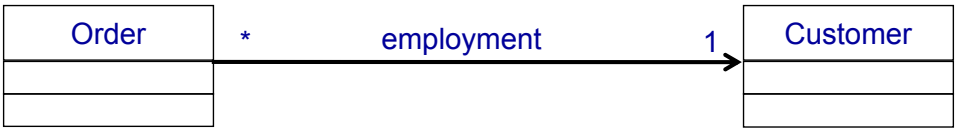
```
class Order {
    private Customer  _customer;
    private Set       _orderLines;

class Customer {
    private Set  _orders;
```



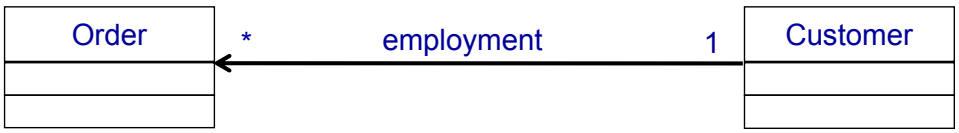
Associations and Navigability (perspective: Spec. and Impl)

- Useful only for Design and Implementation perspective (not for conceptual)
  - unidirectional
  - bidirectional

Navigability (unidirectional associations)

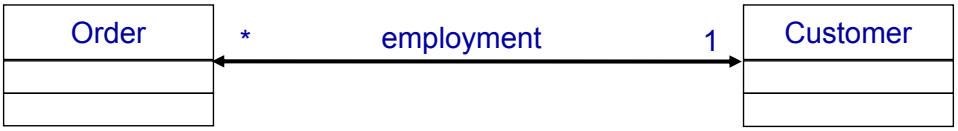


- *Spec*: Order has **responsibility** to tell you which customer it is for
- *Impl*: Order contains a **pointer** to Customer (and not the other way around)



- *Spec*: Customer has **responsibility** to tell you his/her orders
- *Impl*: Customer contains a **set of pointers** to Orders

Navigability (bidirectional associations)

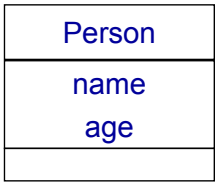


- *Spec*: **Both** have the responsibility to tell you the other end
- *Impl*: **Both** contain pointers to the other end

When we implement a bidirectional association in a programming language we have to be sure that both properties are updated.

note.

Attributes



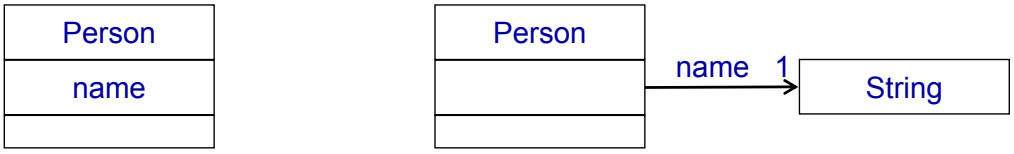
- *Conceptual*: **Property**
  - e.g. a Person has a name
- *Specification:*
  - e.g. a Person object can tell/set its name
- *Implementation*:
  - e.g. a Person object has a field (instance variable)

- Like associations
  - small, simple classes, such as strings, dates, money objects, and non-object values like Integer and Real.

Attribute syntax in UML:

visibility name: type = defaultValue

Difference between Attributes and Associations



- Conceptual perspective: No difference
  - attributes are usually single-valued
  - can be optional, mandatory, have multiplicity
    - e.g. dateReceived [0..1]: Date
- Specification/Implementation perspective:
  - attribute => navigability from the type to the attribute only
  - each person has its own copy of attribute object (value semantics rather than reference semantics)

Operations

Are the processes a class knows to carry out

Company
name
hire(p:Person) fire(p:Person) promote(p,incr)

- *Conceptual* perspective:
  - Indicate the principal responsibilities (described in a couple of words)
- *Specification* perspective:
  - Public methods on a type
- *Implementation* perspective:
  - plus private/protected operations

Operations in UML

syntax:

visibility name (parameter-list): return-type-expression {property-string}

- **visibility**:
  - **+** : public (by all used)
  - **-** : private (by owning class)
  - # : protected (by owning class and its subclasses)
- **name**: a string

Operations in UML

syntax:

visibility name (parameter-list): return-type-expression {property-string}

- **parameter-list**: comma separated parameters with syntax that of attributes (plus direction), i.e. **direction name: type = default value**
  - direction (default: in)
    - in: used for input
    - out: used for output
    - inout: used for both
- **return-type expression**: comma-separated list of return types
  - can be more than one

Operations in UML

syntax:

visibility name (parameter-list): return-type-expression {property-string}

- **property-string**: property values that apply to the given operation
  - {abstract}: it requires a child to complete the implementation
  - {leaf}: not polymorphic (may not be overridden) // like *final* in Java
  - {query}: the execution of the operation leaves the state of the system unchanged
  - {sequential}: only one flow should be in the object at a time
  - {guarded}:
  - {concurrent}
  - {static}: it behaves as a global procedure

Operations in UML

syntax:

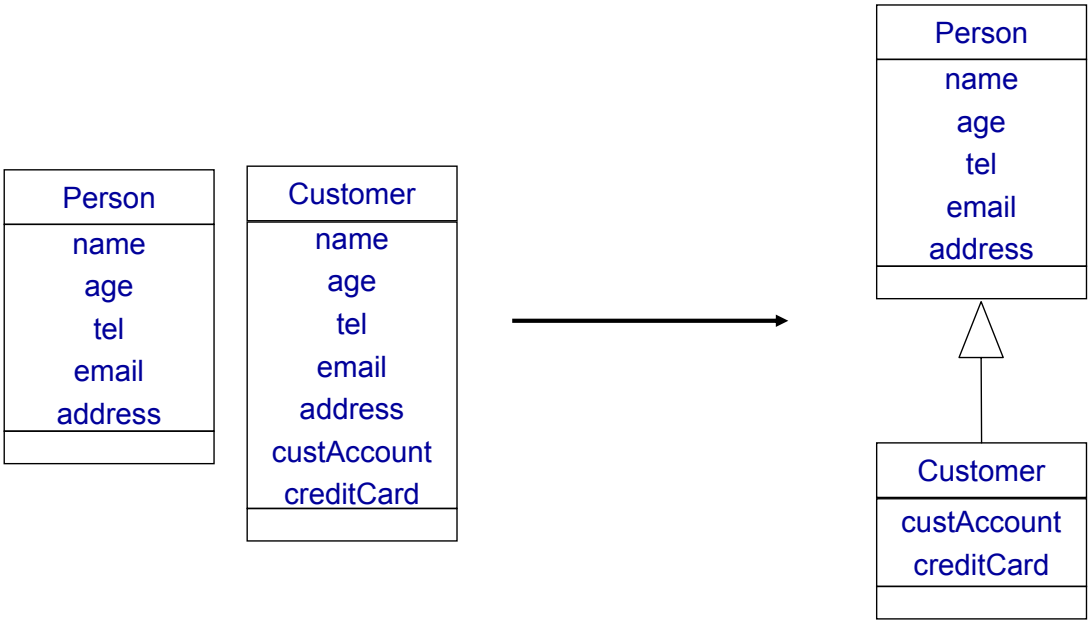
visibility name (parameter-list): return-type-expression {property-string}

- Examples:
  - +balanceOn(date:Date):Money

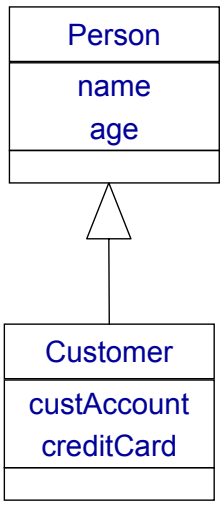
Operations (II)

- Constructor
  - creates an object
- Queries vs Modifiers
  - query: an operation that gets a value from a class without changing its state (i.e. without side effects)
    - we mark them with the constraint {query}
  - modifier: an operation that changes the state
- Operations vs Methods
  - **operation**: the procedure call (else called method call or method declaration)
  - **method**: the body of the procedure (else called method body)
  - the above are different if we have polymorphism
    - if we have a supertype and three subtypes, each of which overrides the supertype's "foo" operation, then we have 1 operation and 4 methods that implement it.

Generalization



Generalization



- *Conceptual* perspective:
  - Subset of instances
  - inheritance of properties
- *Specification* perspective:
  - The interface of the subtype must include all elements from the interface of the supertype.
  - The subtype's interface is said to "conform to" the sypertype interface
- *Implementation* perspective:
  - Associated with inheritance in PLs
  - Subtypes inherit all methods and fields and may override inherited methods

Constraint Rules

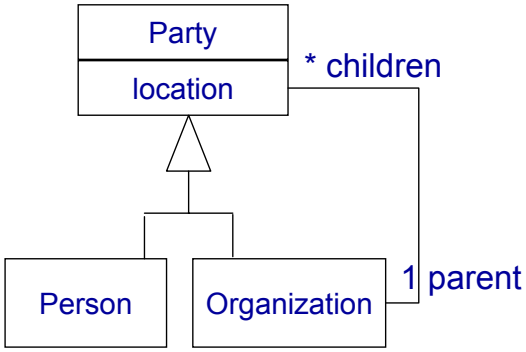
- A diagram actually specifies a set of constraints
- However, we need to express more constraints (apart from those we have seen so far)
- UML wants to put them inside braces { } // e.g. informal English
- There is also a formal **Object Constraint Language** (OCL)
  - Warmer/Kleppe 98. OCL will be covered in a subsequent lecture.
- Ideally, they should be implemented by *assertions* in the PL
- These correspond with the "Design by Contract" notion of invariants.

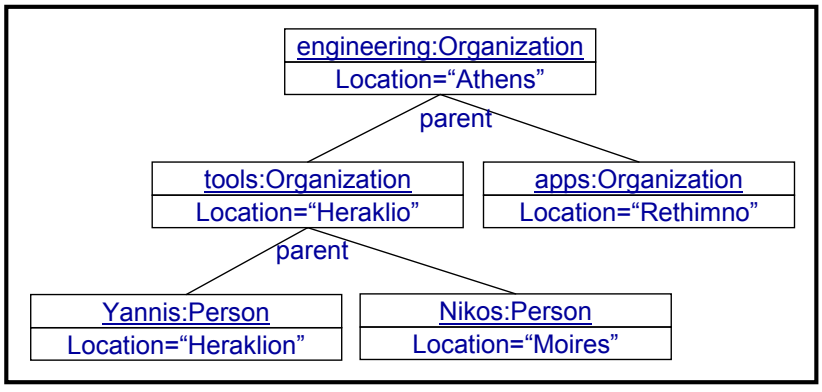
Object Diagrams

Object Diagram

- Shows instances rather than classes. Also called instance diagrams
- Can show an example configuration of objects
  - it is like a collaboration diagram but without messages

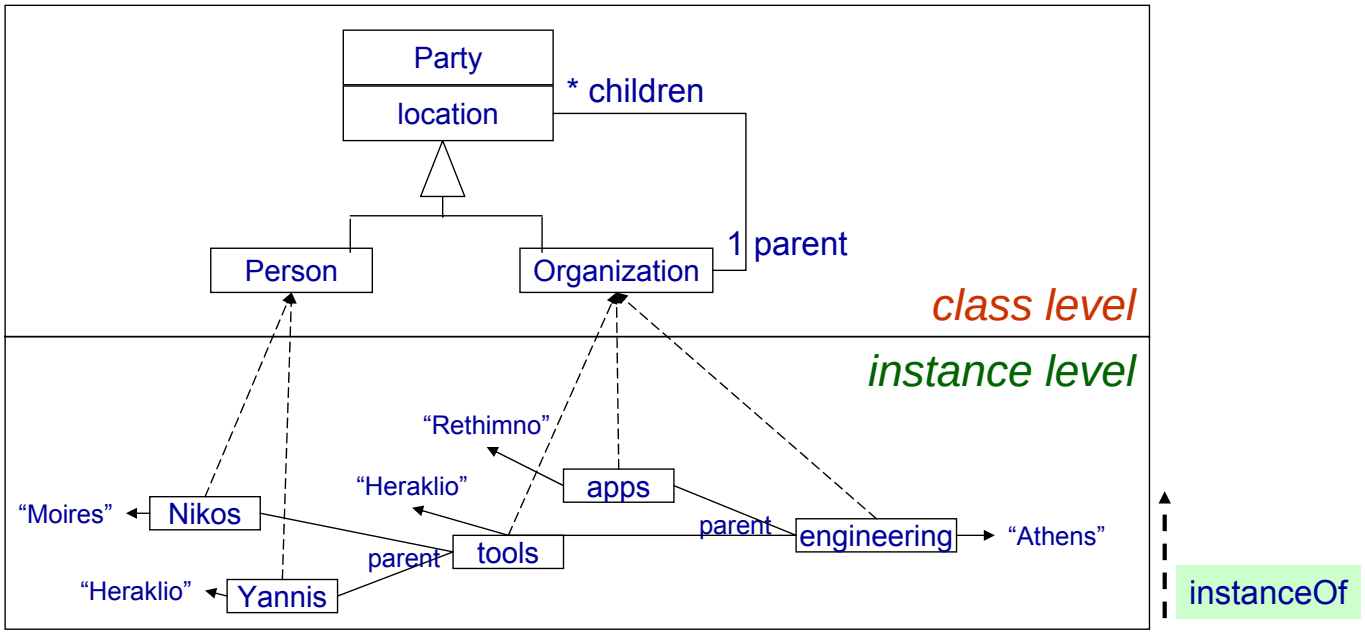
form of names: instanceName:className





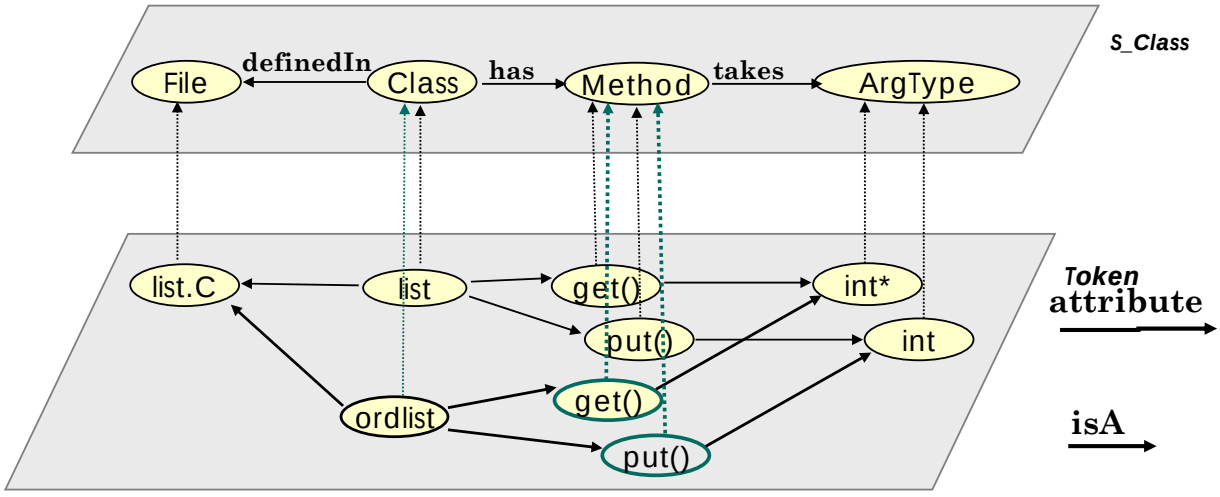
Presenting Class and Object Diagrams Together

- Sometimes useful (if class diagrams are small)

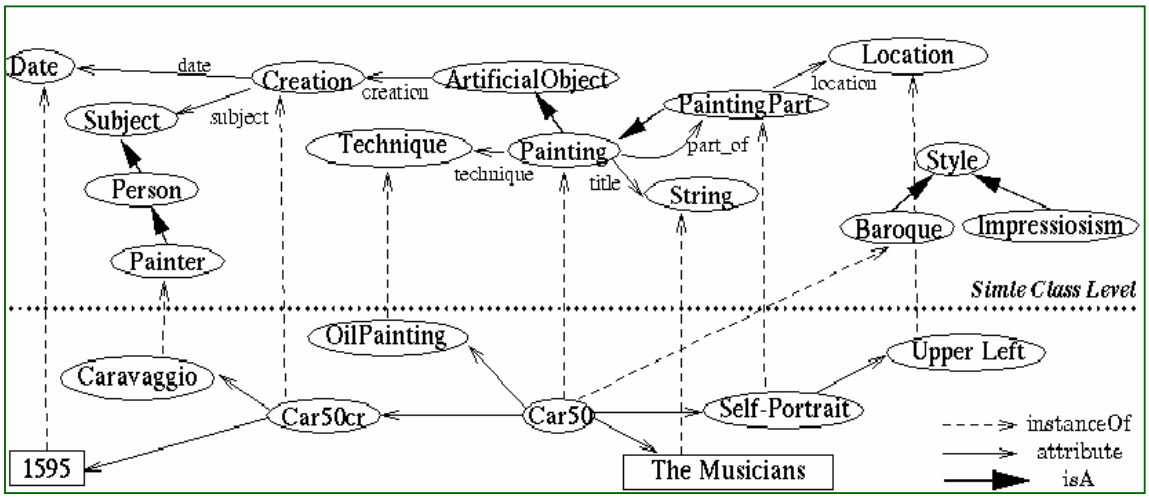


Presenting Class and Object Diagrams Together

- A model for statically analyzing code



Presenting Class and Object Diagrams Together



Tips

- If you are an analyst => draw conceptual models
- If you are a programmer => concentrate on specification models
- Draw implementation models only when you are illustrating a particular implementation technique

- Don’t draw models for everything: concentrate on key aspects

Summary

- *CRC cards* capture the essential parts of classes.
- *Class and object diagrams* show the underlying structure of an object-oriented system.
- Constructing the structural model is an iterative process involving: *textual analysis, brainstorming objects, role playing*, and *creating the diagrams*

Reading and References

- **Systems Analysis and Design with UML Version 2.0** (2nd edition) by A. Dennis, B. Haley Wixom, D. Tegarden, Wiley, 2005. CHAPTER 7
- **UML Distilled: A Brief Guide to the Standard Object Modeling Language** (3rd Edition) by Martin Fowler, Addison Wesley, 2004. Chap. 3
- **The Unified Modeling Language User Guide** (2nd edition) by G. Booch, J. Rumbaugh, I. Jacobson, Addison Wesley, 2004, Chap 8 (advanced: 9-10)

- CRC cards:A tutorial regarding CRC cards can be found at:
  - *http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b/*