Introduction to the Unified Modeling Language (UML)

Part 1

Giancarlo Succi

Goal of the topic

- To formalize the basic ideas of OO …
- … Using the Unified Modeling Language

- *These lectures are* ***NOT*** *a comprehensive review of UML (it would take 6 months full time … ☺)*

Structure of the course

- Brief summary of the core OO features that we will discuss
- Few basic definitions
- Overview of UML
- Object Oriented Concept Modeling in UML -
- Object Oriented Analysis in UML
- Object Oriented Design in UML

CAVEAT!!!

- Lots and lots and lots of details, names, and diagrams
- The core are classes and class diagrams
- It looks simple but it is not ...

Stop me when something get messy

Introduction to OO

Why is OO popular?

- The hope that it will increase productivity
- Natural way of structuring the world
  - Objects
  - Messages
  - Responsibility

What is object-oriented software development?

- A way to view the world of the application
- A way to describe a model of the application
- A comprehensive methodology that
  - allows to develop a software system
  - uses similar concepts within the whole development process

Object oriented system development

- Means to achieve high quality
  - Information Hiding
  - Abstraction
  - Modularization
  - Reuse

An object oriented approach - more or less - forces the software developer to apply these concepts

OO methodologies

- Late 80’s early 90: several OO methodologies developed
  - different notations
  - different processes
- Main approaches
  - Booch
  - Rumbaugh
  - Jacobson
  - UML

Key Idea

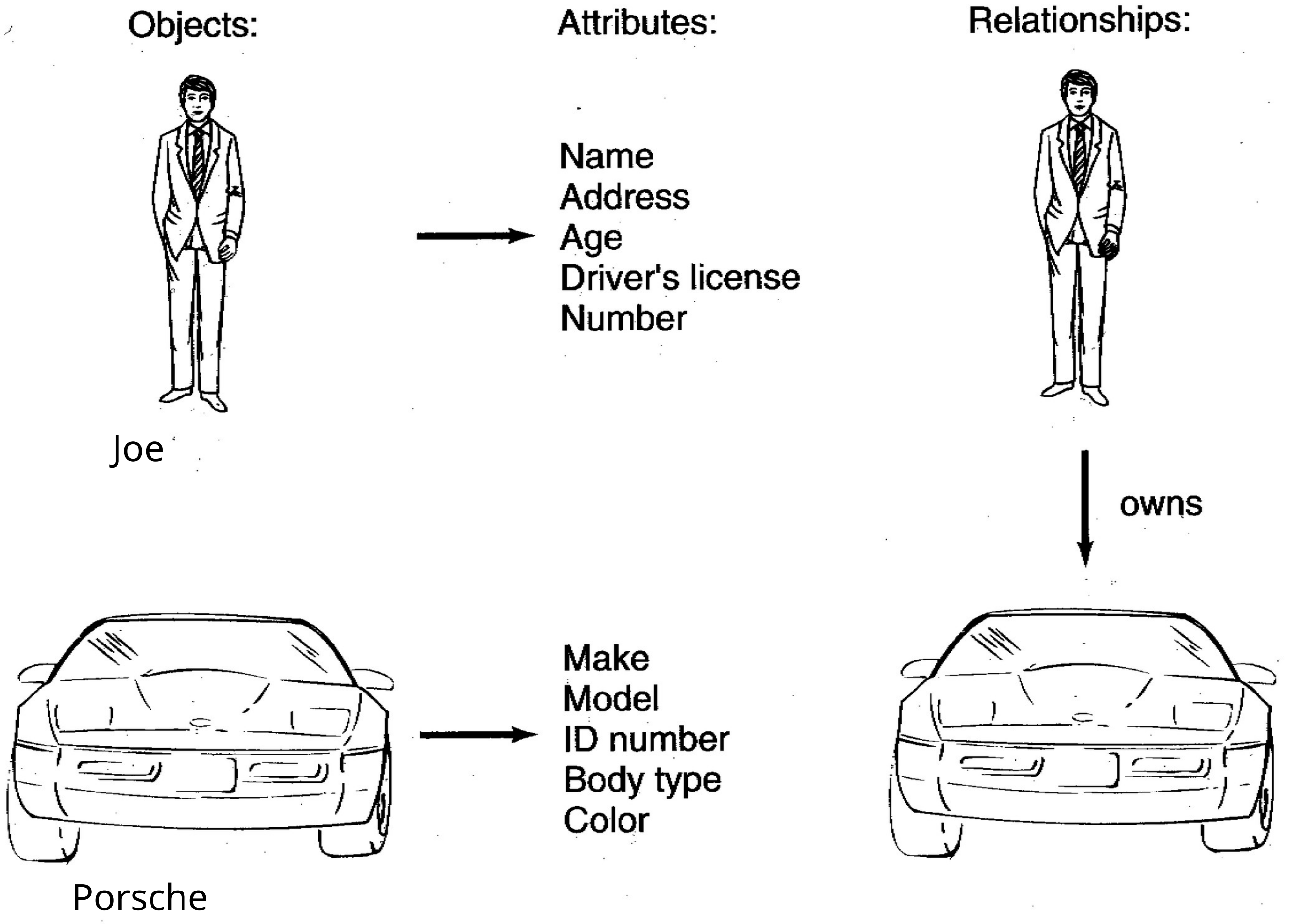
- Represent the world in terms of:

Interacting Objects

- Use this representation in all the life cycle development phases:

OO Concept Modeling
OO Analysis
OO Design
OO Programming

A Simple OO Model



Objects:

Attributes:

Relationships:

Name
Address
Age
Driver's license
Number

Joe

owns

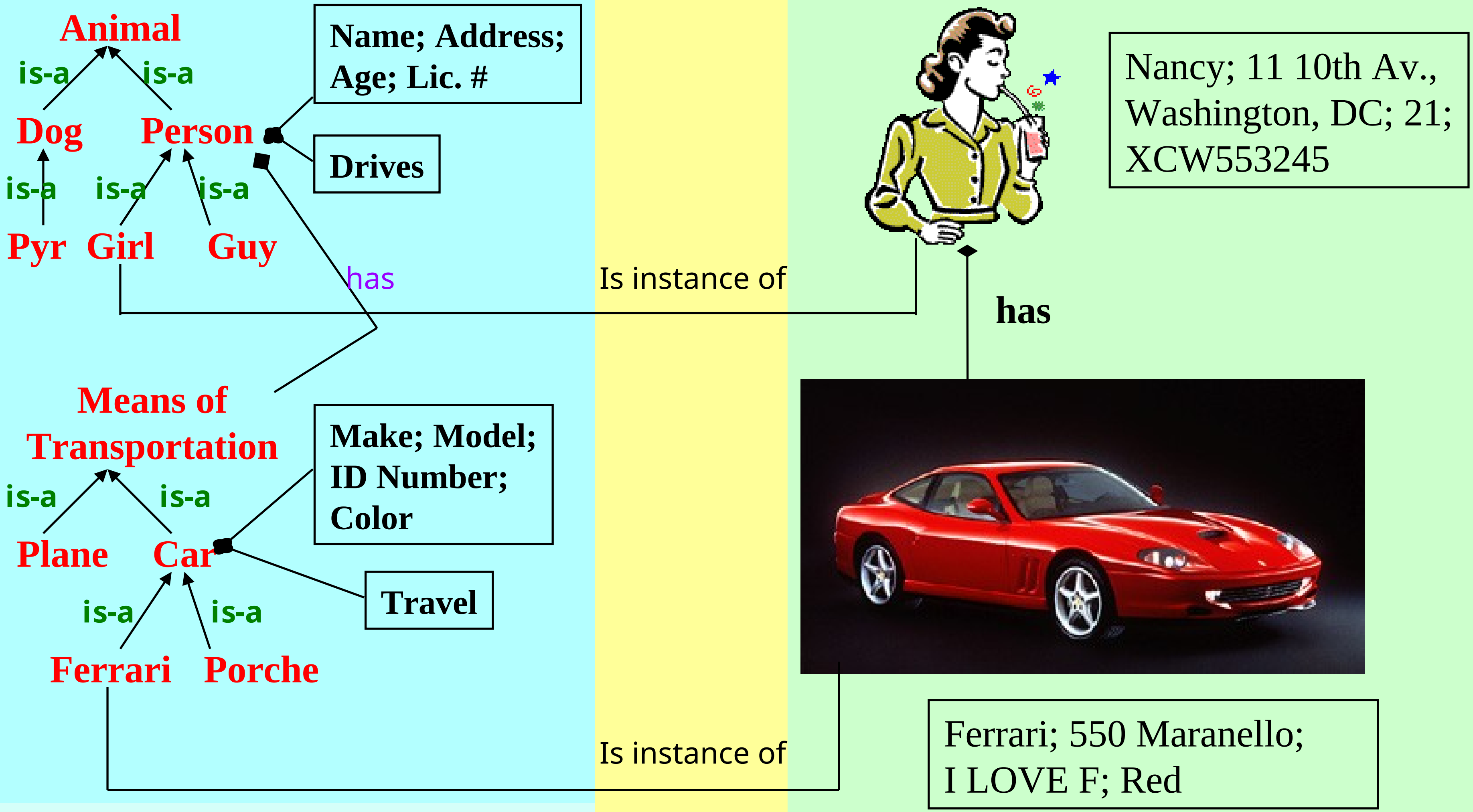
Make
Model
ID number
Body type
Color

Porsche

Key Concepts

- **classes and class hierarchies**
  - attributes
  - methods
  - inheritance
  - relations with other classes
- **objects: instances of classes**
  - attributes with assigned values
  - instantiated relations
- **messages and methods to respond to a message**

An OO Model



Name; Address; Age; Lic. #

Drives

Make; Model; ID Number; Color

Travel

Nancy; 11 10th Av., Washington, DC; 21; XCW553245

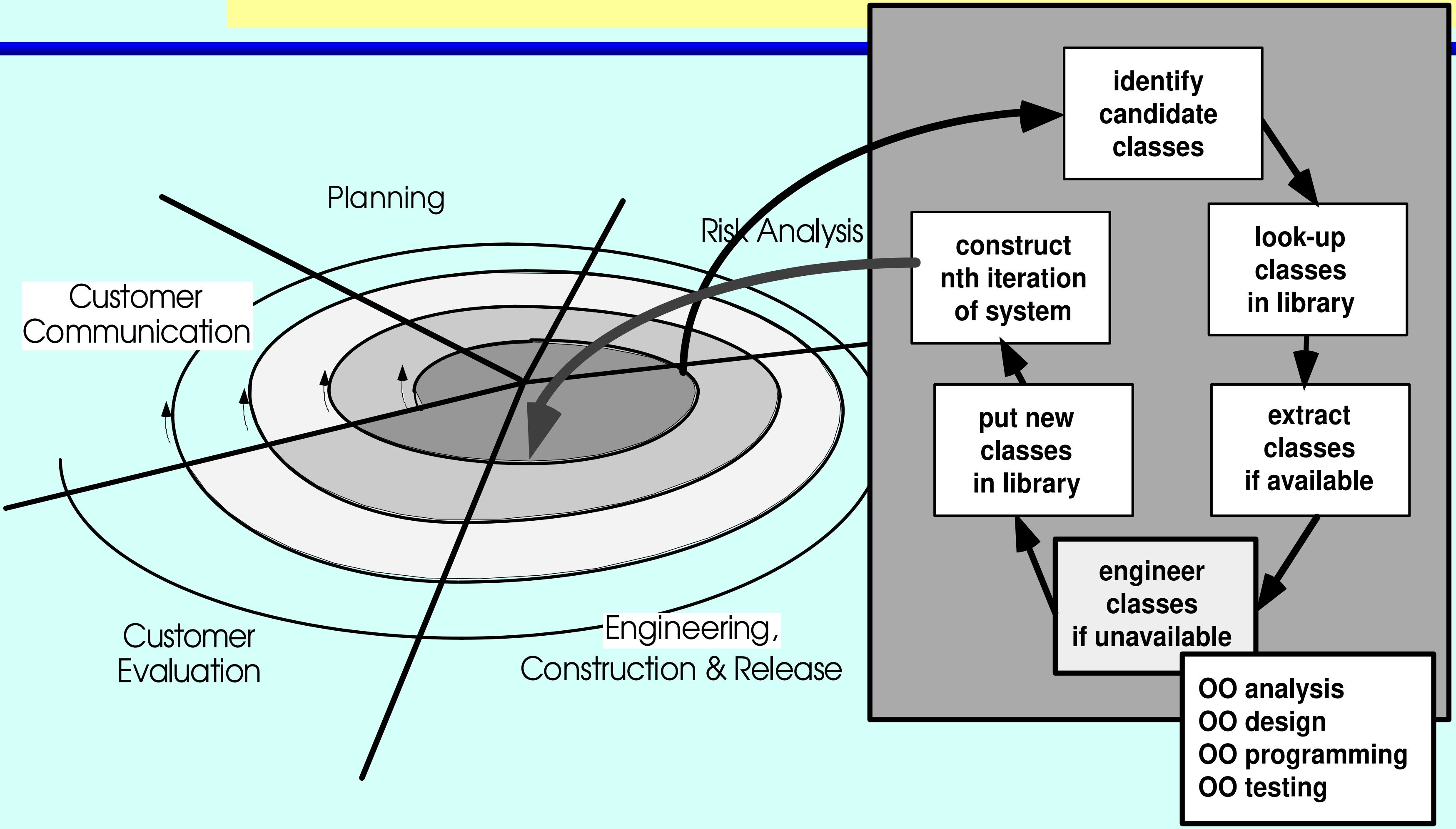
Ferrari; 550 Maranello; I LOVE F; Red

In our example

- Nancy is a girl, with a sequence of attributes that are inherited from Person
  - We identify that specific girl with the sequence of attributes: ***Nancy; 11 10th Av.,Washington, DC; 21; XCW553245***
- The 550 Maranello is an instance of Ferrari, with a sequence of attributes that are inherited from Car
  - We identify that specific Ferrari with the sequence of attributes: ***Ferrari; 550 Maranello; I LOVE F; Red***

➔ ***While inheriting, some attributes can become fixed***

The OO Process Model



Planning

Risk Analysis

Customer Communication

Customer Evaluation

Engineering, Construction & Release

identify candidate classes

look-up classes in library

extract classes if available

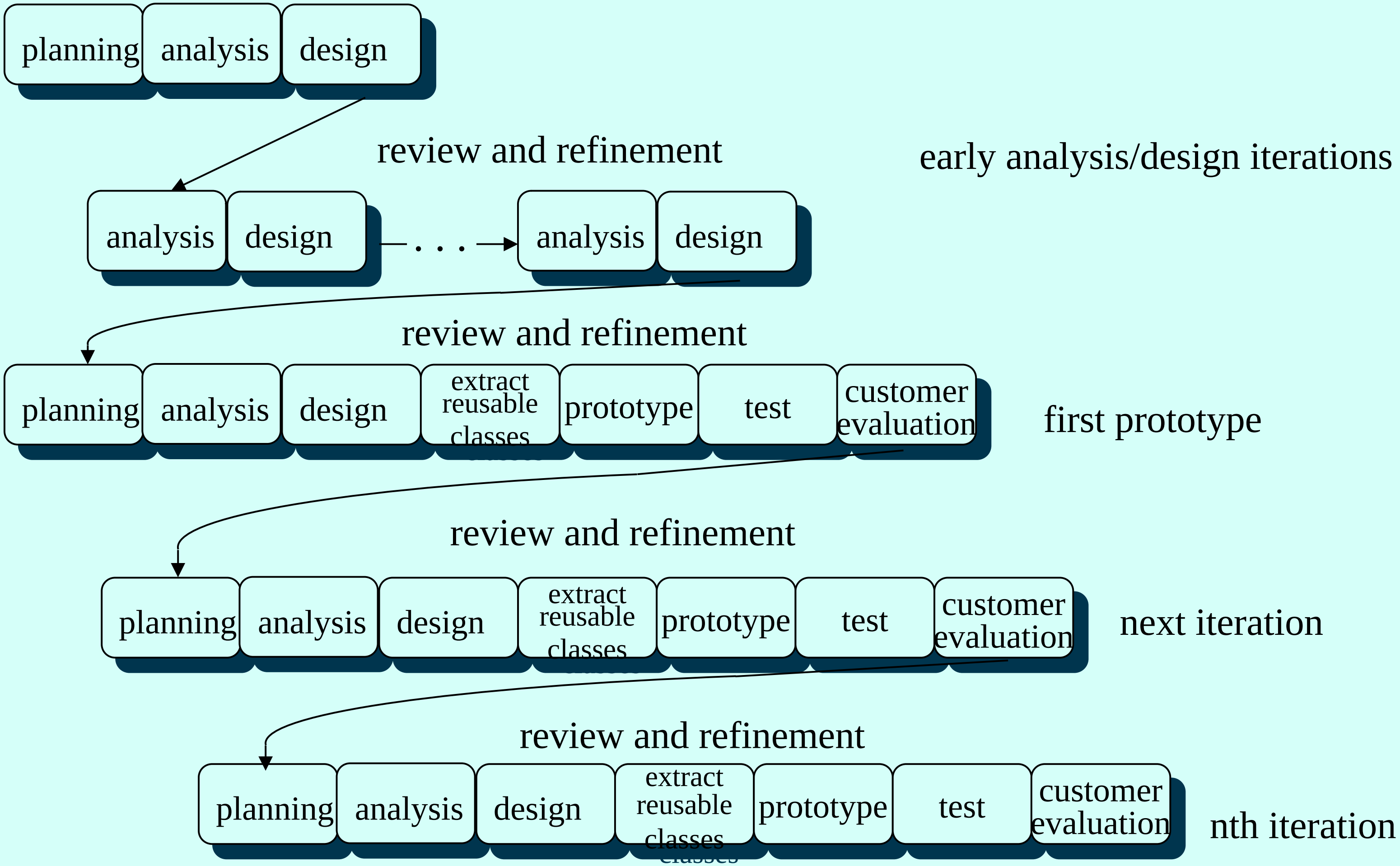
engineer classes if unavailable

put new classes in library

construct nth iteration of system

OO analysis
OO design
OO programming
OO testing

Typical Process for an Object-Oriented Project

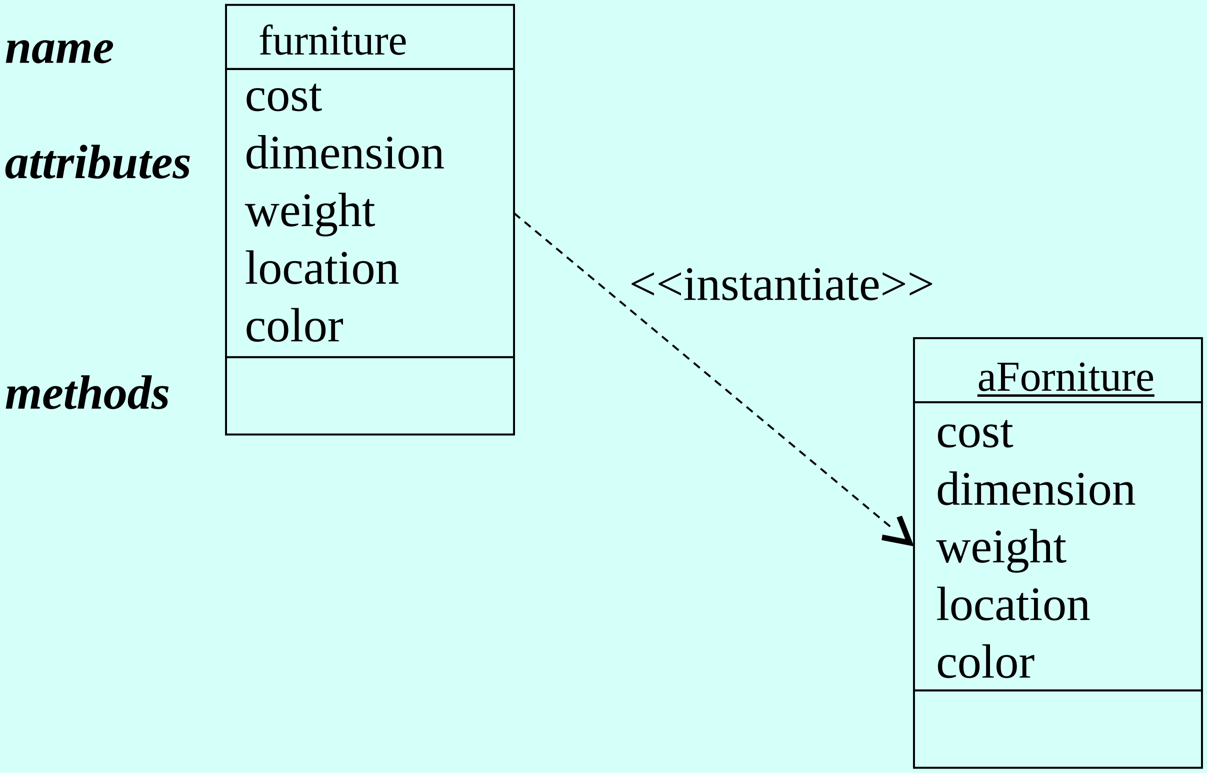


Let’s define some term ...

Classes

- A class is a collection of similar objects; a class often defined as:
  - template
  - generalized description
  - pattern
  - “blueprint” ... describing a collection of similar items
- A class identifies ***properties (attributes)*** that belong to all objects of the class and ***behaviors (operations)*** of all objects of the class
- Once a class of items is defined, a specific instance of the class can be defined

Instantiating Classes



name

attributes

methods

furniture

cost
dimension
weight
location
color

<<instantiate>>

aForniture

cost
dimension
weight
location
color

C++ CODE

```
class funiture{
 public:
    float cost;
    float dimension;
    char * location;
    char * color
     .
     .
 };
.
.
 void main(void){
    .
    .
    furniture aForniture;
   .
   .
 }
```

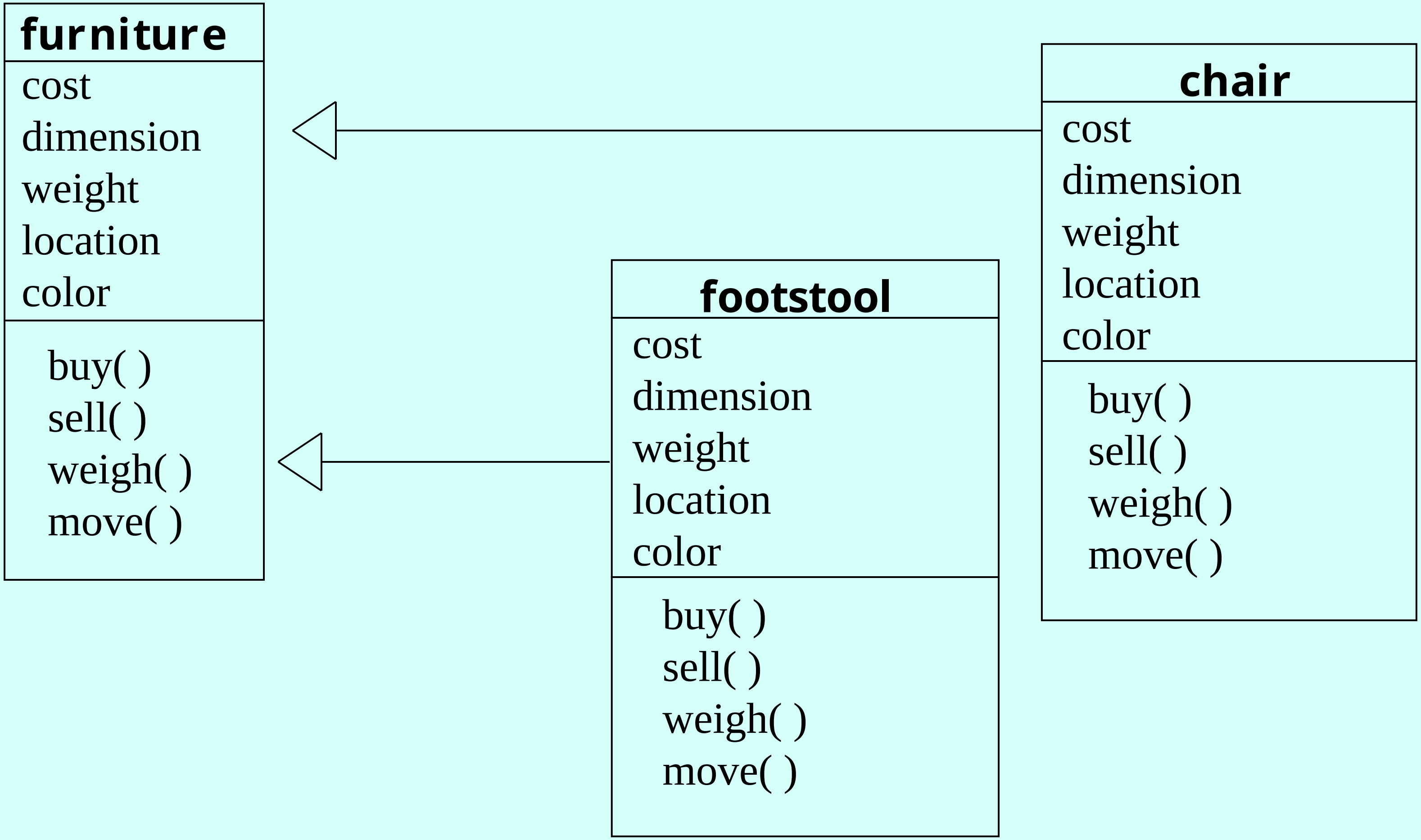
Operations (a.k.a. Services)

- An executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class
- Often textbook say that an operation is invoked via message passing
- The term "operation" has several synonym: Service, Function Entry (Concurrent Pascal), Member Function (C++), … *? Methods ?*

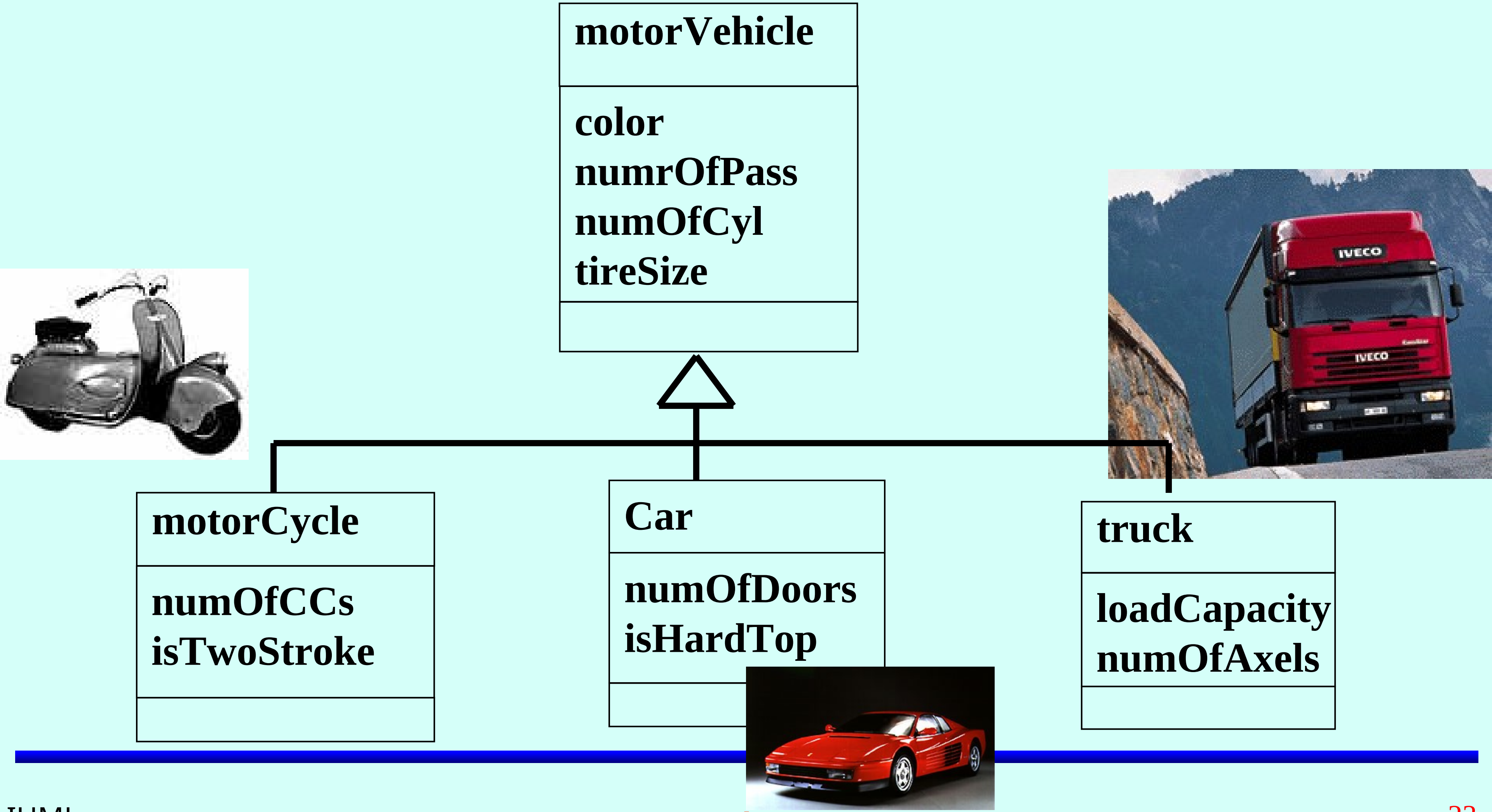
Inheritance

- Inheritance is the ability to **define classes that are extensions** of other classes with new and/or specialized attributes and methods
- For instance class Dog inherits from class Animal, meaning that Dog has (inherits) all the attributes and the methods of Animal, and can redefine some of them and add new ones
- People say: **Dog "is-a" Animal**, **Dog "extends" Animal**, **the class of Animals "contains" the class of Dogs**, **Animal "generalizes" Dog**, …

Class Inheritance



A Representation of Inheritance



motorVehicle

color
numrOfPass
numOfCyl
tireSize

motorCycle

numOfCCs
isTwoStroke

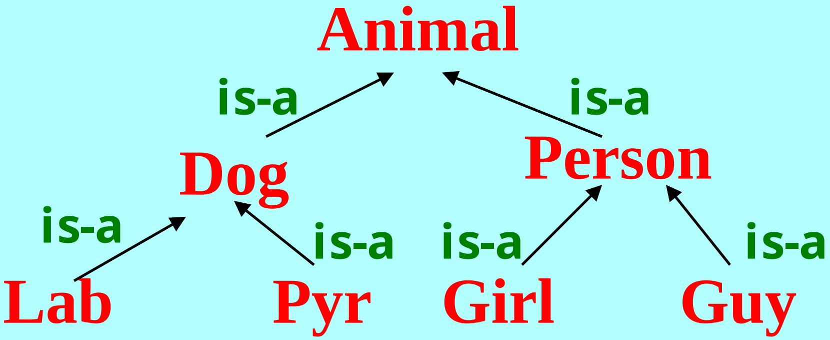
Car

numOfDoors
isHardTop

truck

loadCapacity
numOfAxels

Inheritance is NOT Instantiation



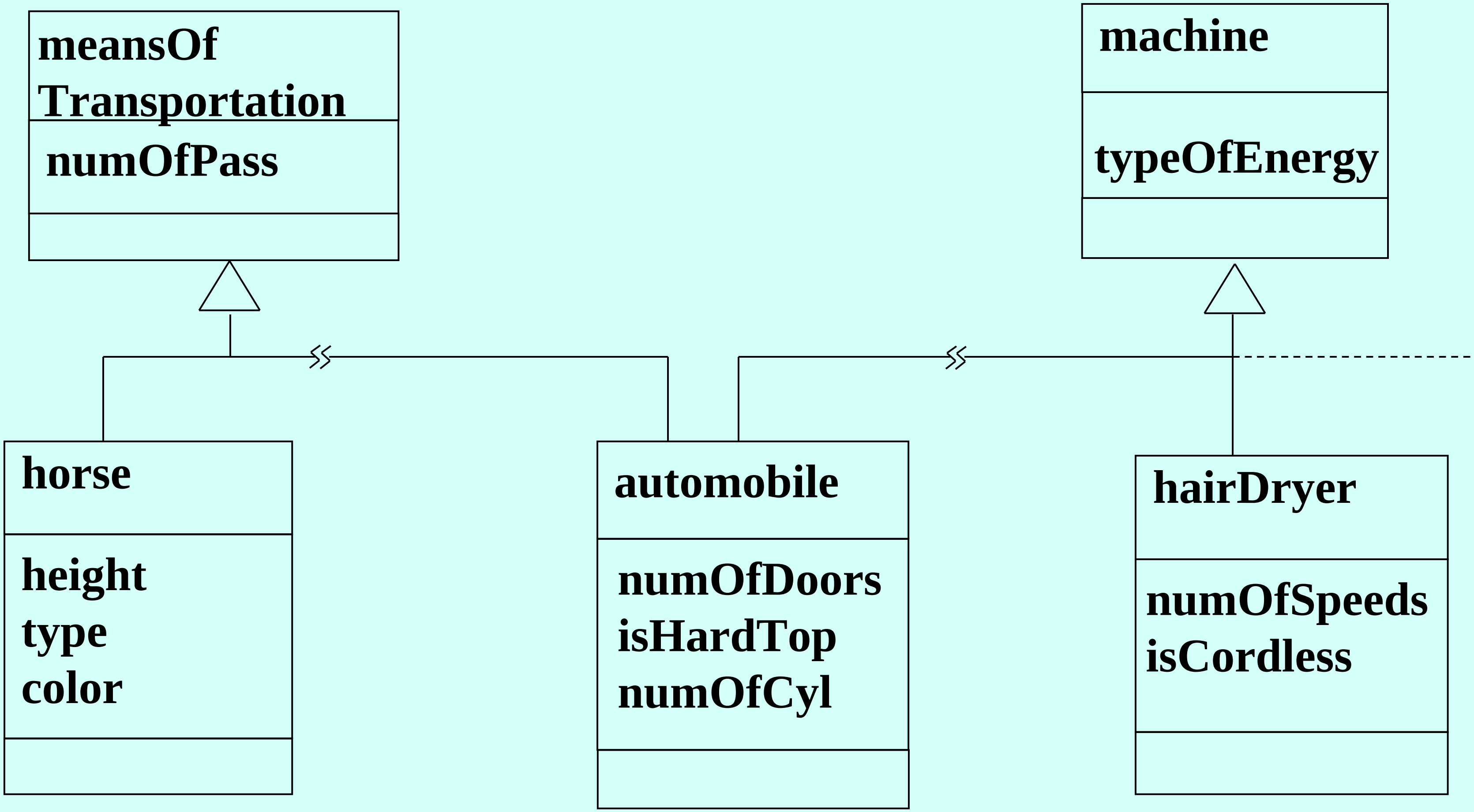
I Love Pluto!!!

I refer to Pluto, an instance of class Dog - I do not know and I do not care which sub-class of dog Pluto belongs to

I Love Pyrs!!!!

I refer to the class of Pyrs that is derived from the class of Dogs

Multiple Inheritance



meansOf Transportation

numOfPass

machine

typeOfEnergy

horse

height
type
color

automobile

numOfDoors
isHardTop
numOfCyl

hairDryer

numOfSpeeds
isCordless

Polymorphism

- Polymorphism is the ability to use the **same name** for methods performing operations of the “**same kind**” on different objects
- In Math there are several example of polymorphism:
  - **+ is used to sum any kind of number (Natural, Integers, Real, Complex, …) but also vectors and matrixes**
- Polymorphism help managing large set of classes with similar operations, without having to remember bizarre names (e.g., **`printf`**, **`fprintf`**, **`sprintf`**, …)

Various forms of Polymorphism

- **Ad-hoc polymorphism**, also called "**overloading**": several functions are defined with the same name but different parameters
  - `print(file)`, `print(string)`, `print(number)`
- **Generic polymorphism**: a general template defines a structure common to a set of classes / functions

```
template <class A> void swap(A &x, A &y) {
  A t=x; x=y; y=t; }
```

- **Inheritance polymorphism** or **overriding** …

Polymorphism: Overriding

```
class Graph{//base class
public: virtual void draw( ){
        cout<<“in base\n”; }
};

class LineGraph : public Graph
public: virtual void draw( ){
        cout<<“in LineGraph\n”; }
};

class PieChart : public Graph{
public:virtual void draw( ){
        cout<<“in piechart\n”; }
};
```

```
void DrawCorrect(Graph &t){
     t.draw( );
}
```

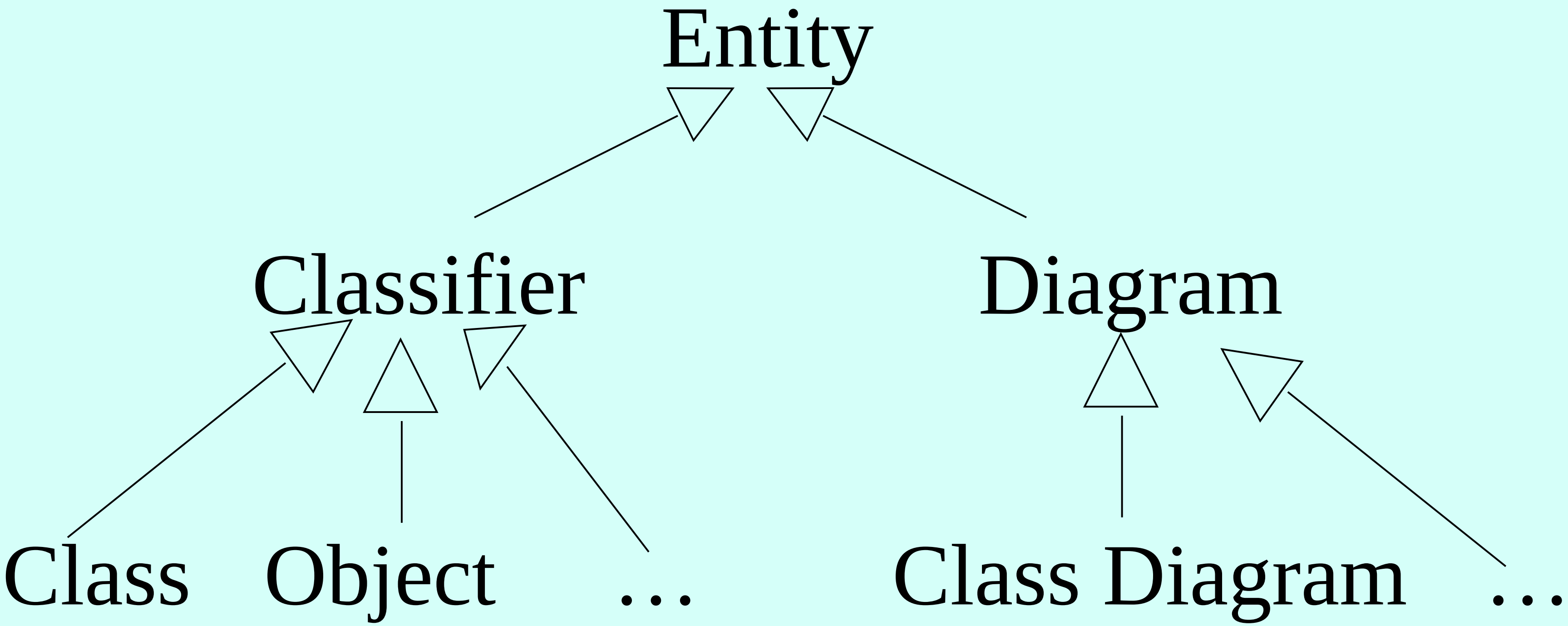
```
void main(void){
   LineGraph lg;
   PieChart pc;
   DrawCorrect(lg);
   DrawCorrect(pc);
   Graph *list[10];
   int i;
   for(i=1;i<10;i++)
     list[i]= … ;
   for(i=1;i<10;i++)
     DrawCorrect(*list[i]);
}
```

UML in Some Details

UML

- The Unified Modeling Language tries to integrate older approaches
- Developed by Rational (CASE tool)
  - they hired Booch, Rumbaugh, Jacobsen
- Standardized by the OMG (Object management group)
- Supported by almost all OO CASE tools … but with some limitations …
- Currently it is at version 1.3 …

UML has many entities …



Entity

Classifier

Diagram

Class

Object

…

Class Diagram

…

UML Classifiers

- Class
- Interface
- Datatype
- Component
- Node
- Use Case
- Subsystem
- …

UML has 9+ kinds of diagrams

❶ Class Diagram

❷ Object Diagram

❸ Component Diagram

❹ Deployment Diagram

Structural Diagrams

❺ Use Case Diagram

❻ Sequence Diagram

❼ Collaboration Diagram

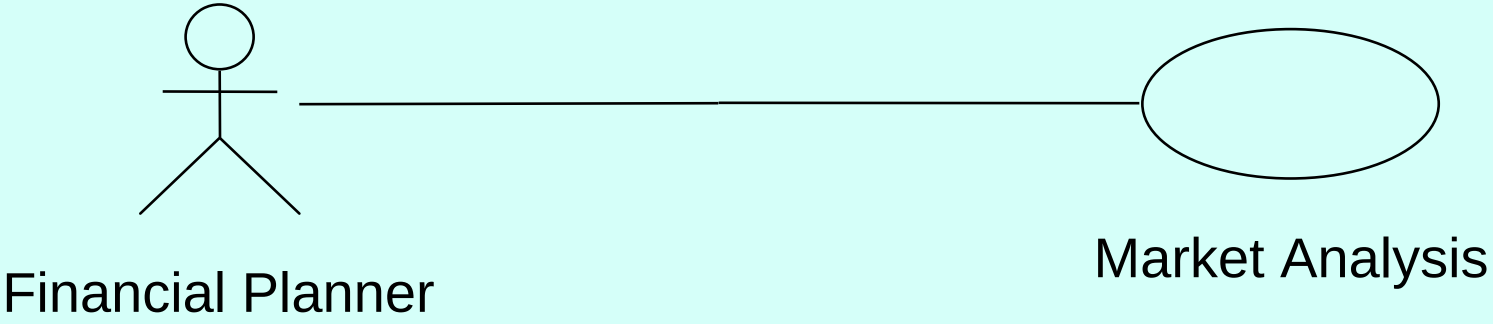
❽ Statechart Diagram

❾ Activity Diagram

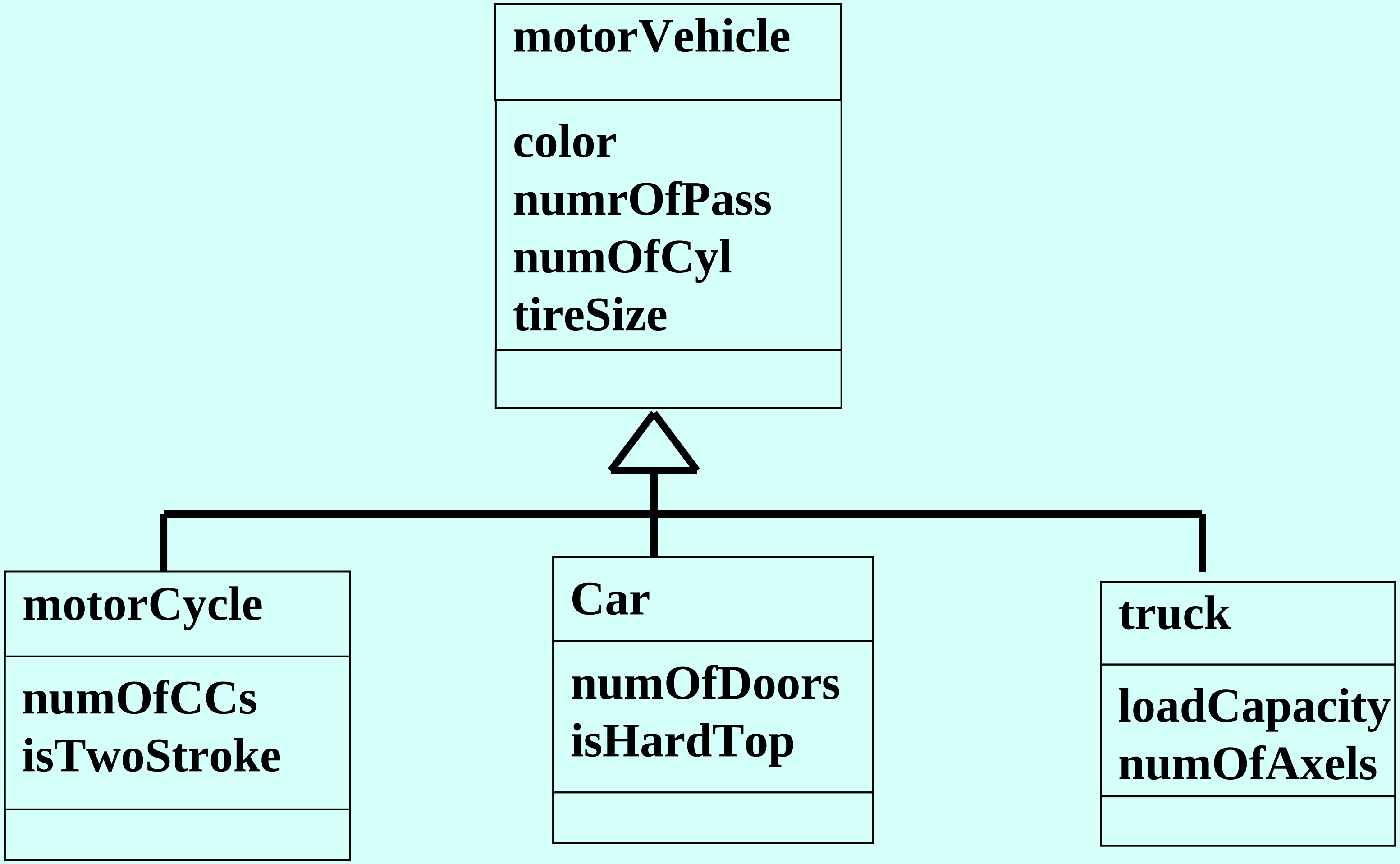
Behavioral Diagrams

Use case diagrams

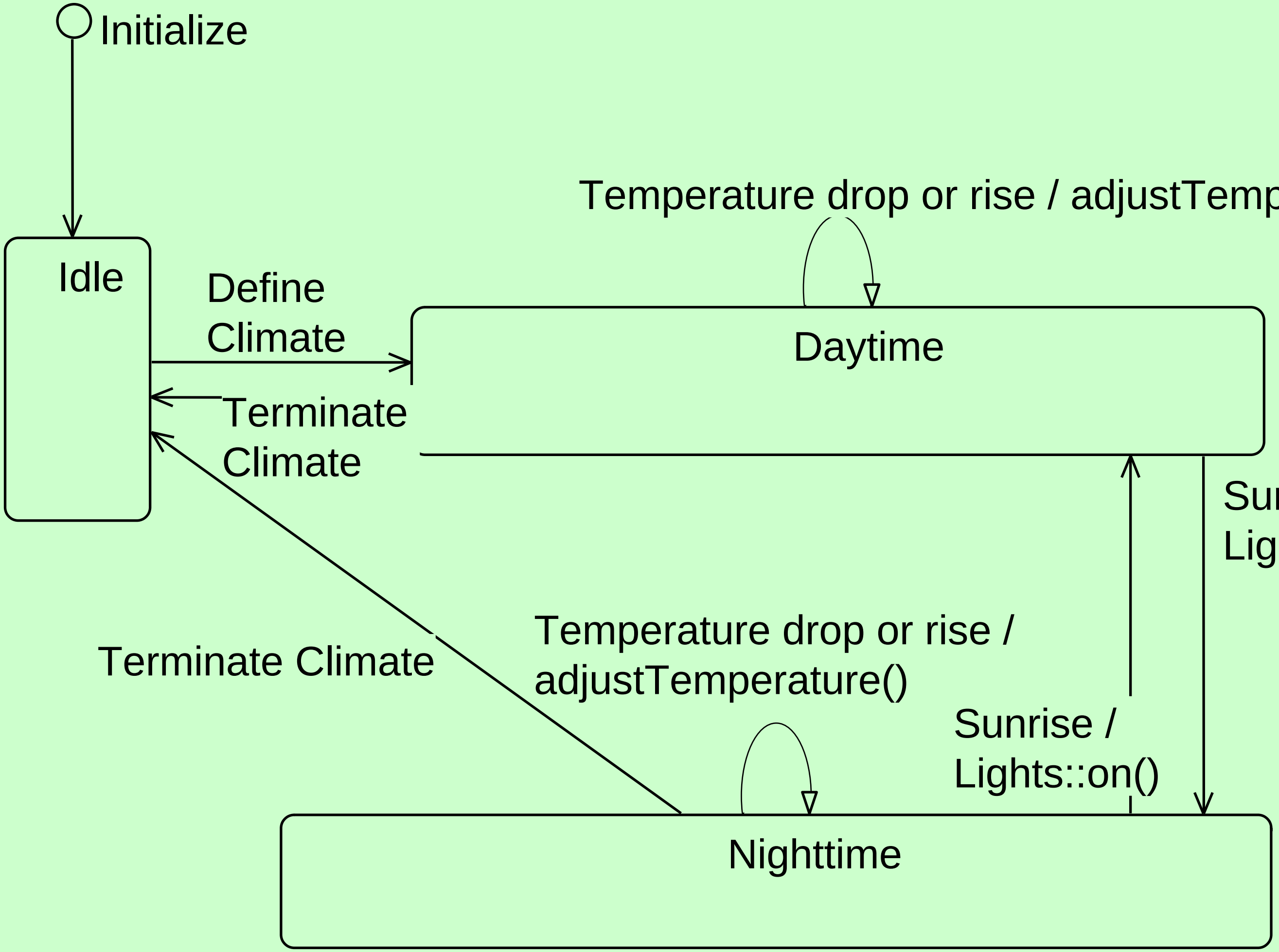
- Requirements/ early analysis



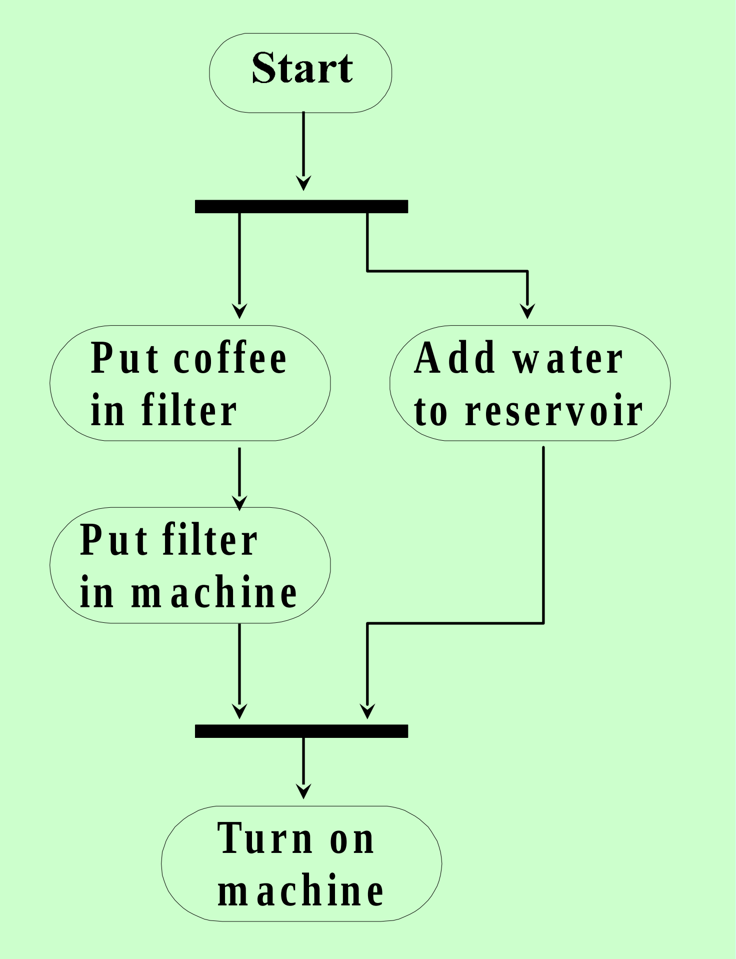
Class diagrams (motor vehicle)



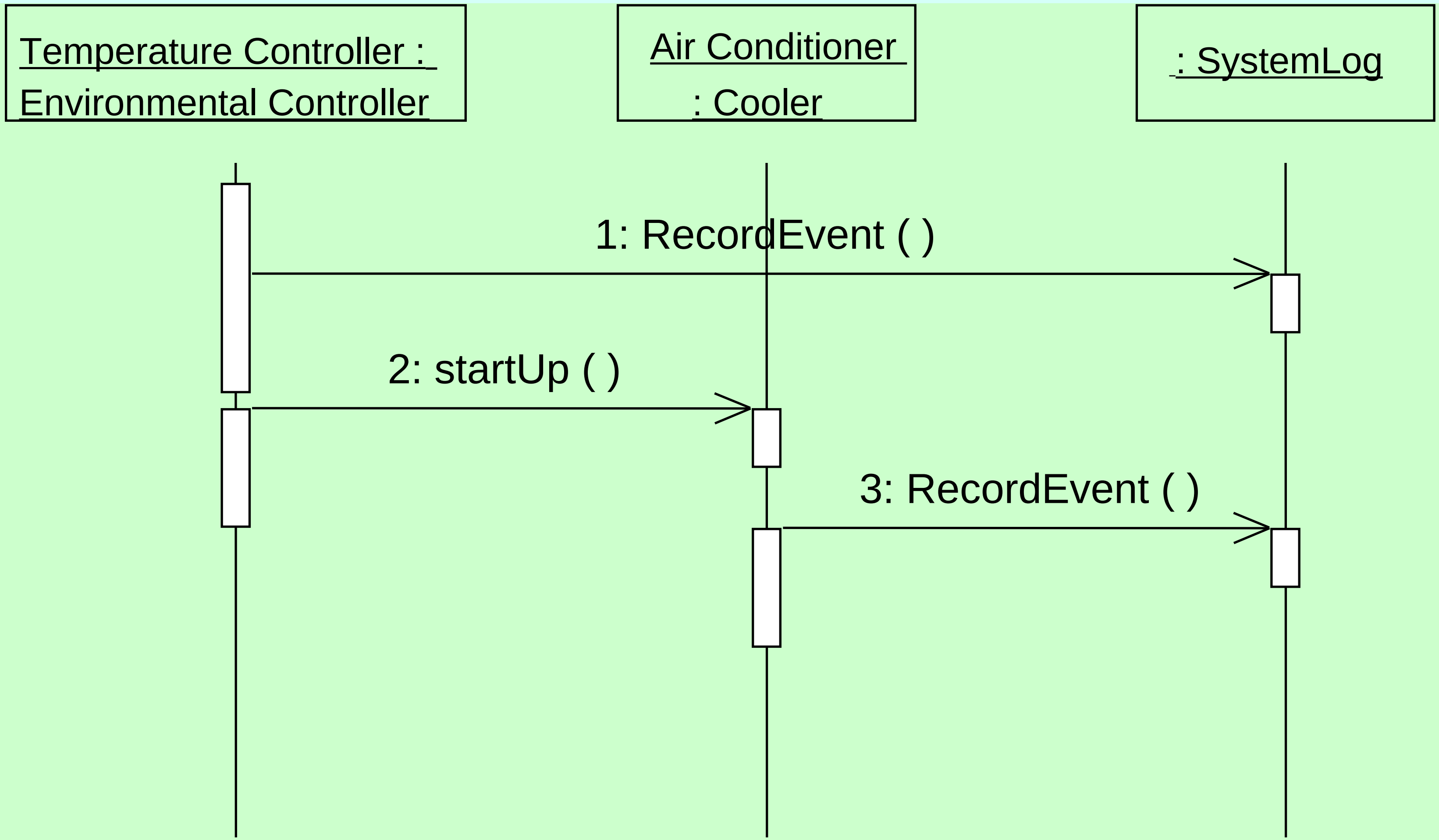
Statechart diagrams (air cond.)



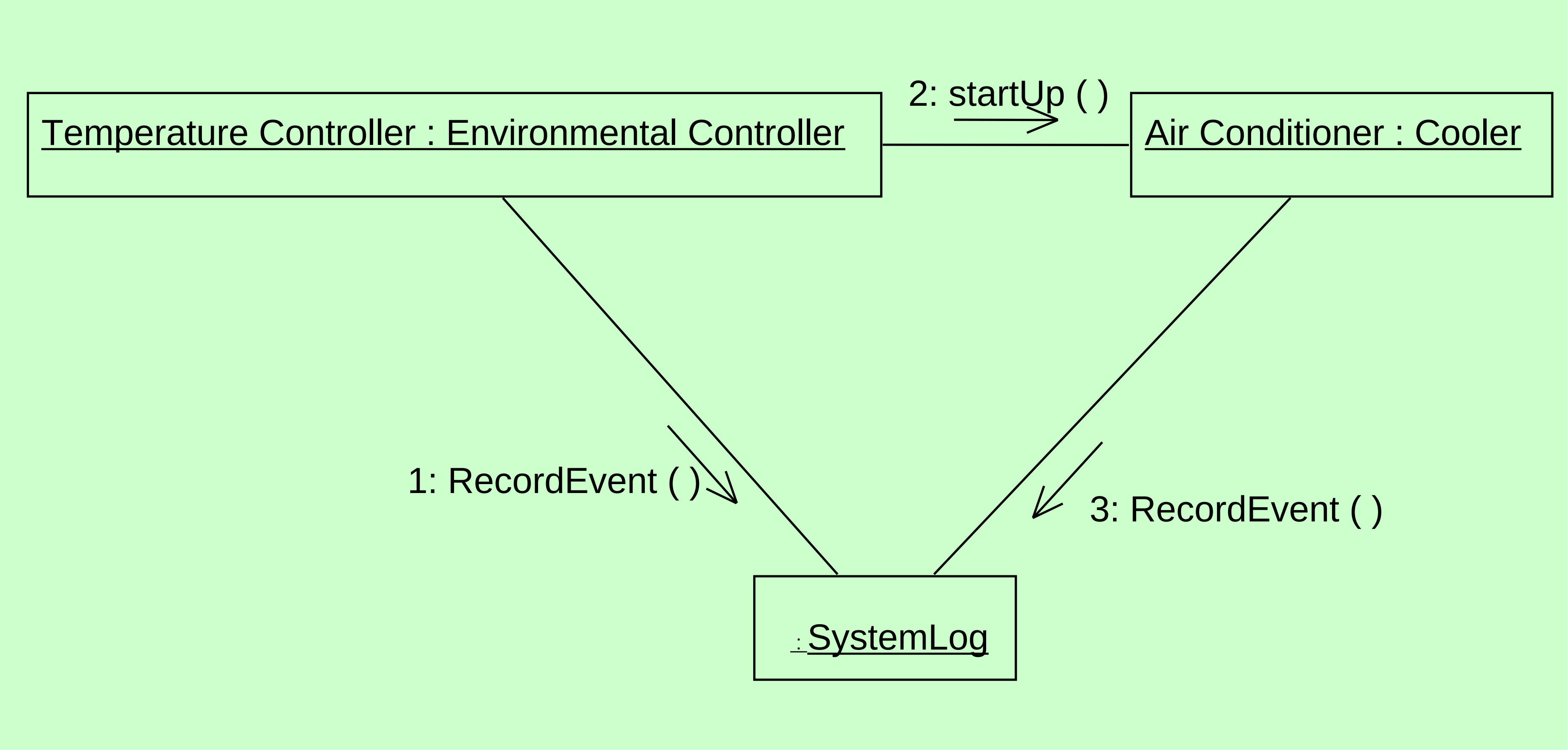
Activity diagrams (coffee machine)



Sequence diagrams (again air cond.)



Collaboration diagrams (again …)



Object Oriented Concept Modeling

Goals of OO Concept Modeling

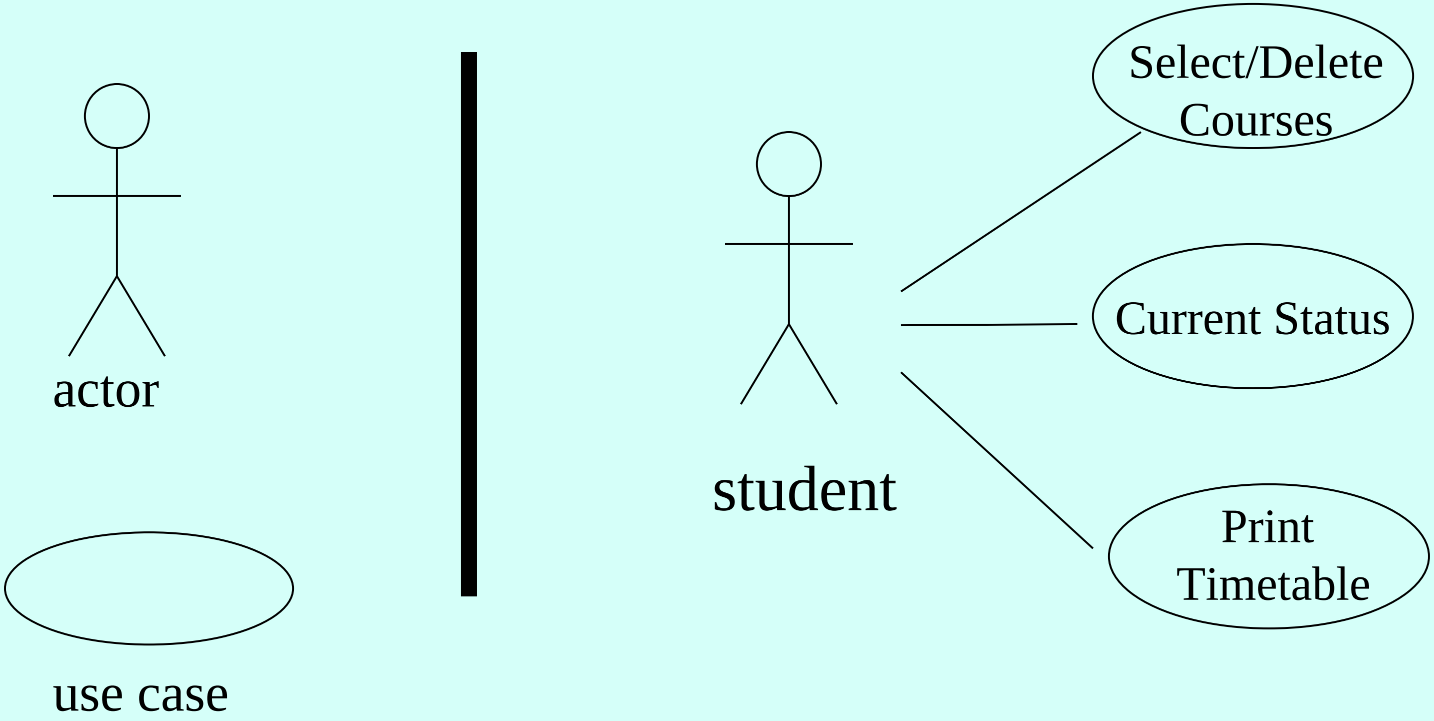
- Understanding the operational context of the system
  - ☑ OO Context Analysis
- Understanding the effective requirements of the system
  - ☑ OO Requirement Analysis

➔ *Sometimes people refer to this phase only as Requirement Analysis, but they do mean both activities*

Use Cases for OO Concept Modeling

- A Use Case Description is a scenario that describes a “thread of usage” for a system
- A Use Case Description includes:
  - A ***Diagram***, with ***actors*** representing roles people or devices play as the system functions and ***use cases,*** that is, cases of use of the system
  - A ***Textual Description*** sequencing the activities

Use Case Diagram - Early Example



actor

use case

student

Select/Delete Courses

Current Status

Print Timetable

What is a Use Case?

- Typical interaction between actors and system
- Process that satisfies a user’s need
- Describes a scenario -i.e., how the system is used
- Example: *for a Word processor*
  - ***Make some text bold***
  - ***Create an index***
  - ***Delete a word***

Facts on Use Cases

- Granularity: Small or large
- Often: Use cases capture user-visible function
- Always: Use cases achieve a discrete goal
- Always: Use cases describe externally required functionality

When and how

- Contest capture - first thing to do
- Use case: Every discrete thing your customer wants to do with the system
  - give it a name
  - describe it shortly (few paragraphs)
  - ✓ you will add details later

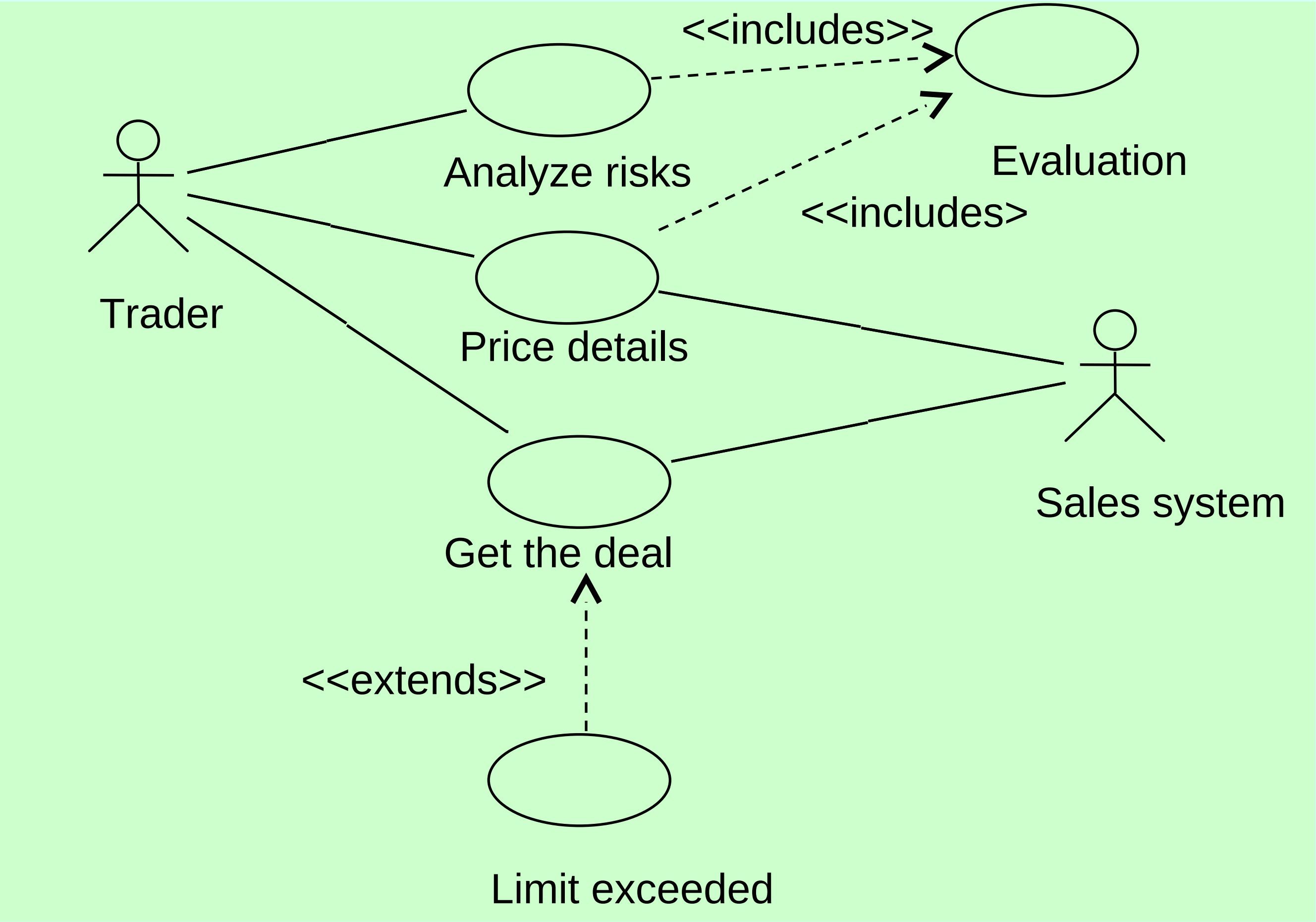
Developing a Use Case

- What are the main tasks or functions that are performed by the actor?
- What system information will the the actor acquire, produce or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Example

- Suppose we want to model the NYSE …
  - ☑ There are traders ...
  - ☑ … and Sales systems

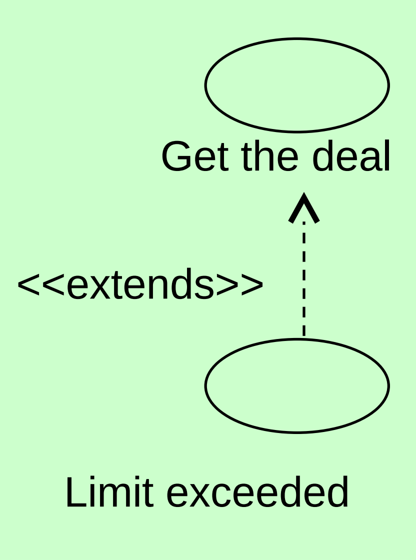
Use case diagram



Actors

- *Role* that a user plays with respect to the system
- Actors carry out use cases
  - look for actors, then their use cases
- Actors do not need to be humans!
- Actors can get value from the use case or participate in it

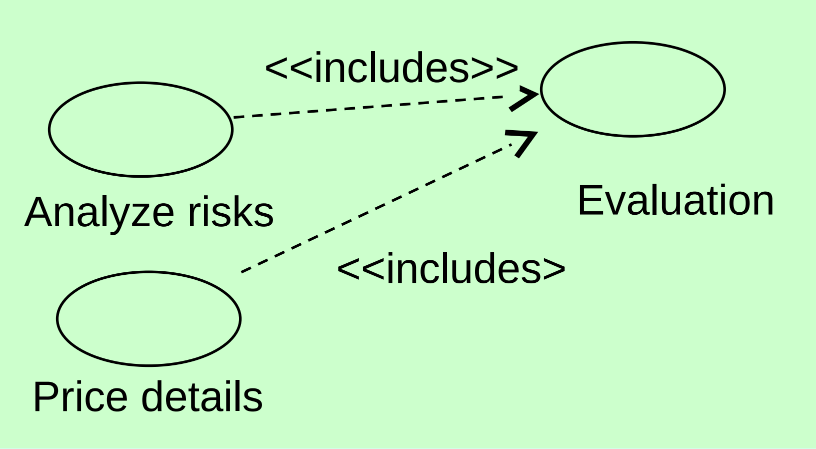
Extends relationship



- Extends: One use case is similar to another but does a bit more
  - Capture the simple, normal use case first
  - For every step ask
    - what could go wrong
    - how might this work out differently
  - Plot every variation as an extension of the use case

Includes relationship

- Used when a chunk of behavior is similar *across* more than one use case



- Avoids copy-and-paste of parts of use case descriptions

Comparing extends/includes

- Different intent
  - extends
    - same actor performs use case and all extensions
    - actor is linked to “base” case
  - includes
    - often no actor associated with the common use case
    - different actors for “caller” cases possible

Textual description

- Generic, step-by-step written description of the interactions between the actor(s) and a use case
- Clear, precise, short descriptions

Example use case description

- **Use Case: Get the deal**

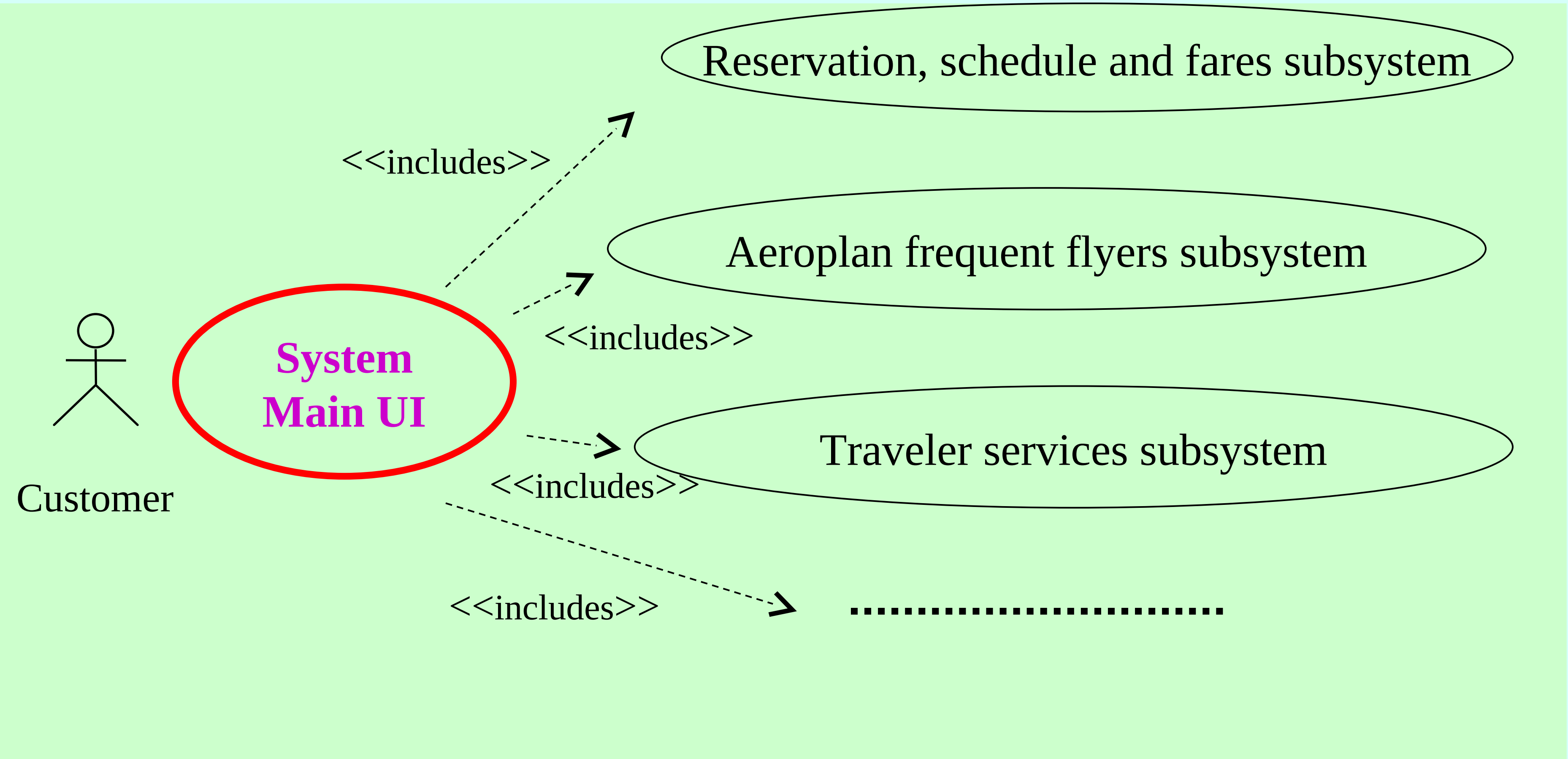
1. Enter the user name & bank account
2. Check that they are valid
3. Enter number of shares to buy & share ID
4. Determine price
5. Check limit
6. Send order to NYSE
7. Store confirmation number

Notice that ...

- Not all cases of use have been listed (each diagram provides a partial view)
- The include relation support the factorization of the common specs of the system
- There is NOT a 1:1 correspondence between screens and ovals

☠ BEWARE ...

The KILLER



Reservation, schedule and fares subsystem

<<includes>>

Aeroplan frequent flyers subsystem

<<includes>>

System Main UI

Customer

Traveler services subsystem

<<includes>>

<<includes>>

....

Textual Description (only for the frequent flyer use case)

The user enters the subsystem to gain more information about its frequent flyer status. Inside the subsystem, the user can access **(a)** *general information about the frequent filer program -the reward schema, how to enroll, how to get miles with partner companies, and* **(b)** *specific information on her/his status, such as the miles earned, the status level. The user can also update her/his address.*

NOTICE: This is a very different format!!!

Proposed Exercise (1)

Develop an OO Concept Model for a system supporting the reservation and scheduling for taxi drivers.

Proposed Exercise (2) OOCM for *Stocks Trading Service*

- By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the user to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
- The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a particular stock is no longer available.

Proposed Exercise (3)
OOCM for *Network Printing Service*

- There is a super high-resolution colour laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc…). In addition, users can monitor their own print jobs and delay or delete jobs as they see fit.
- To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.

Proposed Exercise (4)
OOCM for *Component Brokerage System*

- This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.

- Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.

- Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.

Proposed Exercise (5)
OOCM for *Bug Tracking System*

- Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
- An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.

Not covered

- Generalization in Use Cases
- Generalization in Actors
- Presence of extension points

Object Oriented Analysis

OOA - Content

- Structure of OO Analysis
- Extraction of Classes
- Representing classes in class diagrams
- Associations
- Roles
- Advanced Stuff on Associations
- Classes vs. Objects
- Attributes
- Operations
- Aggregation
- Inheritance
- When to use class diagrams

OOA- A Generic View

- extract candidate classes
- establish basic class relationships
- define a class hierarchy
- identify attributes for each class
- specify methods that service the attributes
- indicate how classes/objects are related
- build a behavioral model
- iterate on the first five steps

Extraction of Classes

- Normal sequence: Get the deal

  1. Enter the **user** name & **bank account**
  2. Check that they are valid
  3. Enter number of shares to buy & **share ID**
  4. Determine price
  5. Check limit
  6. Send order to **NYSE**
  7. Store **confirmation number**

Another Example

- Extract the classes from the previously discussed Lufthansa web site.

Extraction of Classes in the Lufthansa example

The **user** *enters the subsystem to gain more information about its* **frequent flyer status***. Inside the subsystem, the user can access (a) general information about the* **frequent flyer program** *-the* **reward schema***, how to enroll, how to get* **miles** *with* **partner companies***, and (b) specific information on her/his* **status***, such as the* **miles earned***, the* **status level***. The user can also update her/his* **address***.*

Class diagram

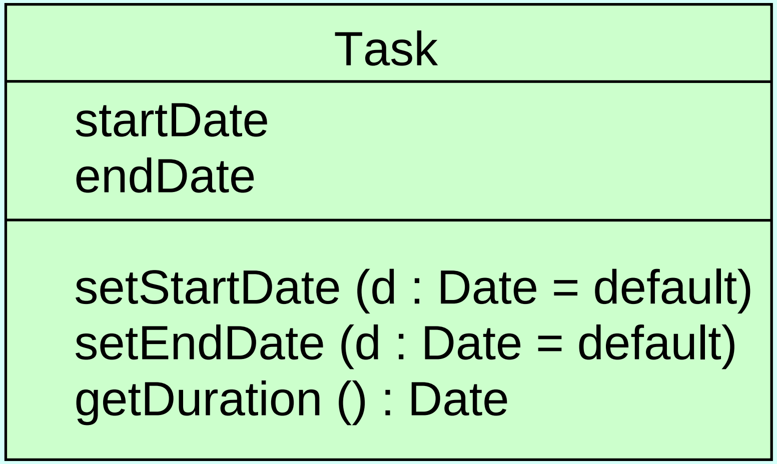
- Central for OO modeling
- Shows static structure of the system
  - Types of objects
  - Relationships
    - Association
    - Subtypes
    - Dependency

We Have 3 Perspectives

- Conceptual (OOA)
  - ☑ Shows concepts of the domain
  - ☑ Independent of implementation
- Specification (OOD)
  - ☑ General structure of the running system
  - ☑ Interfaces of software (types)
- Implementation (OOP)
  - ☑ Details of the implementation
  - ☑ Most often the only used

A Class

- Set of objects
- Defines
  - name
  - attributes
  - operations



Task

startDate
endDate

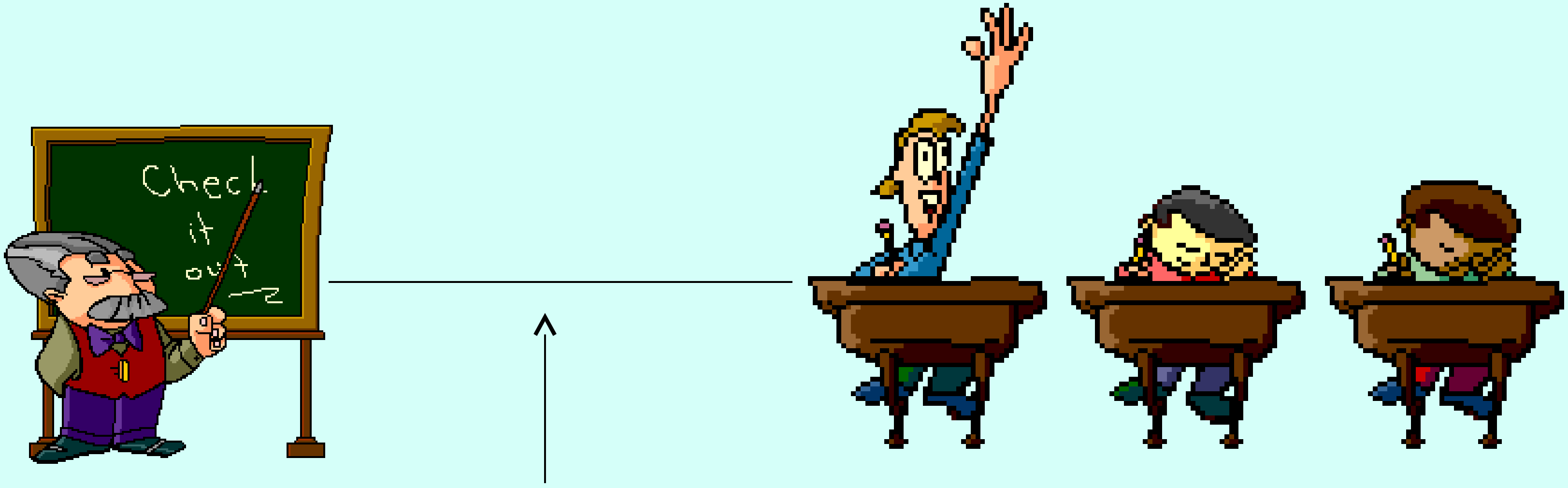
setStartDate (d : Date = default)
setEndDate (d : Date = default)
getDuration () : Date

Class versus type

- ✓ Type
  - → protocol understood by an object
  - → set of operations that are used
- ✓ Class
  - → implementation oriented construct
  - → implements one or more types
- *In Java a type is an interface, in C++ a type is an abstract class*
- *UML 1.3 has the <<type>> stereotype*

Association

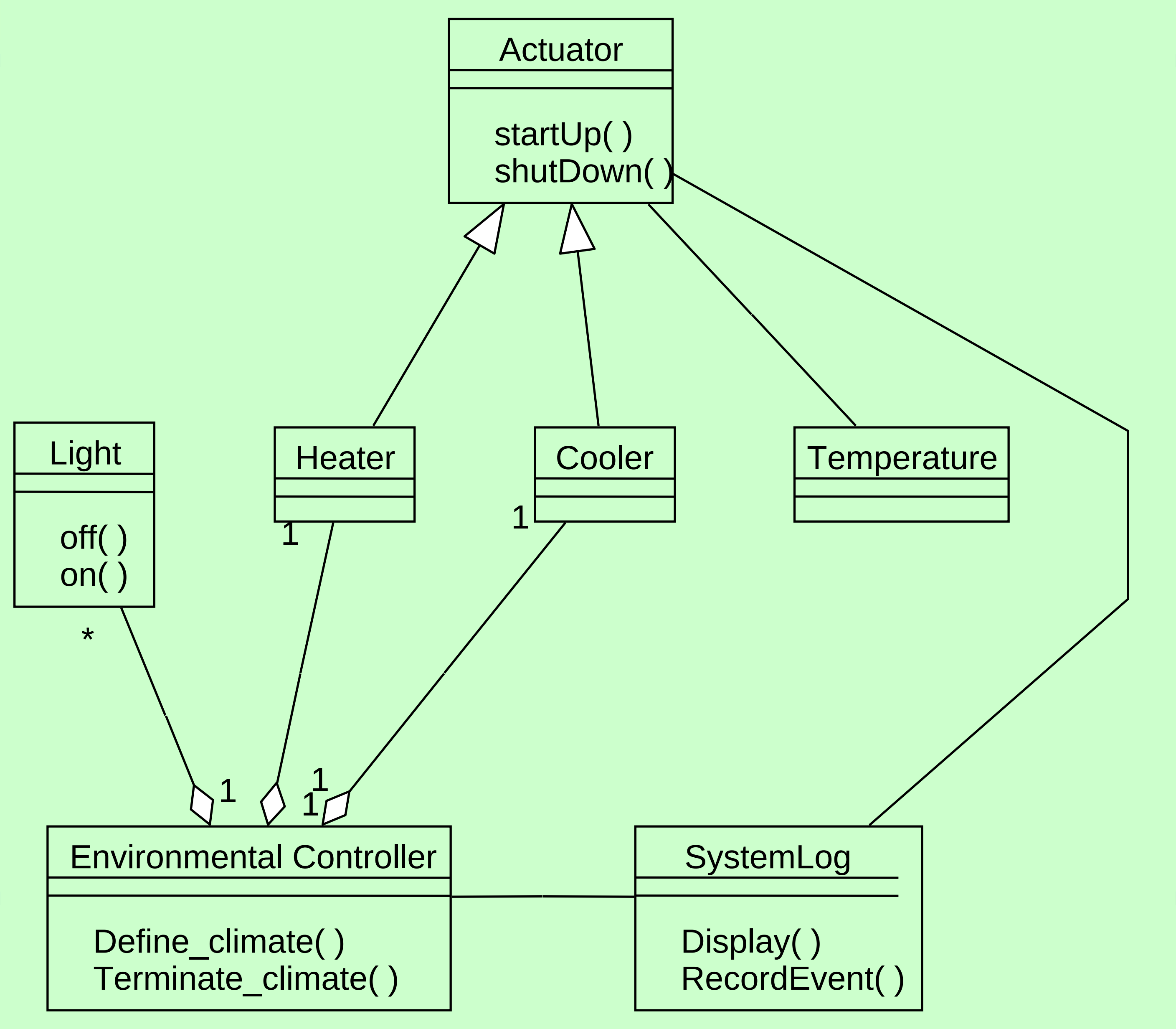
- Relationship between instances of classes
  - A student is registered for a course
  - A professor is teaching the course



Association

Class diagram example

Back to the example of the air cond. system



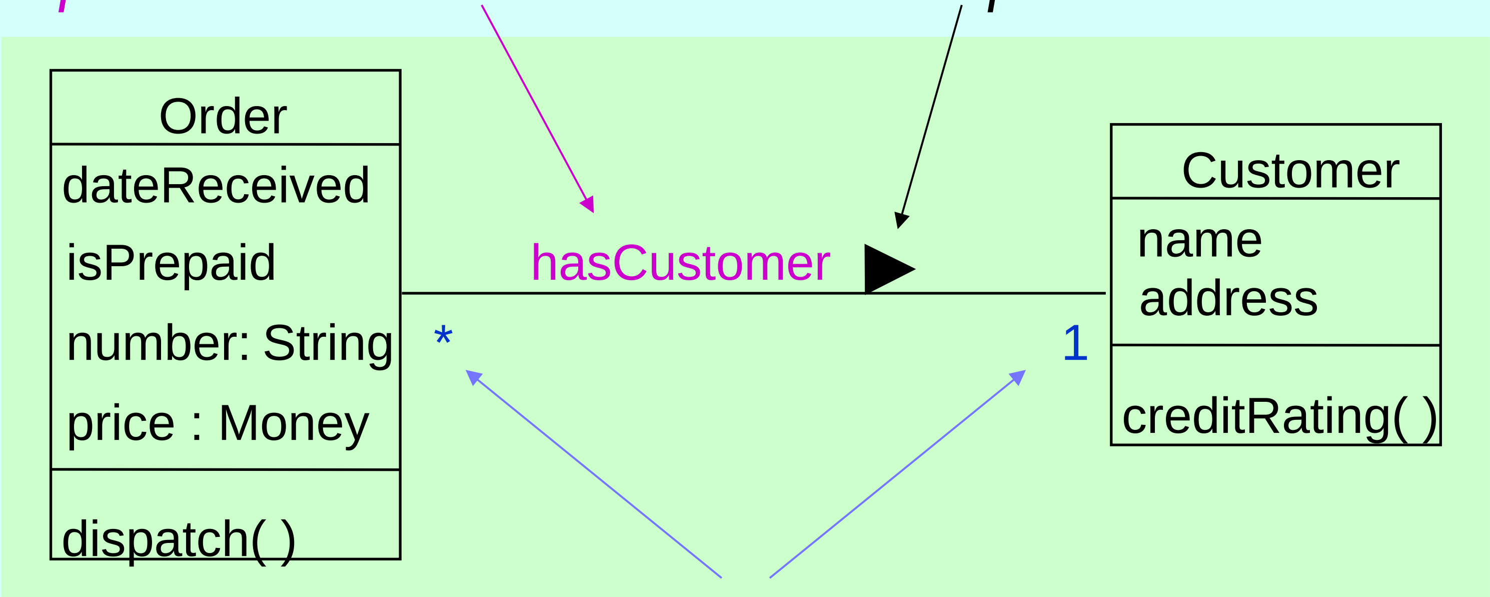
Classes and Diagrams

- One class can be part of several diagrams
- Diagrams shall illustrate specific aspects
  - – Not too many classes
  - – Not too many associations
  - – Hide irrelevant attributes/operations
- ***Several iterations are needed to create a "proper" diagram***

Association: Relationship between classes

Name of the Association

Direction of the Association



Order

dateReceived

isPrepaid

number: String

price : Money

dispatch( )

hasCustomer ▶

*

1

Customer

name

address

creditRating( )

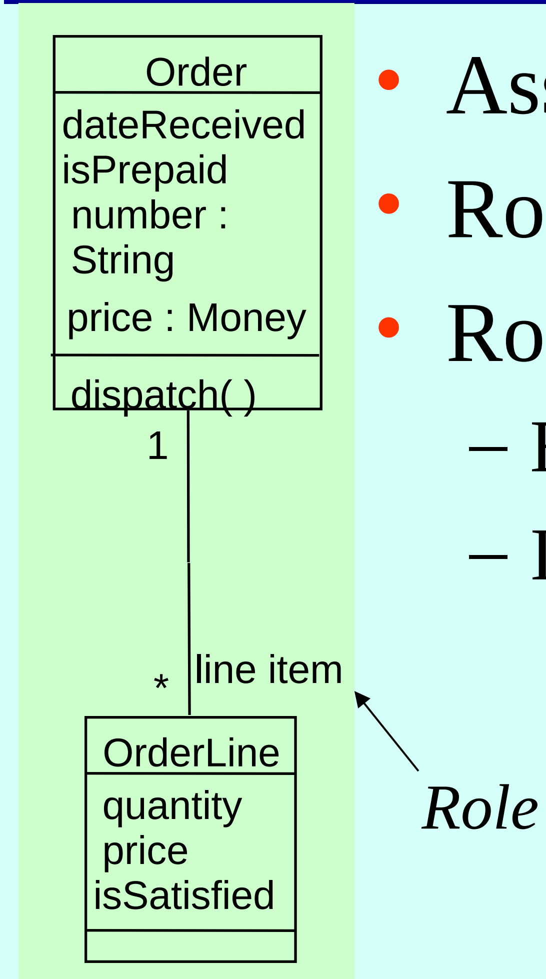
Multiplicity of each end

An order comes from one customer: a customer may make several orders

Naming associations

- Avoid meaningless names
  - associated_with
  - has
  - is_related_to
- Name is often a verb phrase
  - has_part
  - is_contained_in

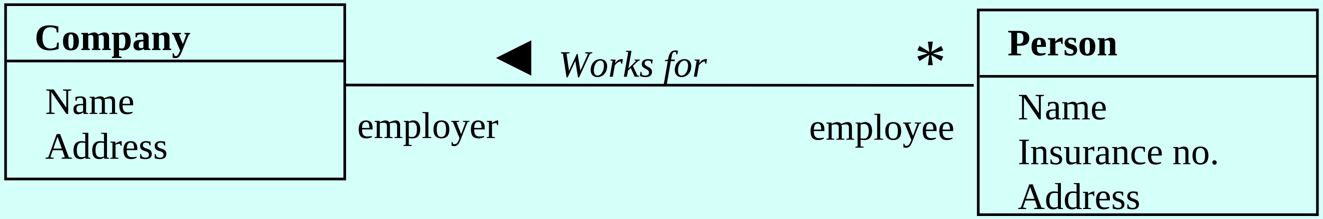
Roles



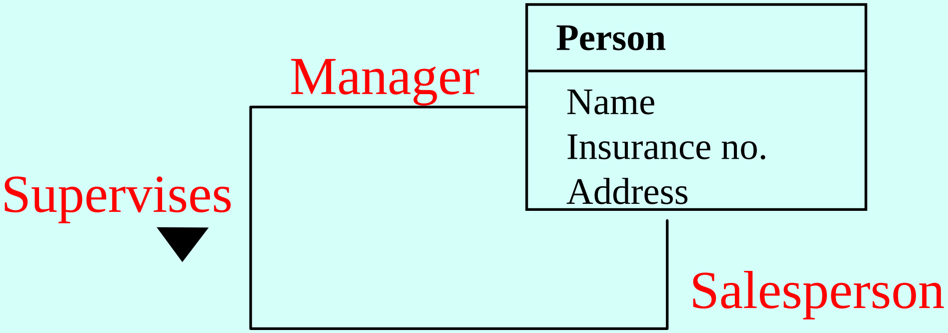
- Association has two roles
- Role is a direction on the association
- Role
  - Explicit labeled
  - Implicitly named after the target class

Role names

- A Role identifies one end of an association

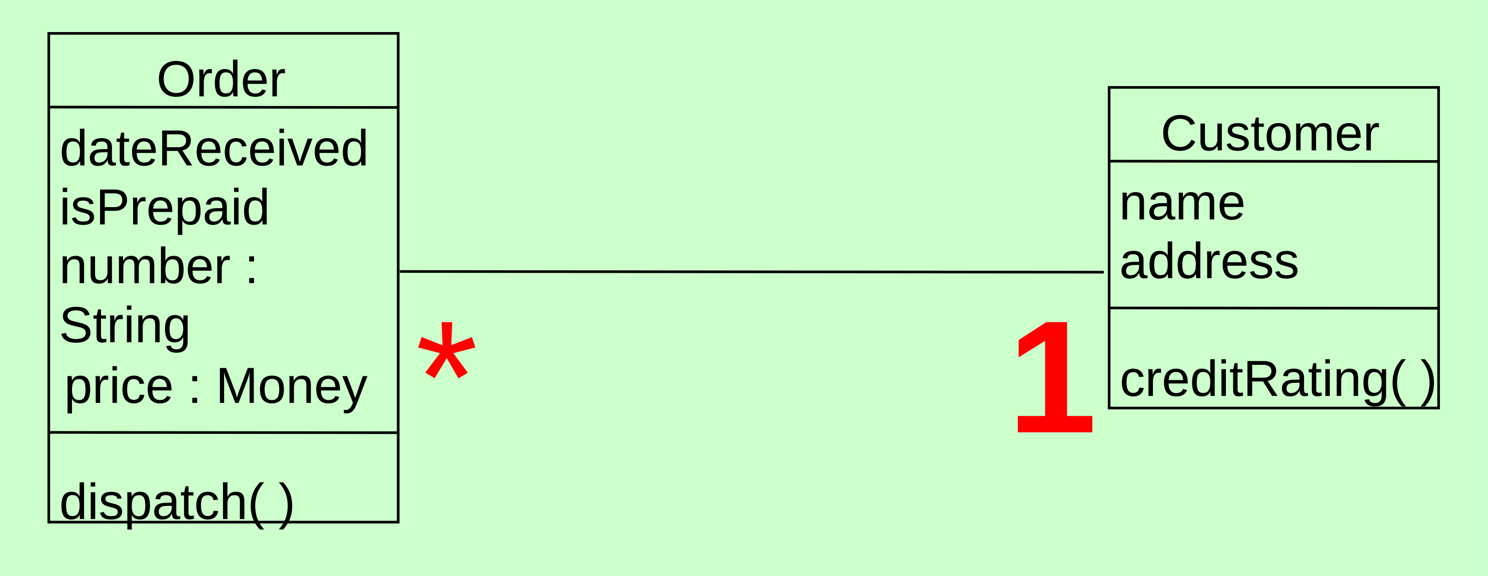


- Role name is obligatory for associations between objects of the same class



Multiplicity

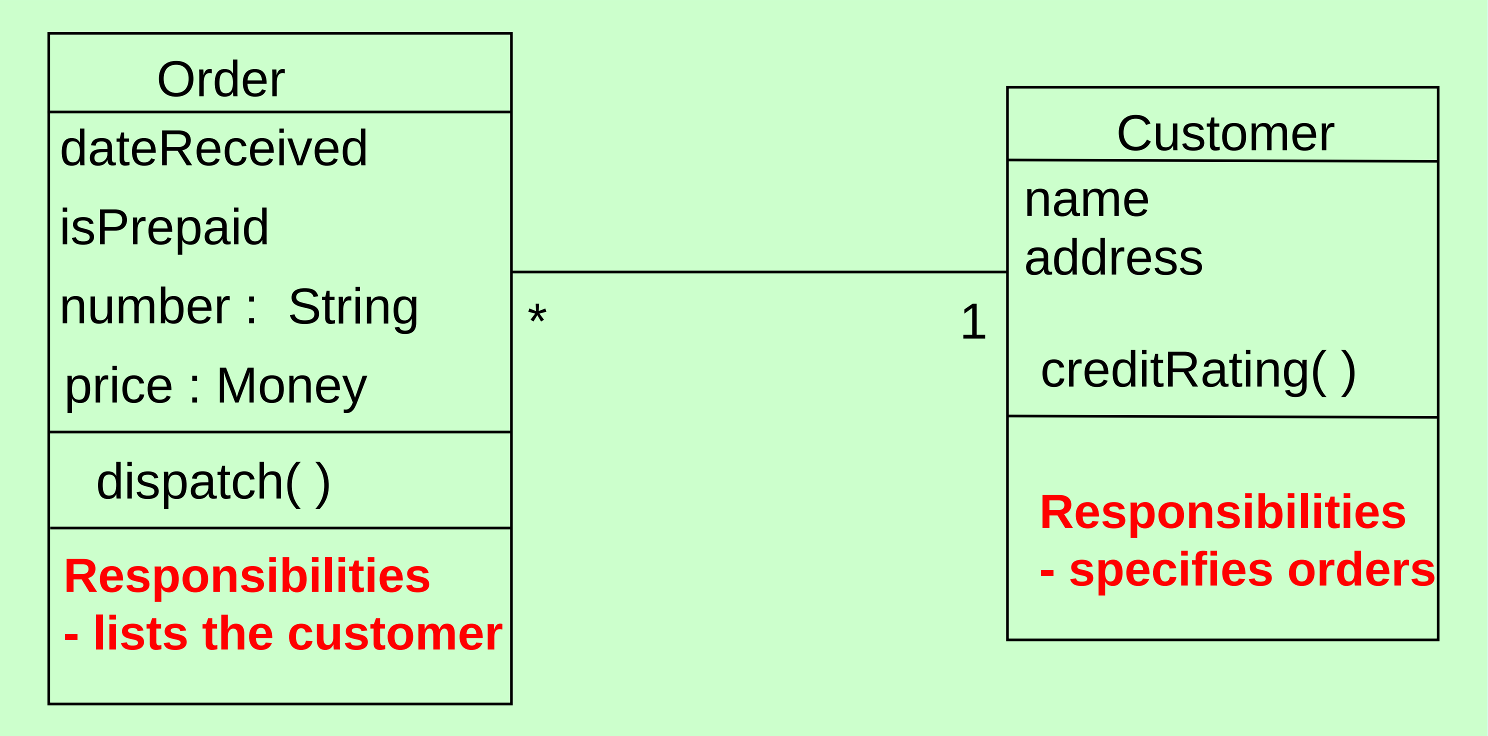
- Indicates how many object can participate in the relationship



Multiplicity (2)

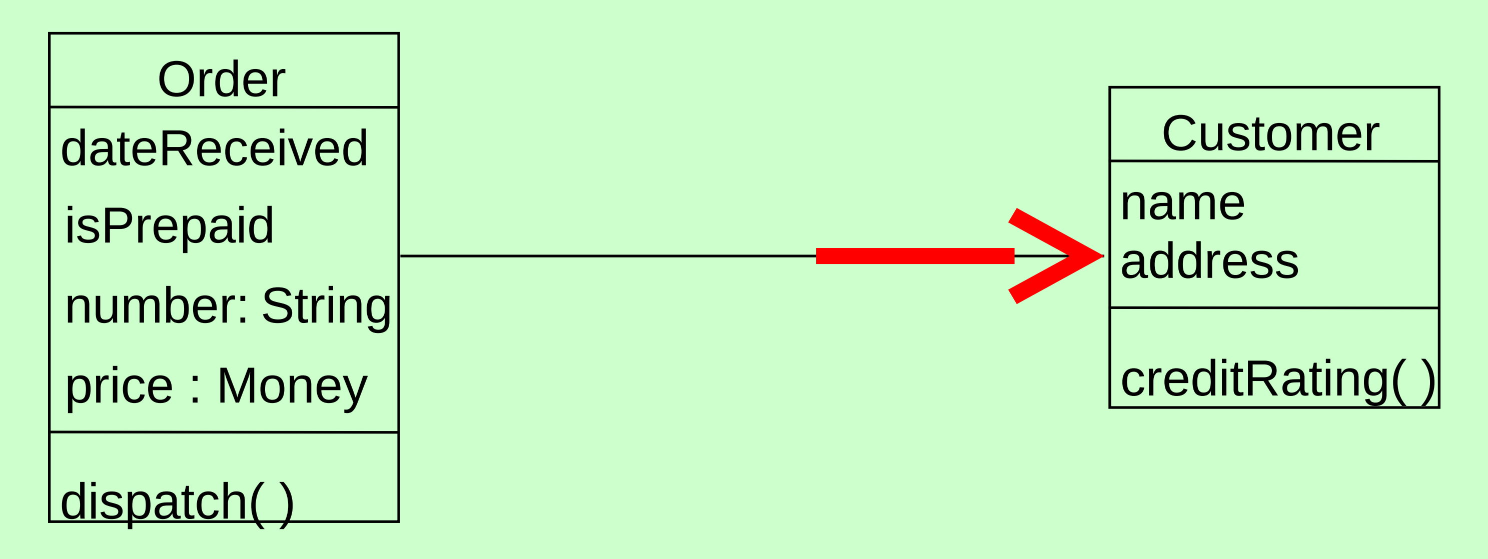
- *: 0..infinity
- 1: 1..1
- 0..1
- 1..100
- 2,4,5

Responsibilities



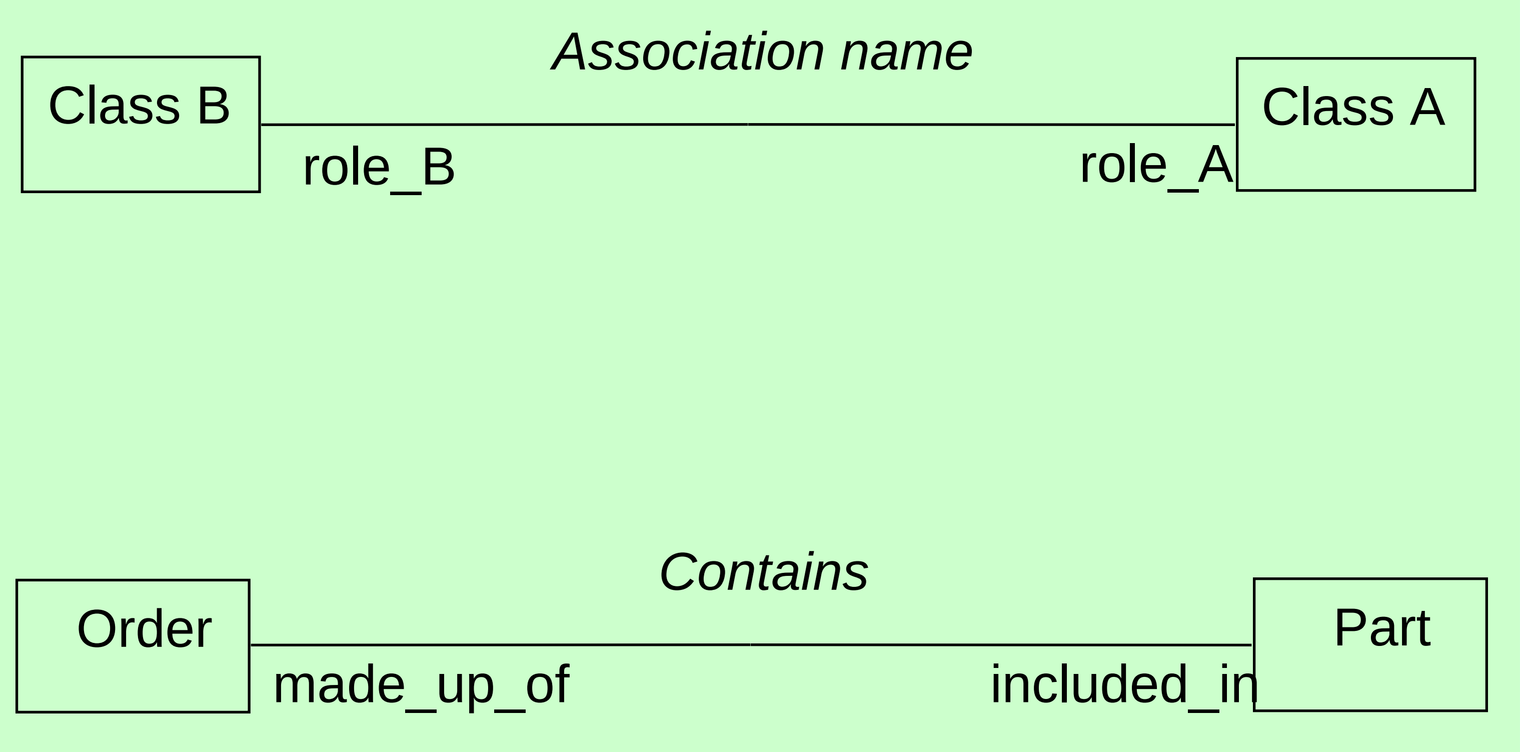
- The Customer specifies the Orders
- The Orders list the Customer

Navigability - Indicated by Arrow

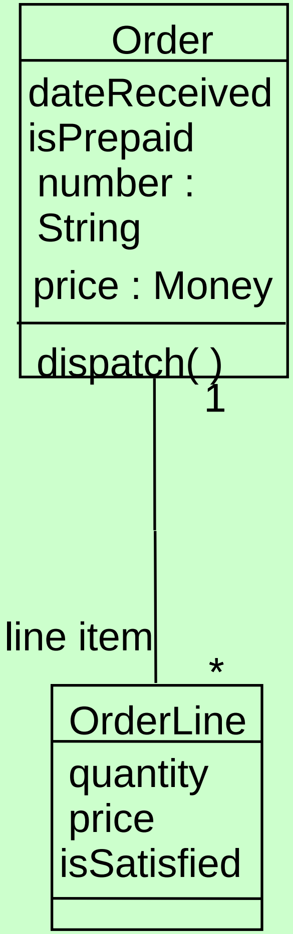


- Order has to be able to determine the Customer
- Customer does not know all Orders
- Bi-directional association: Navigability in both directions (requires roles for proper identification)

Summary: Basic notation for associations



Naming conventions

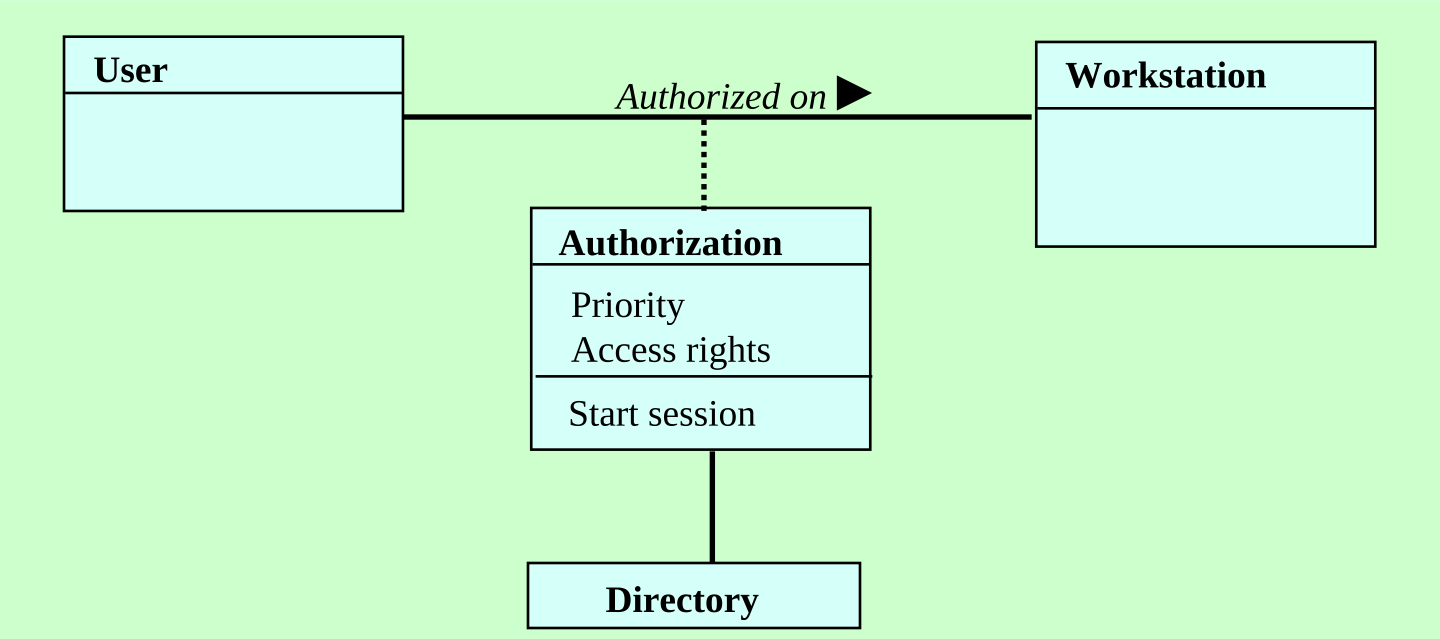


- Naming conventions allow often to infer the names of messages from the diagram

```
class Order {
 public Enumeration orderLines();
 public Customer customer();
}
```

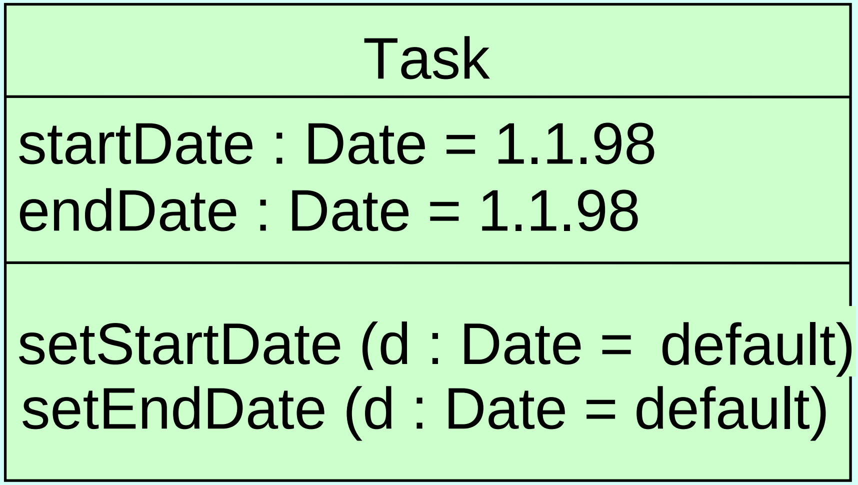
Association classes

- Useful if
  - attributes don’t belong to any one class but to the association



Classes and Objects

- As mentioned, a class defines the structure of a “group” of objects
- It defines:
  - – name
  - – attributes
  - – operations

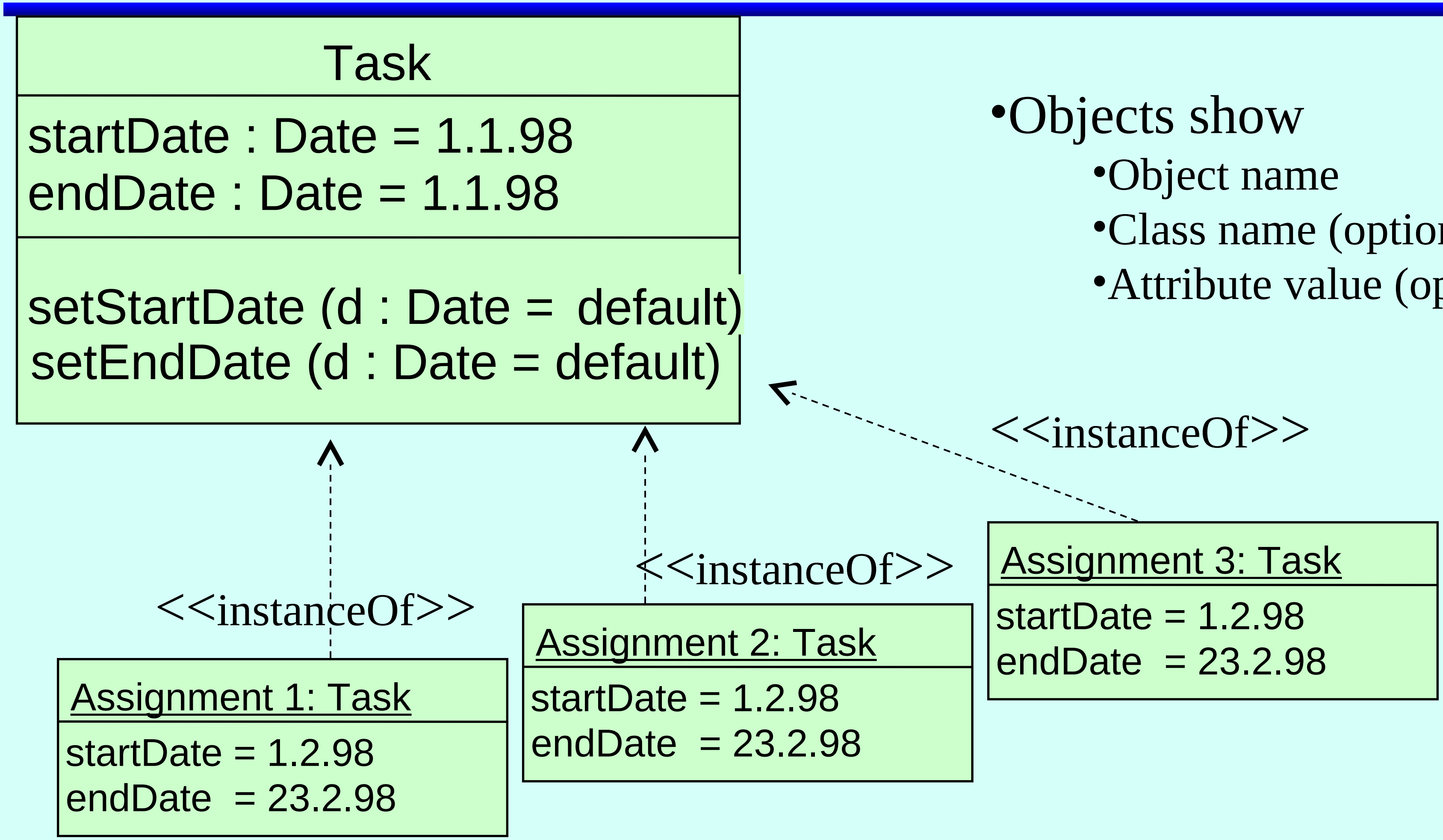


Task

startDate : Date = 1.1.98
endDate : Date = 1.1.98

setStartDate (d : Date = default)
setEndDate (d : Date = default)

Again on Classes and Objects



Task

startDate : Date = 1.1.98
endDate : Date = 1.1.98

setStartDate (d : Date = default)
setEndDate (d : Date = default)

<<instanceOf>>

<<instanceOf>>

<<instanceOf>>

Assignment 1: Task

startDate = 1.2.98
endDate = 23.2.98

Assignment 2: Task

startDate = 1.2.98
endDate = 23.2.98

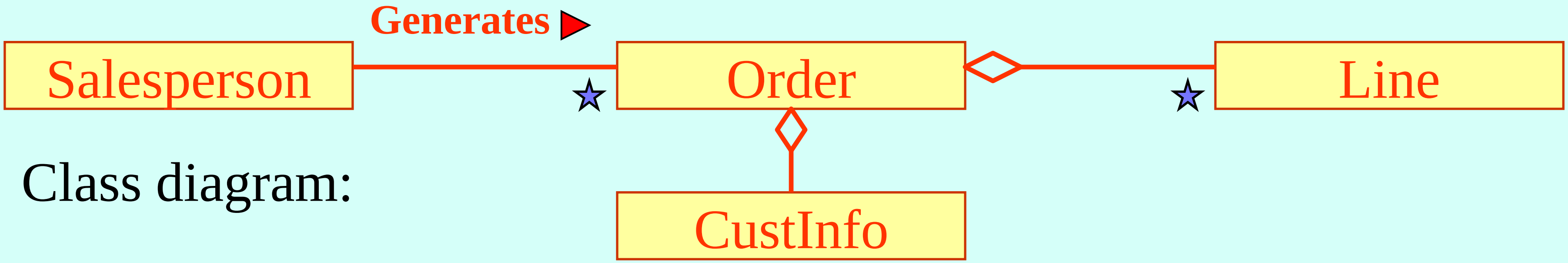
Assignment 3: Task

startDate = 1.2.98
endDate = 23.2.98

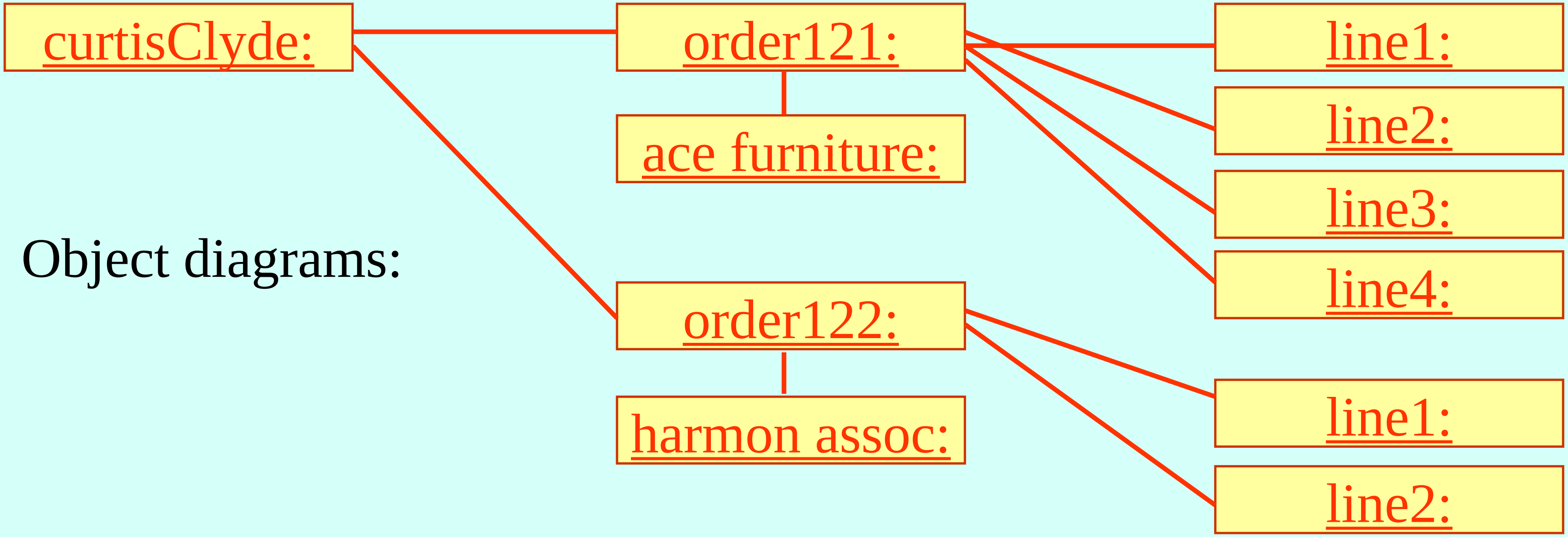
- Objects show
    - Object name
    - Class name (optional)
    - Attribute value (optional)

Example of Classes and Objects

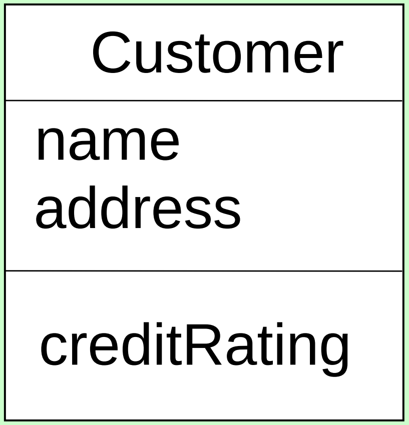
Class diagram:



Object diagrams:



Attributes



Customer

name
address

creditRating

- Conceptual: Indicates that customer have names
- Specification: Customer can tell you the name and set it
- Implementation: An instance variable is available

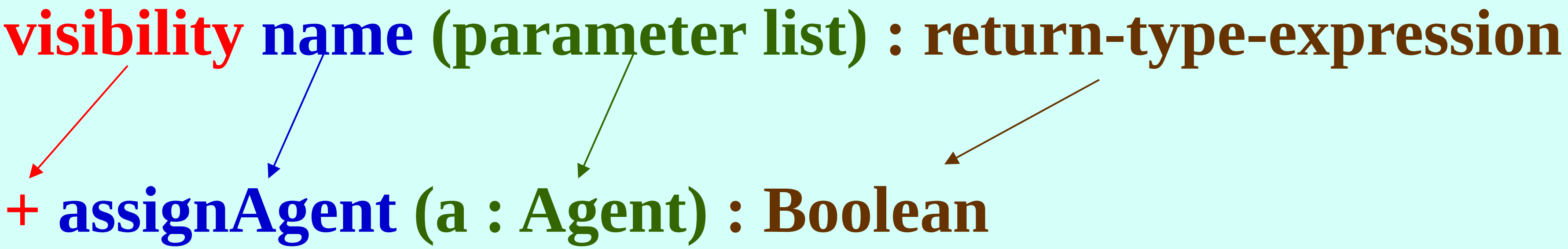
Difference between attributes and associations

- Conceptual perspective
  - not much of a difference!
  - Attributes are single valued (0..1)
- Specification/implementation perspective
  - Navigability from type to attribute
  - Attribute stores values NOT references
    - no sharing of attribute values between instances!
- Often: Stores simple objects
  - Numbers, Strings, Dates, Money objects

Operations

- Processes that a class knows to carry out
- Correspond to messages of the class
- Conceptual level
  - – principal responsibilities
- Specification level
  - – public messages = interface of the class
- Normally: Don’t show operations that manipulate attributes

UML syntax for operations



visibility name (parameter list) : return-type-expression

+ assignAgent (a : Agent) : Boolean

- **visibility**: **public (+)**, **protected (#)**, **private (-)**
  - Interpretation is language dependent
  - Nor needed on conceptual level
- **name: string**
- **parameter list: arguments (syntax as in attributes)**
- **return-type-expression: language-dependent specification**

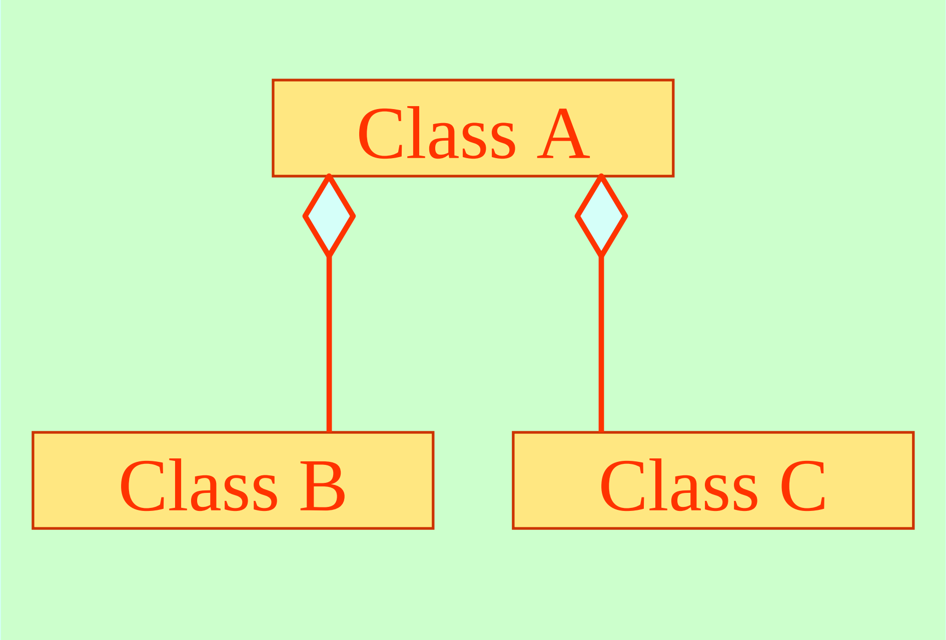
Types of operations

- *Query* = returns some value without modifying the class' internal state
- *Modifier* = changes the internal state
- Queries can be executed in any order
- Getting & setting messages
  - getting: query
  - setting: modifier

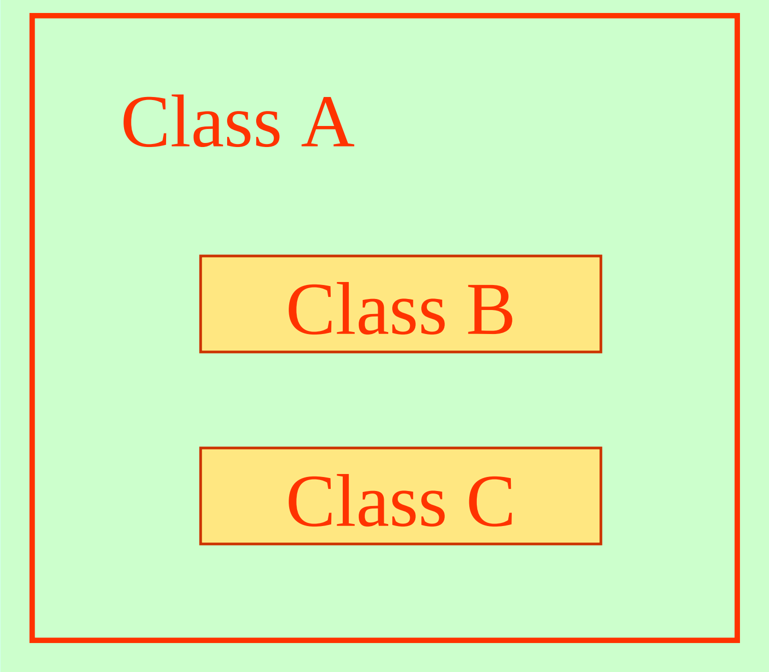
Aggregation

- Special form of association
- Components are *parts of* aggregated object
  - Car has an engine and wheels as its part
- Aggregation is transitive
- Typical example:
  - parts explosion
  - organizational structure of a company

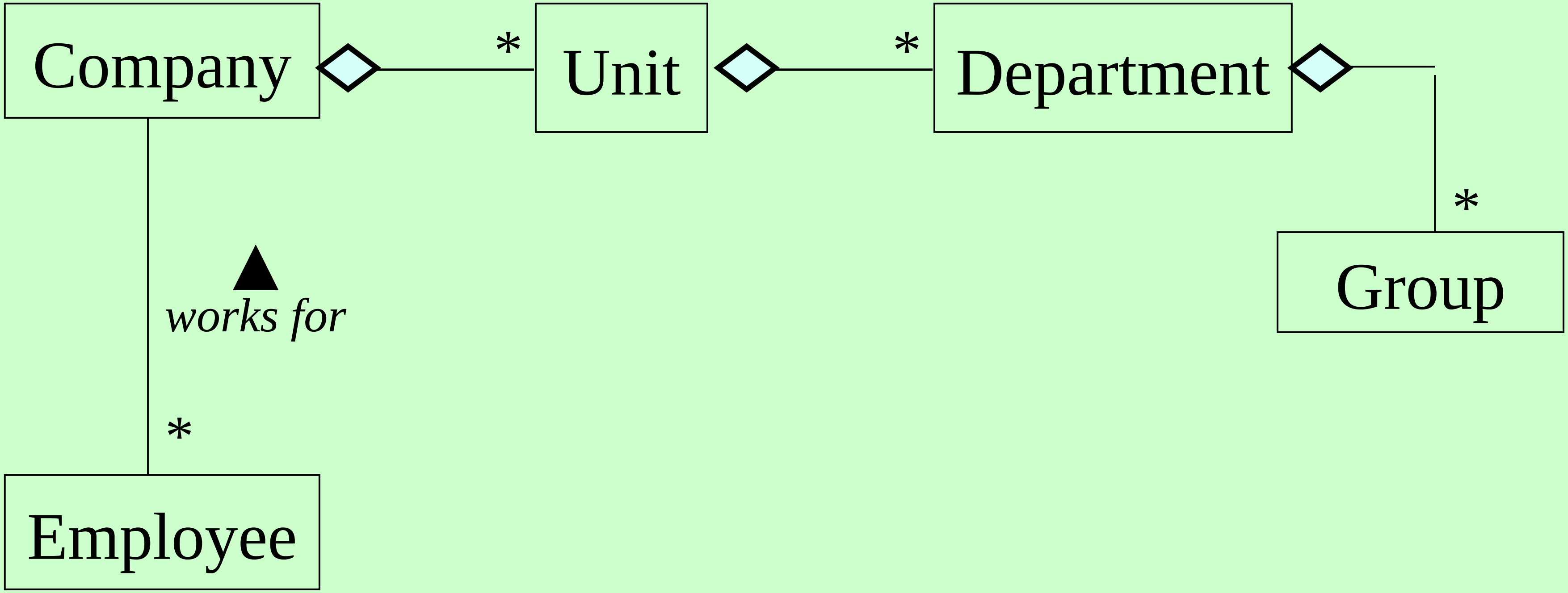
Notation for aggregation



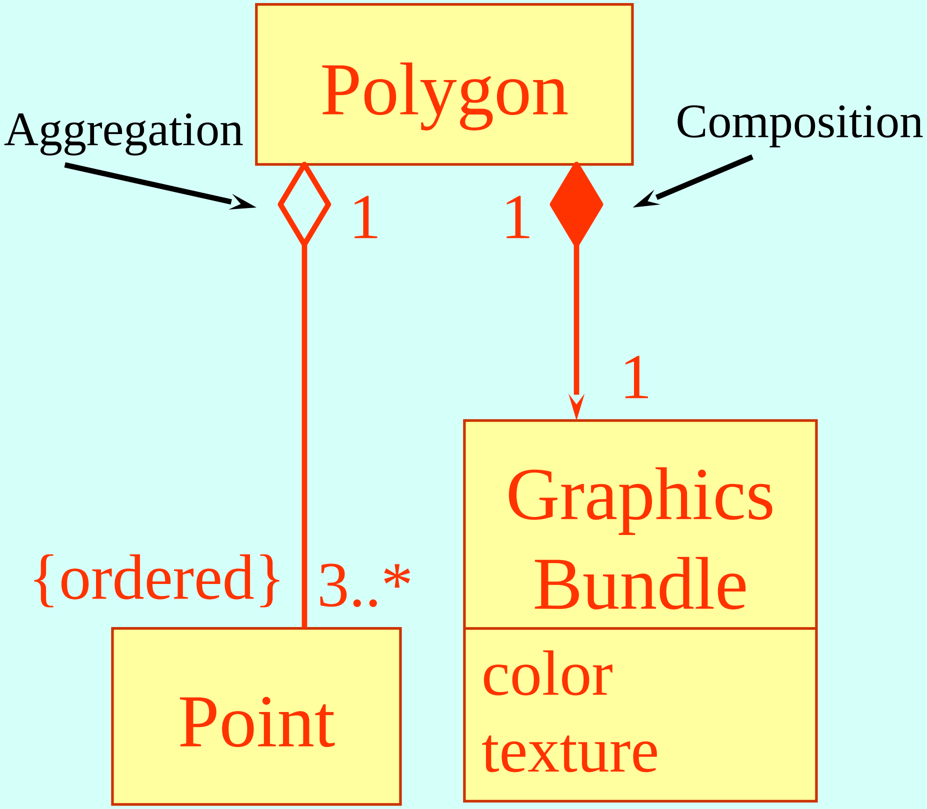
or



Example: Aggregation



Aggregation and composition



- Composition
  - Components belong only to one whole
  - Parts live and die with the whole
    - cascading delete
    - also needed for 1..1 associations
  - The players can be aggregated for the Flames
    BUT
    they are not killed when the Flames disappear

How to recognize aggregation?

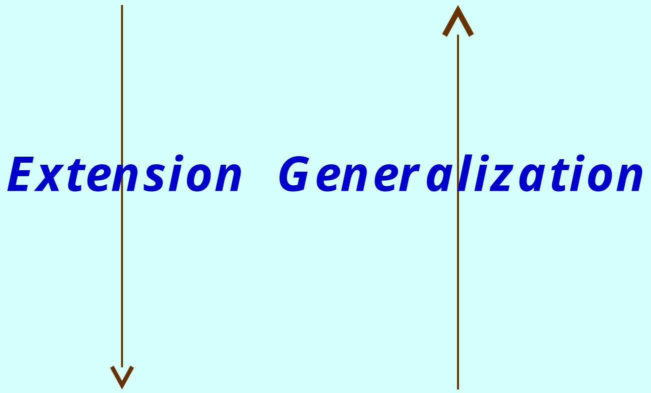
- Question: association or aggregation
  - Description “part of” correct?
  - Operation on whole affects parts too?

Proposed Exercise

- Develop the class diagram for the case of the NYSE

Generalization vs. Extension

- Car
  - Truck
  - Bus
  - Station wagon



Extension Generalization

- Many things in common
- Some differences

Instantiation and generalization

1. Shep is a Border Collie.
2. A Border Collie is a Dog.
3. Dogs are Animals
4. A Border Collie is a Breed.
5. Dog is a Species

1+2: Shep is a Dog
1+2+3: Shep is a animal
1+4: Shep is a breed?????
2+5: A Border Collie is a Species?????

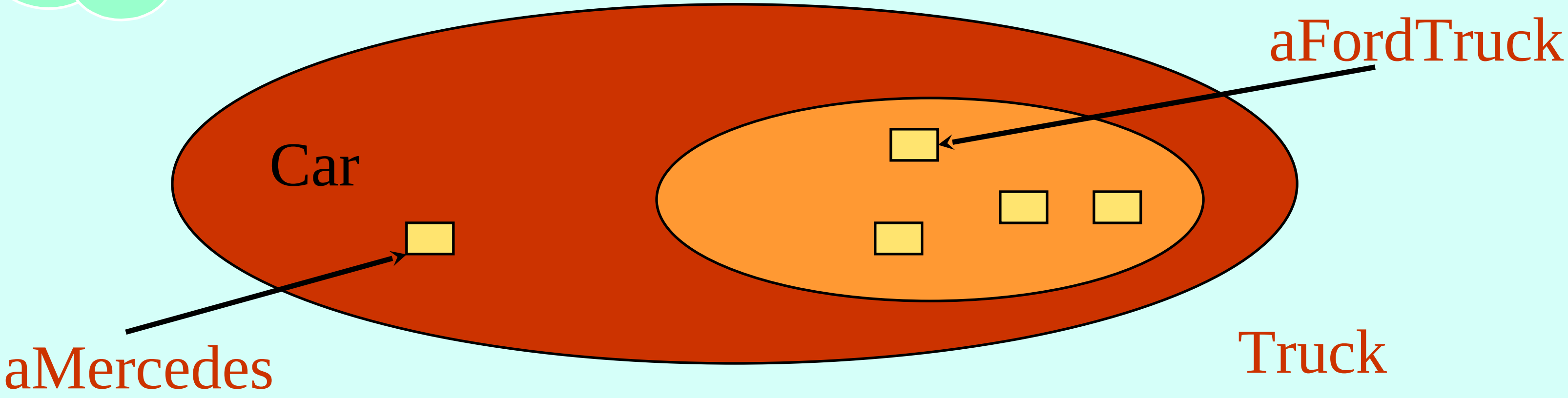
Generalization is transitive (is kind of)
Instantiation is not (is instance of)

Concept of generalization

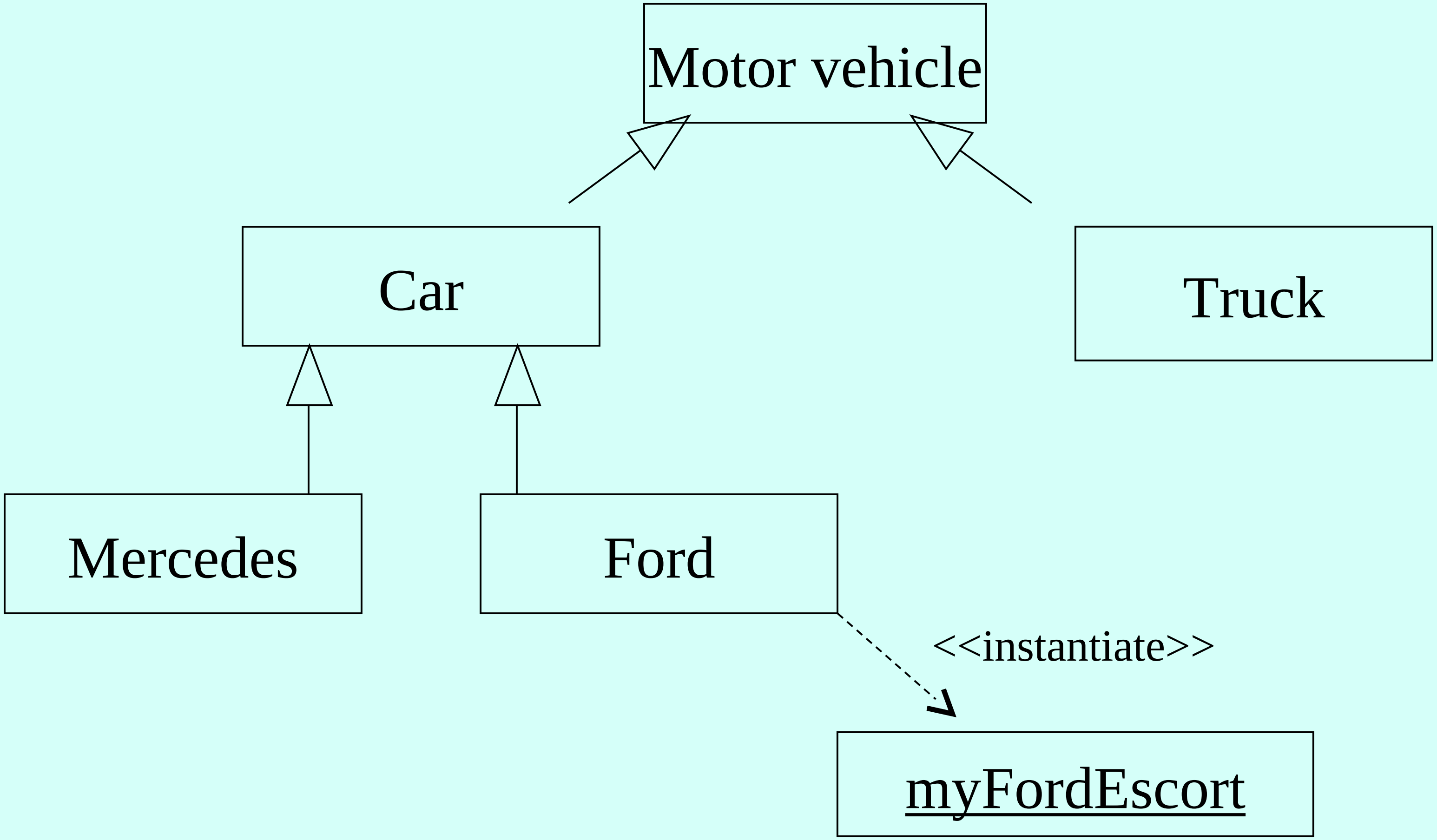
instantiation

- Class: Implicitly defines a set of objects
  - aCar ∈ Car = Set of all cars
- Generalization: Subset relation
  - Truck ⊆ Car

generalization



Class Diagram with Inheritance



How to define classes (revised)?

- Look for nouns in the Use Cases
- Define a class for every noun (+ add others)
- Document the set of rules that determine the set of objects belonging to the class
- Add associations to model the relations
- Think about the subset relationship to build generalizations ...

To which class does an object belong?

- Definition of class membership
  - implicit by rules
    - rules define condition for being a class member
    - attribute values available ⇒ class can be determined
    - terminological logic of AI (subsumption, classifier)
  - explicit by enumeration
    - instantiation defines class membership
    - problem: forbid operations that violate restrictions

Changing classes

- UML gives to an object the possibility to change its class dynamically (using the type stereotype)
- This is done in C++ and Java having a common base class and then changing the pointed/referenced objects with suitable constructors

Generalization: extension & restriction

- Attributes & operations of an ancestor class are inherited to the subclass
- Extension: adding of new attributes or operations
- Restriction:Additional restrictions on ancestor attributes
  - circle = Ellipse with equally long axes
  - Caution: arbitrary change of size of an axis of the circle can violate restriction

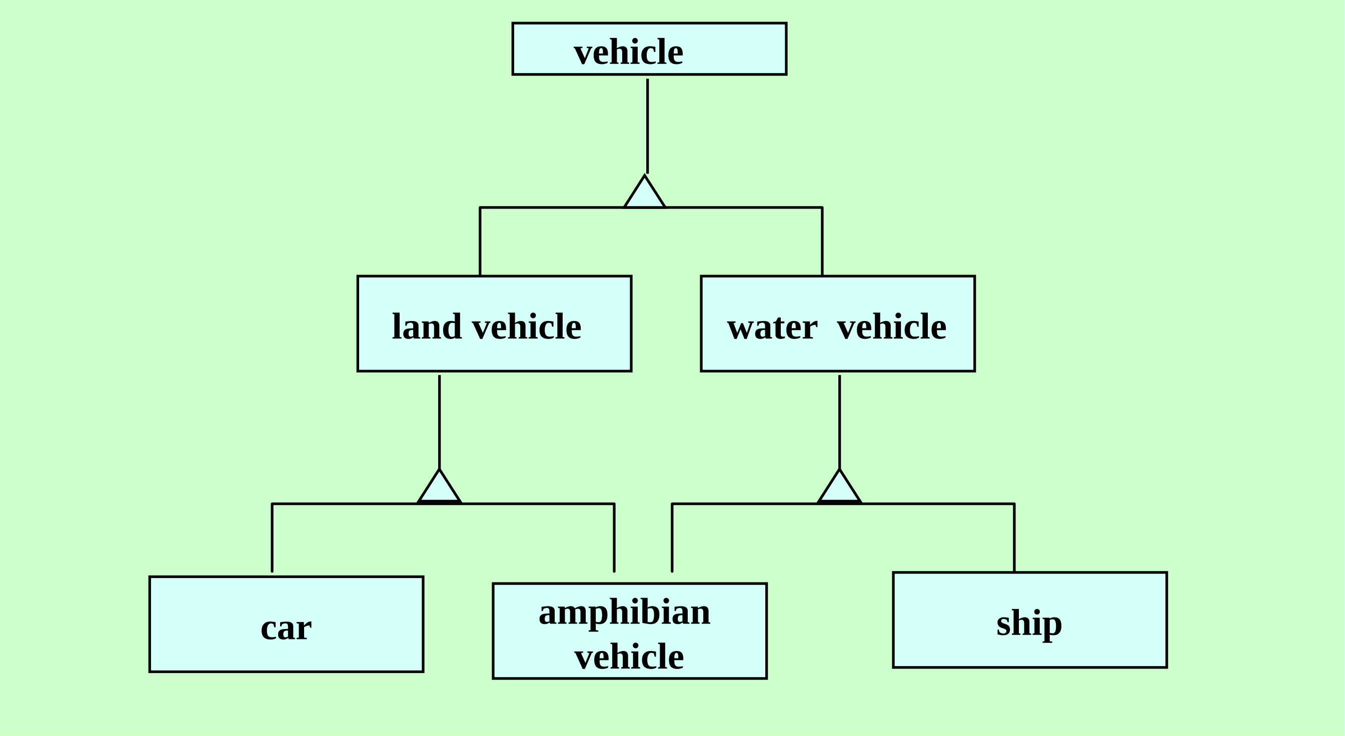
Perspectives

- Conceptual: Subset relationship
- Specification: Subtype conforms to supertype interface
- Implementation: Implementation inheritance, subclassing

☠BEWARE: Do not subclass when the conceptual level does not support it -- Use aggregation (A stack is ***not*** a list with some overriding!)

Multiple inheritance

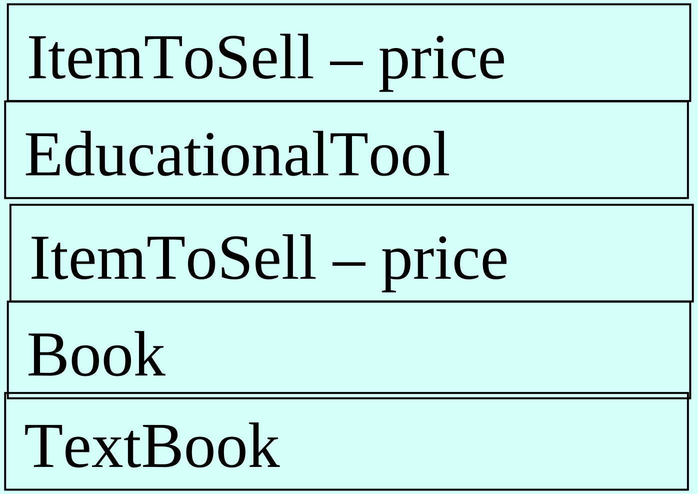
- Class inherits features from several superclasses



Virtual inheritance

- `Class ItemToSell { double price;};`
- `Class EducationalTool : public ItemToSell{ };`
- `Class Book : public ItemToSell{ };`
- `Class TextBook: public EducationalTool, public Book { };`

Inside



ItemToSell – price

EducationalTool

ItemToSell – price

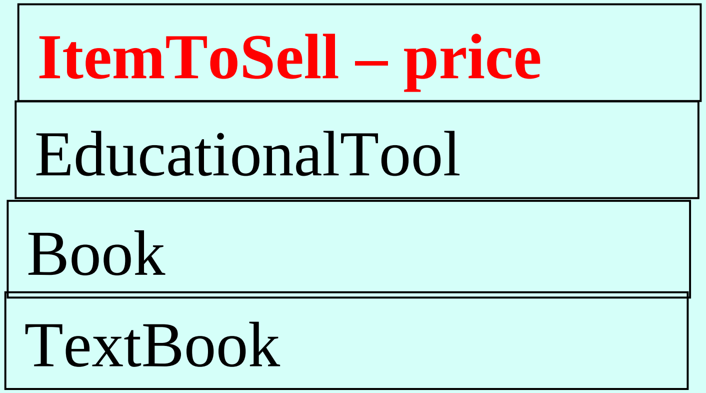
Book

TextBook

Virtual inheritance

- Class ItemToSell {double price;};
- Class EducationalTool : virtual public ItemToSell{ };
- Class Book : virtual public ItemToSell{ };
- Class TextBook: public EducationalTool, public Book { };

Inside



ItemToSell – price

EducationalTool

Book

TextBook

Proposed Exercise

- Define the class of “textbooks” as derived from the concept of “educational tool” and of “book”
- Redefine the class avoiding multiple inheritance

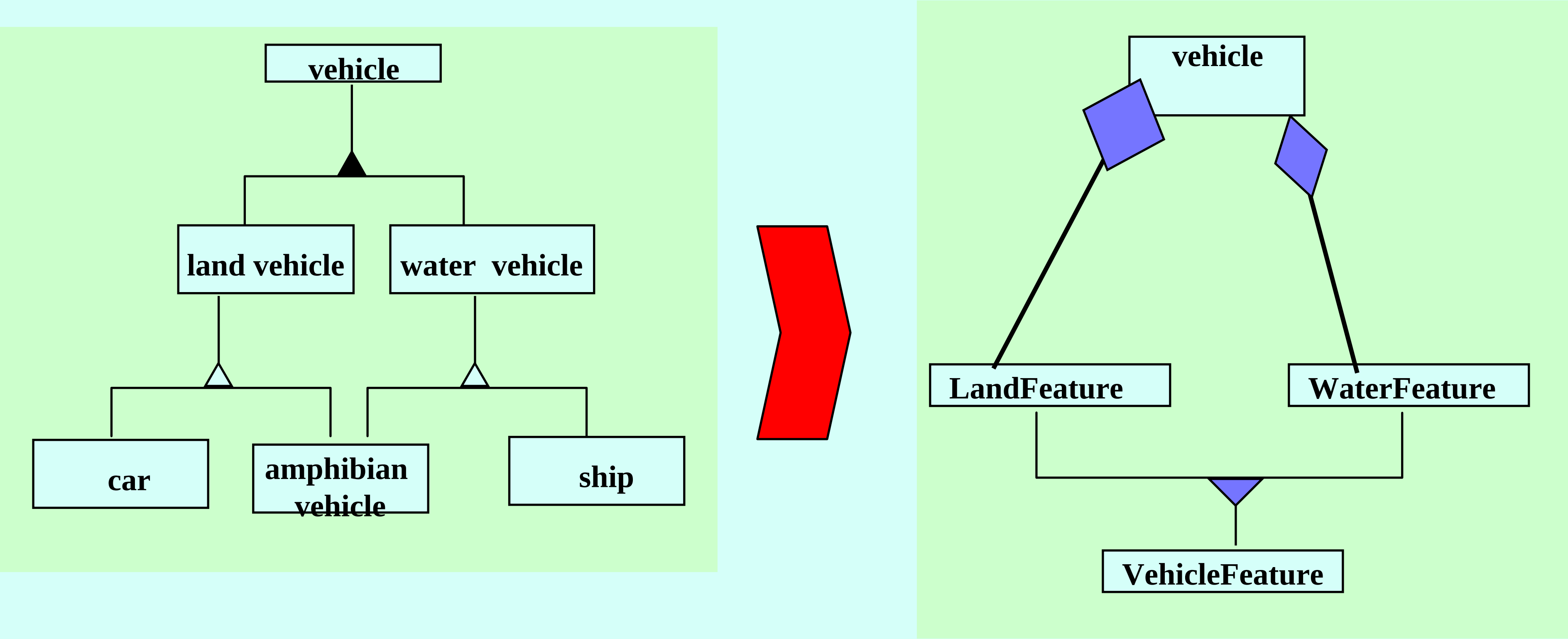
Discussing multiple inheritance

- Advantages:
  - closer to human thinking
  - higher flexibility for specifying classes
  - higher chances for reuse
- Disadvantages:
  - loss of clarity → which method is executed
  - implementation more complicated
  - conflict resolution is necessary for multiple inherited features

Avoiding multiple inheritance

- Basically: question of implementation
- Often the simplest way: restructure model
- Techniques for restructuring:
  - Delegation & aggregation
  - Inheritance based on the most important feature and delegation of the rest
  - Generalization based on different dimensions

Delegation & aggregation



vehicle

land vehicle

water vehicle

car

amphibian vehicle

ship

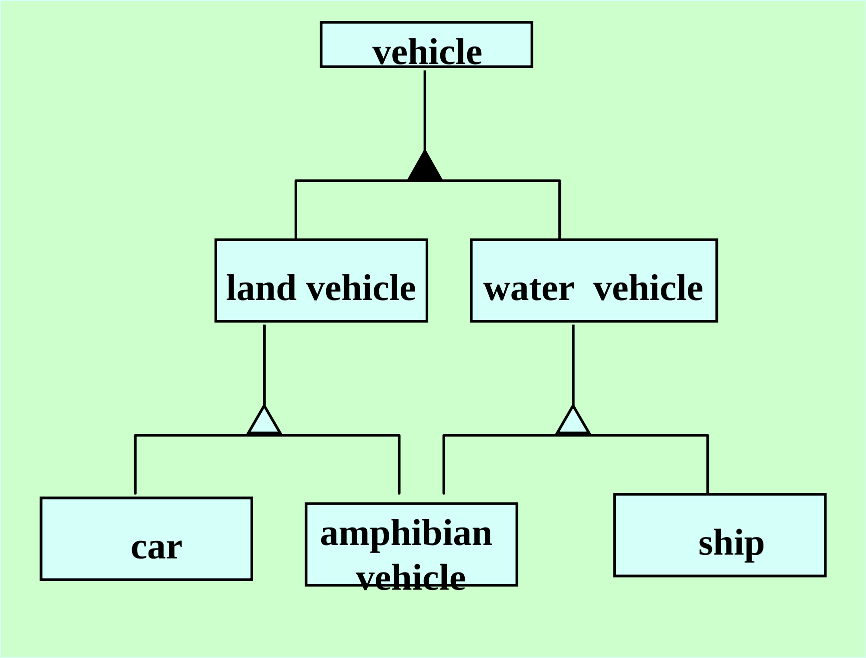
vehicle

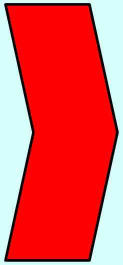
LandFeature

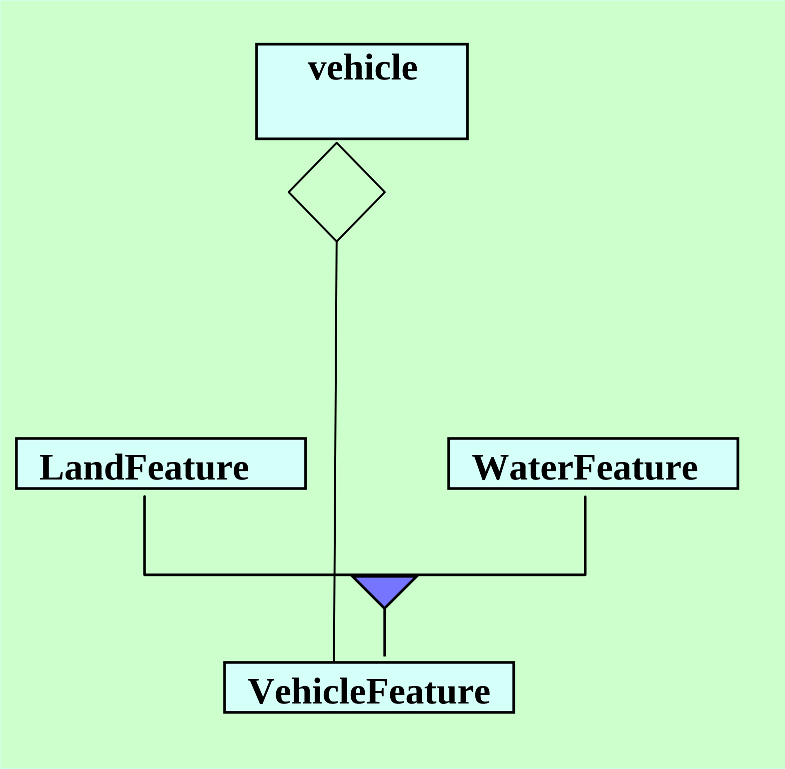
WaterFeature

VehicleFeature

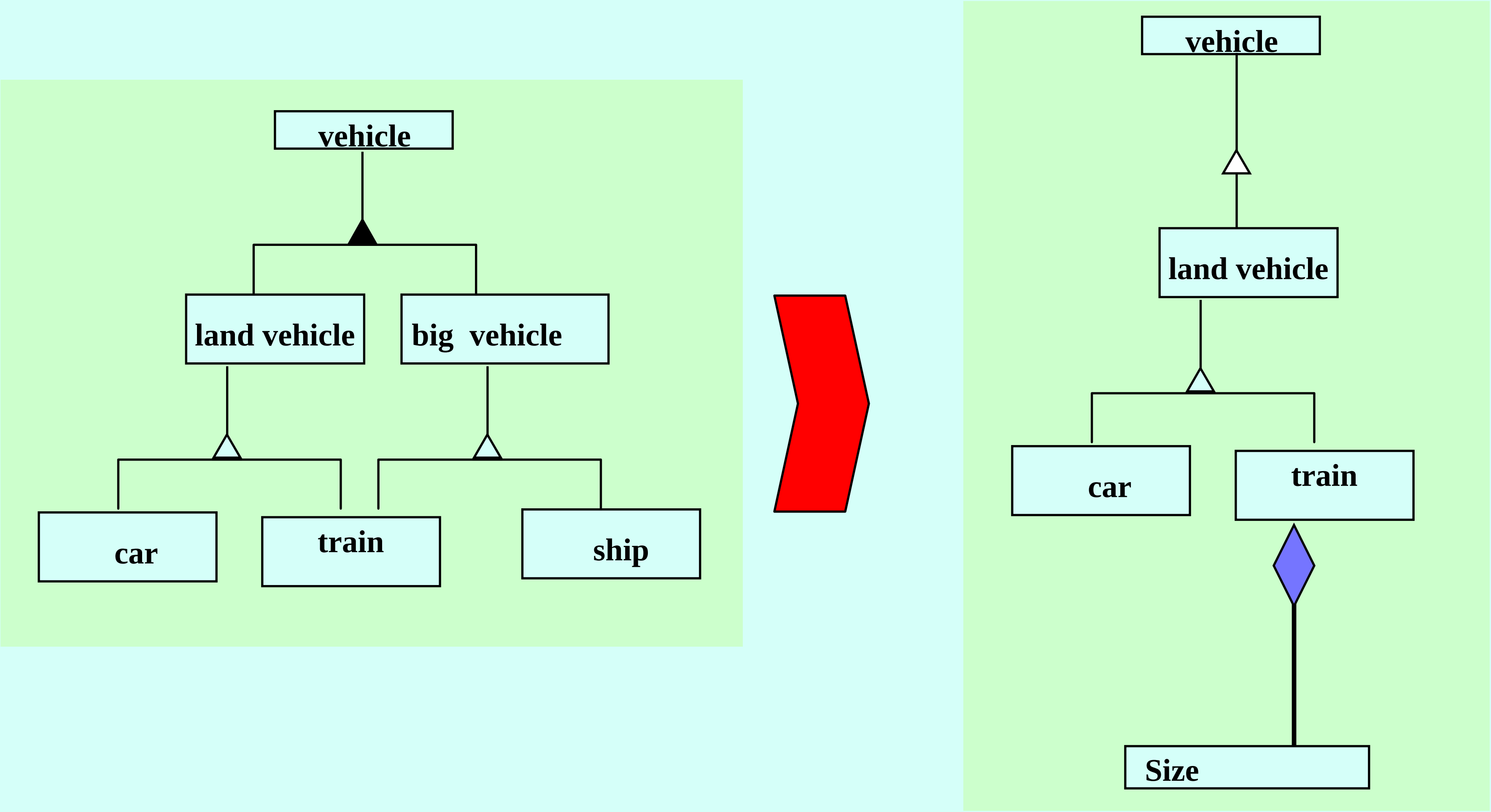
Delegation & aggregation



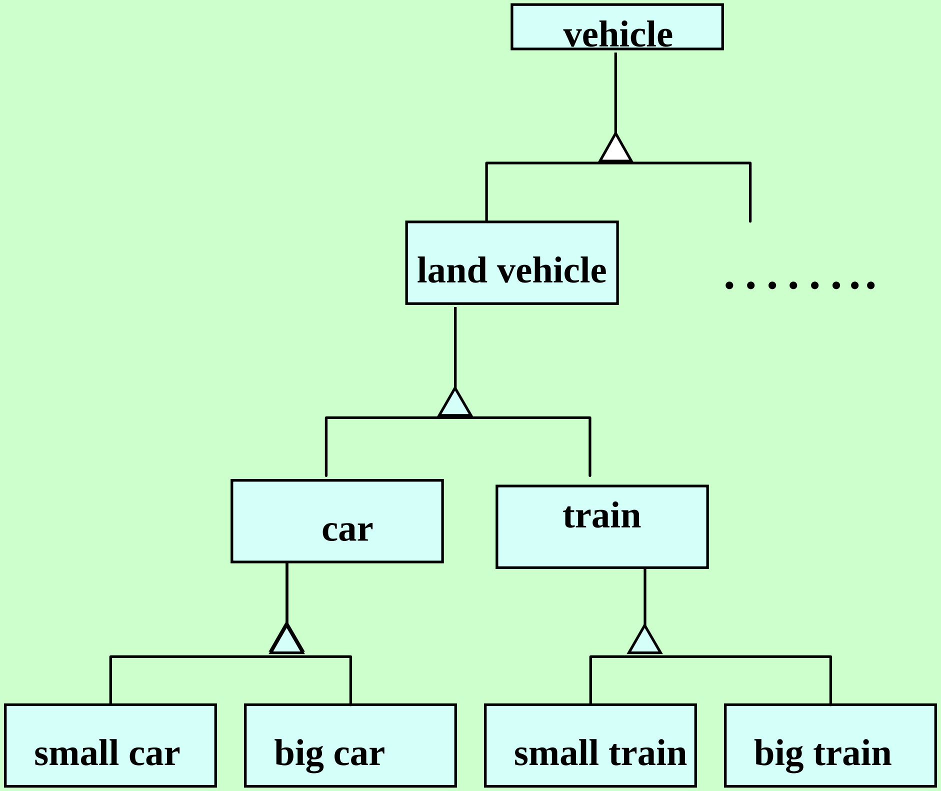




Most important feature & aggregation



Generalization based on different dimensions



When to use class diagrams

- Class diagrams are the backbone of OO development approaches
- Don’t use all the notations
  - start with simple stuff
- Take the perspective into account
  - not to many details in analysis
  - specification often better than implementation
- Concentrate on key areas
  - better few up-to-date diagrams than many obsolete models

Creating a class diagram

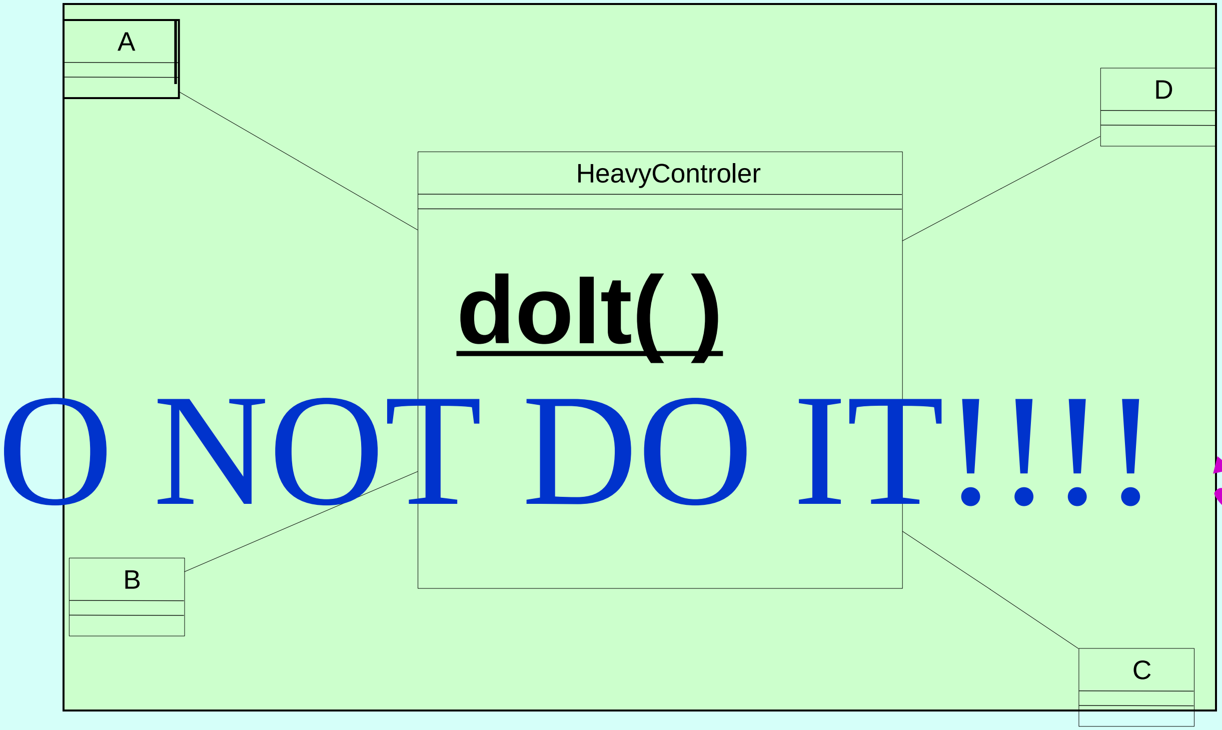
- Start simple
  - major classes & obvious associations
- Then add
  - Attributes
  - Multiplicity
  - Operations

Rules of thumb

- One class can be part of several diagrams
- Diagrams shall illustrate specific aspects
  - Not too many classes
  - Not too many associations
  - Hide irrelevant attributes/operations
- Several iterations needed to create diagram

Avoid “Heavy” classes

- Controller does everything
- Other classes: Data encapsulation only



□DO NOT DO IT!!!! ☠

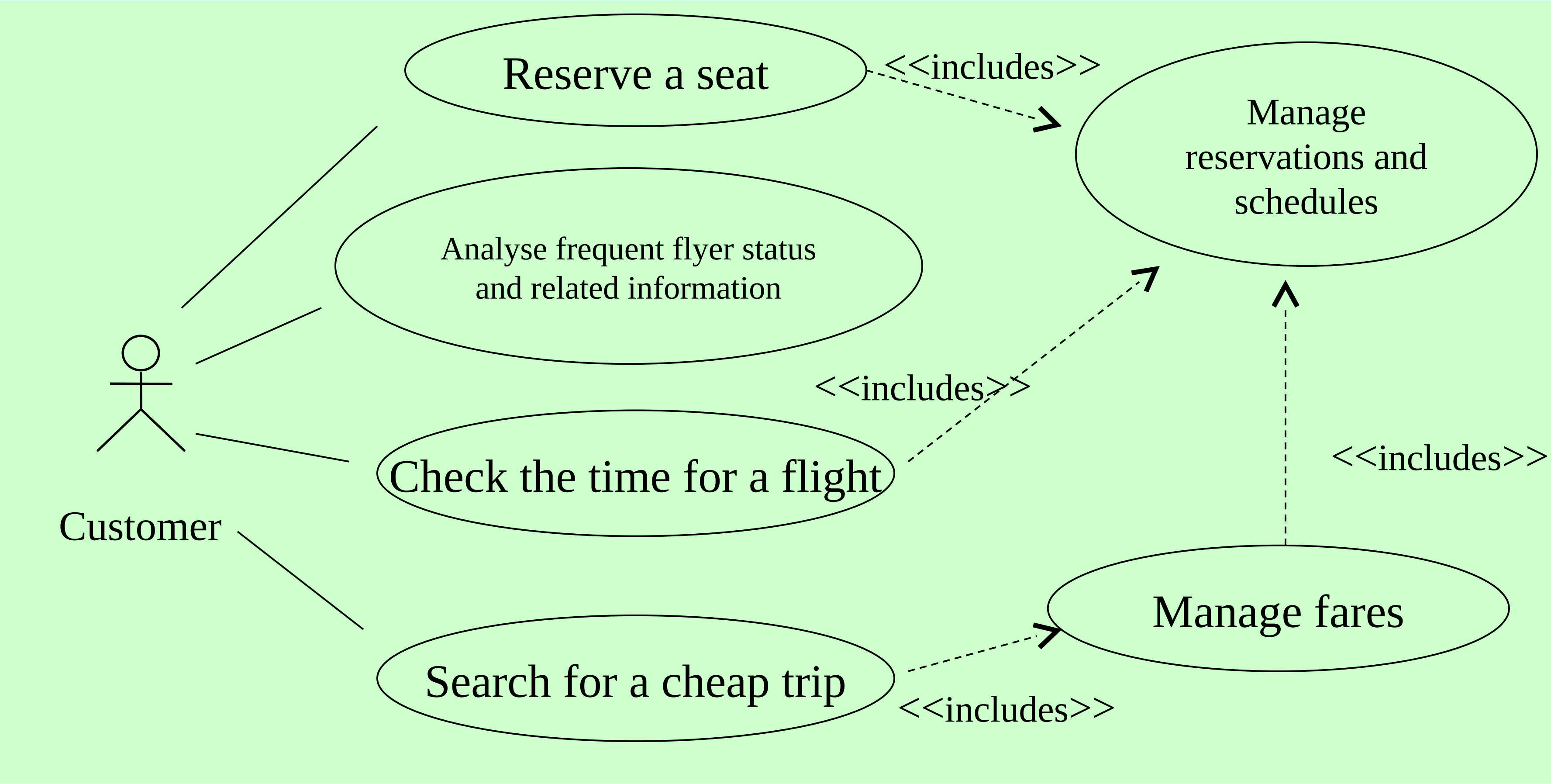
Proposed Exercise

- Model the classes and the objects in the Lufthansa web site. The starting point is the analysis of the use cases.
- *[Shorter form]* Focus on the frequent flyer subsystem.

Extraction of Classes in the Lufthansa example

*The **user** enters the subsystem. To do so it needs to be a **frequent flyer**. Inside the subsystem, the user can access (a) general information about the **frequent flyer program** -the **reward schema**, how to enroll, how to earn **miles** with **partner companies**, and (b) specific information on her/his **miles earned**, the **status level**. The user can also update her/his **address**.*

Partial Solution


Reserve a seat
<<includes>>
Manage reservations and schedules
Analyse frequent flyer status and related information
<<includes>>
Customer
Check the time for a flight
<<includes>>
Manage fares
Search for a cheap trip
<<includes>>

Further Proposed Exercise (1)

Develop an OO Analysis Model for a system supporting the reservation and scheduling for taxi drivers that was discussed in the first day of the course.

Further Proposed Exercise (2) *Stocks Trading Service*

- By connecting to the service, a user can connect to different banks to acquire stock prices. The system also allows the user to perform some trend and prediction analysis of prices. If users are interested in ordering some stocks, they can choose to order them immediately or with a delay. They can also either bid at a single price or within a range of prices.
- The system should handle the situation where the connection to a bank is down, there is a conflict of bids, or if a particular stock is no longer available.

Further Proposed Exercise (3) *Network Printing Service*

- There is a super high-resolution colour laser printer available on the network for users to print documents to. The service allows users to preview the output of their document on their screens. In addition, the user can also view the status of the printer to see whether there are other documents waiting to be printed and whether there are any problems with the printer (such as paper jams, out of paper, low on toner, etc…). In addition, users can monitor their own print jobs and delay or delete jobs as they see fit.
- To use this service, a user needs to have the proper authorization and print quota to print. A system administrator manages users and their print quotas.

Further Proposed Exercise (4) *Component Brokerage System*

- This system essentially acts as a broker for software components. When developers have completed development of their software, they can deploy them as reusable software components for others to use. By connecting to the system over the Internet, these developers can submit their components to the system. An administrator then reviews the component for its functionality and ways of connecting to other components, categorizes it, and publishes it in a publicly-viewable area.

- Customers (such as other developers) can then connect to the public system and browse/search the components. When they have found something useful, they can download it for use on their own machine.

- Later, the providers of the components can connect to the system and view the download statistics of their components. They can also add/remove components from the system.

Further Proposed Exercise (5)
Bug Tracking System

- Developers use this system to track bugs in an on-going software project. Developers who find bugs can submit a report. Other developers can then assign the bug to a particular developer (especially the developer responsible for the software module) to fix it. In addition, users can browse/search all the bugs in the system so far.
- An administrator manages users to restrict access to the bug tracking system. In addition, the administrator should also be able to generate reports on the state of the project in the form of a set of web pages updated daily at 2am.