Software Modelling
and analysis
(with UML and OCL)

http://uml.org

Juan de Lara, Elena Gómez, Esther Guerra
{Juan.deLara, MariaElena.Gomez, Esther.Guerra}@uam.es
Computer Science Department
Universidad Autónoma de Madrid

Masters: I2TIC and Formal methods

Index

- **Introduction.**

- Class and Object Diagrams.
- Other Diagrams.
- OCL: The Object Constraint Language.
- Bibliography.

Introduction

- *Unified Modeling Language.* OMG Standard http://www.uml.org

- Standard notation for describing object oriented systems, derived from previous modelling notations
  - **Booch** - OOD.
  - **Rumbaugh** - OMT.
  - **Jacobson** - OOSE and Objectory.

- Combines the better properties of:
  - Data modelling (ERD)
  - Business models (*workflow*)
  - Object models
  - Component models

Introduction

- Graphical language to model, visualize and document parts of a software system from different viewpoints

- Can be used
  - With any development process
  - Throughout the software lifecycle
  - With different implementation technologies

- It is being used in many other areas, like
  - process modelling, web engineering, systems engineering, communications engineering (e.g. antennas), services, telecommunications, voice-based applications,…
  - Adaptation/extension by means of ***profiles***

Introduction

- It is **not** a method, **not** a process, **not** a methodology.
- **Does not** establish which models to build during the development process.
- For an optimal use, it should be used in a process:
  - Guided by use cases
  - Architecture centered
  - Iterative and Incremental (e.g., Rational Unified Process)

The UML Language

- UML is a family of notations, useful to describe different aspects of a system
  - ***Static:*** describes the elements of a system and its relations
  - ***Dynamic:*** describes the behaviour of the system as time progresses
    - ***Use cases.*** From the point of view of the user (actors)

Models

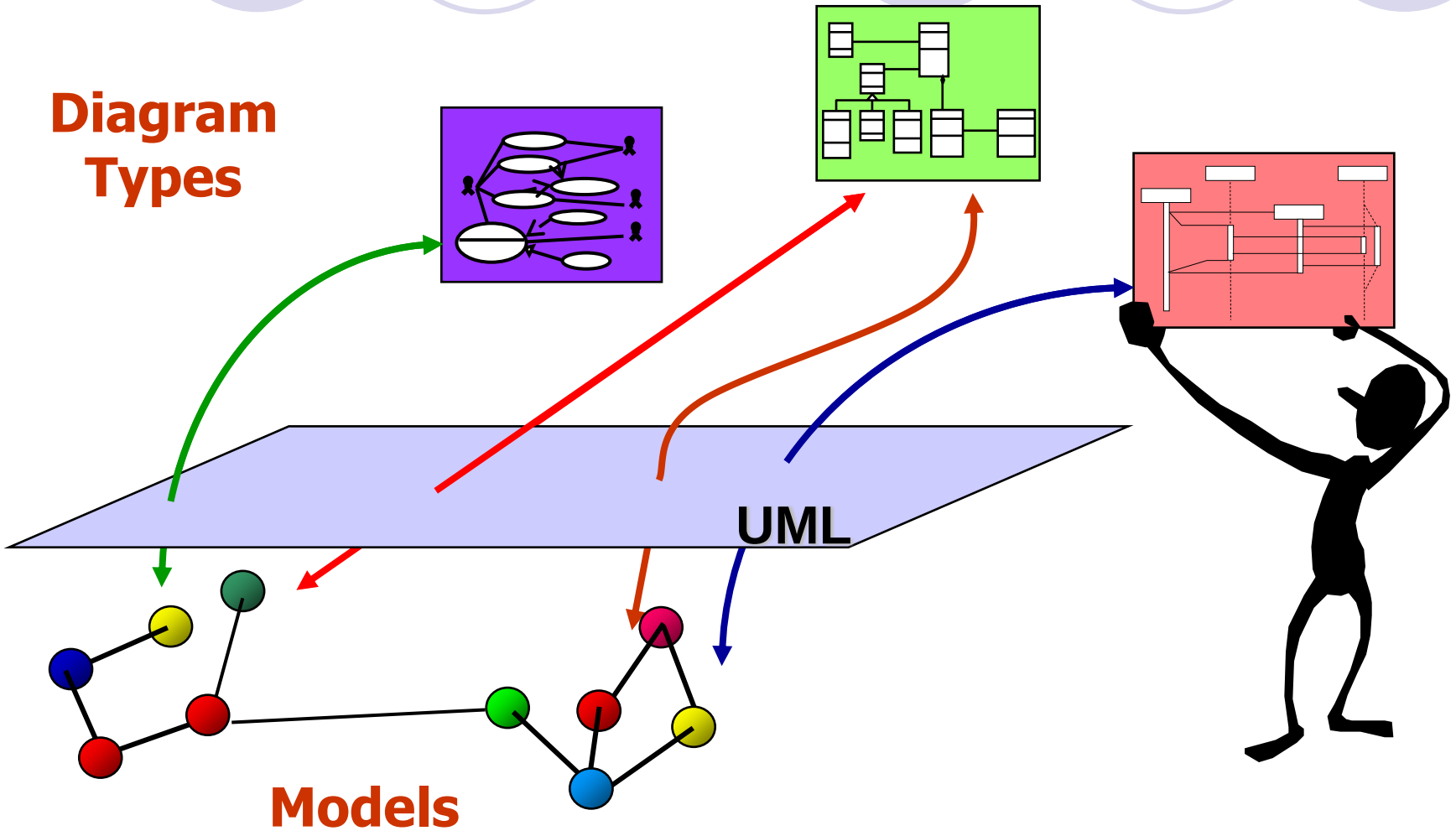
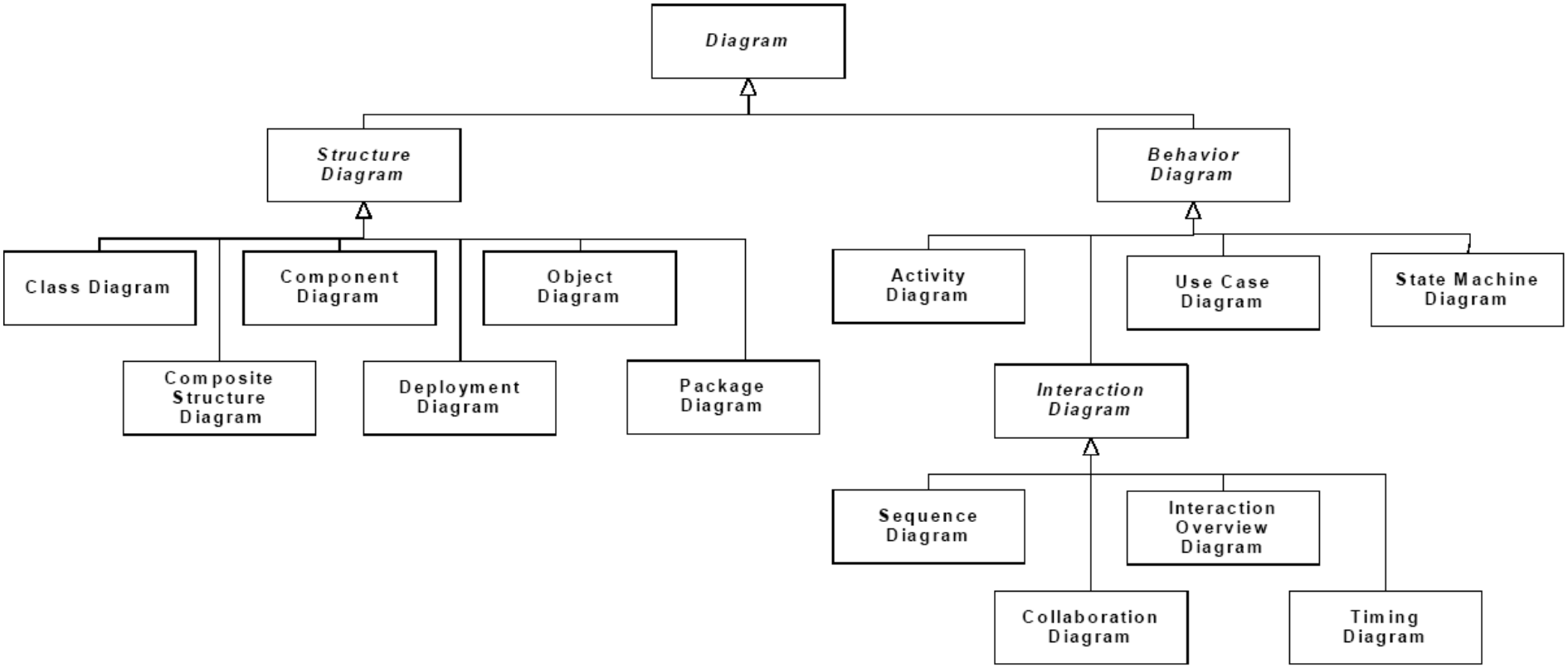


Diagram Types



Some challenges

- Model consistency
- Properties of the whole system
- Precise semantics of diagrams and their combinations
- UML vs. Domain Specific Languages

Index

- Introduction.
- **Class and Object Diagrams.**

- Other Diagrams.
- OCL: The Object Constraint Language.
- Bibliography.

Classes and Objects

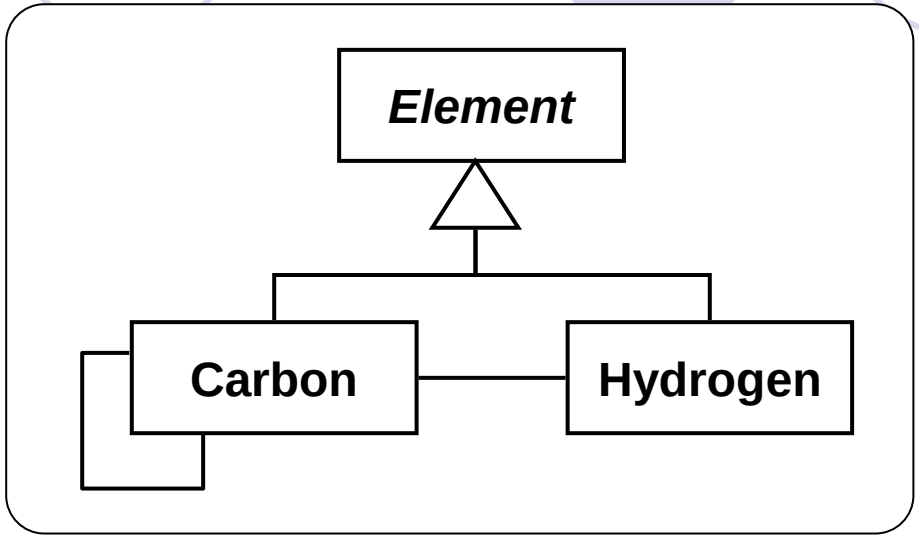
- Class and Object diagrams are the main diagrams to represent structural aspects

- ***Class Diagrams***. Structure of the System
  - Classes
    - Attributes: visibility, types, initial values, modifiers
    - Operations: visibility
  - Relations with other classes: Associations

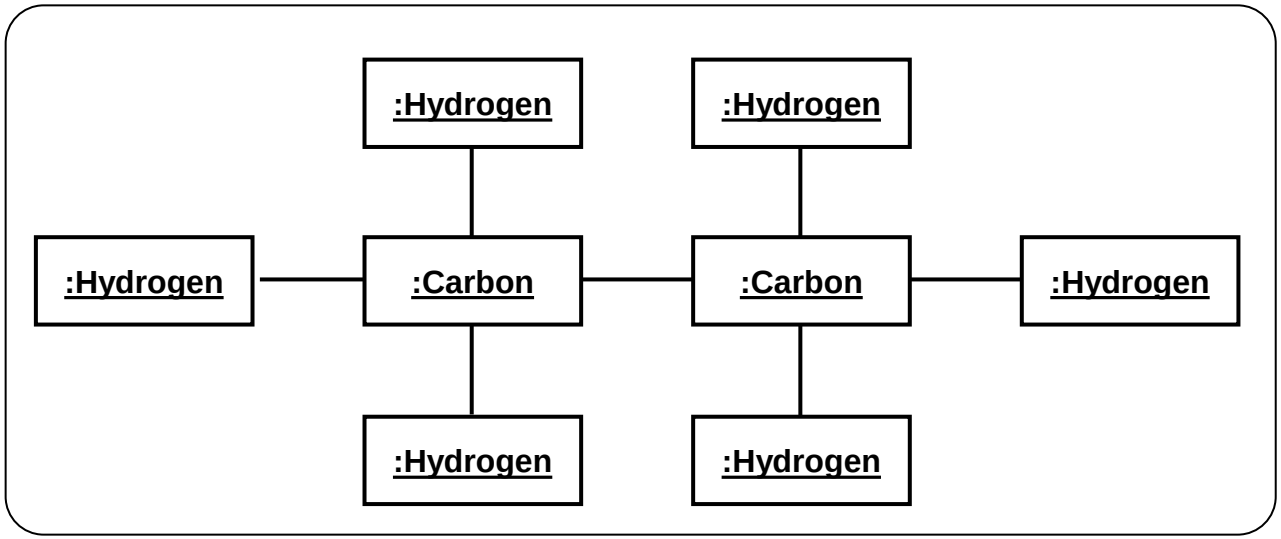
- ***Object Diagrams***. Structure of the system at runtime. Snapshots
  - *Objects*. Class instance
    - Slots (actual values of attributes)
  - *Links.* Object relations, association instances

Classes and Objects

Class Diagram

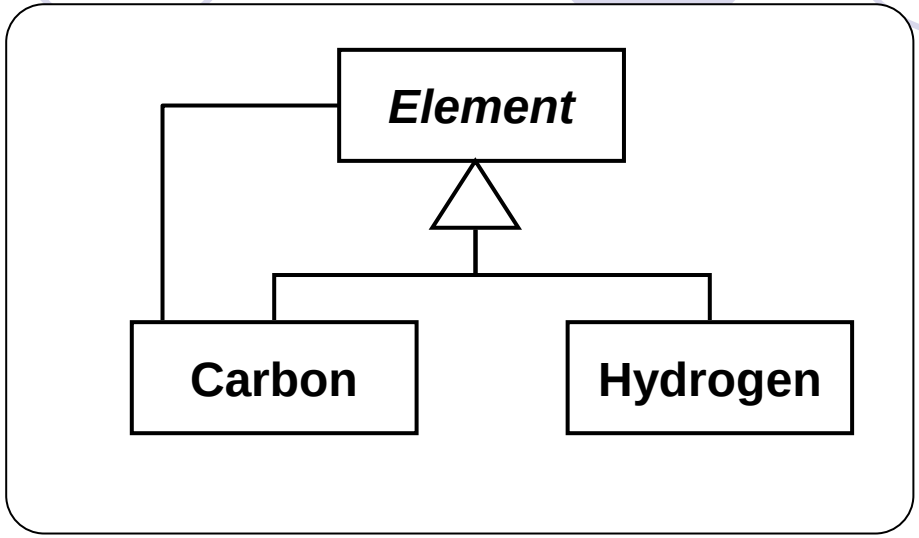


Object Diagram

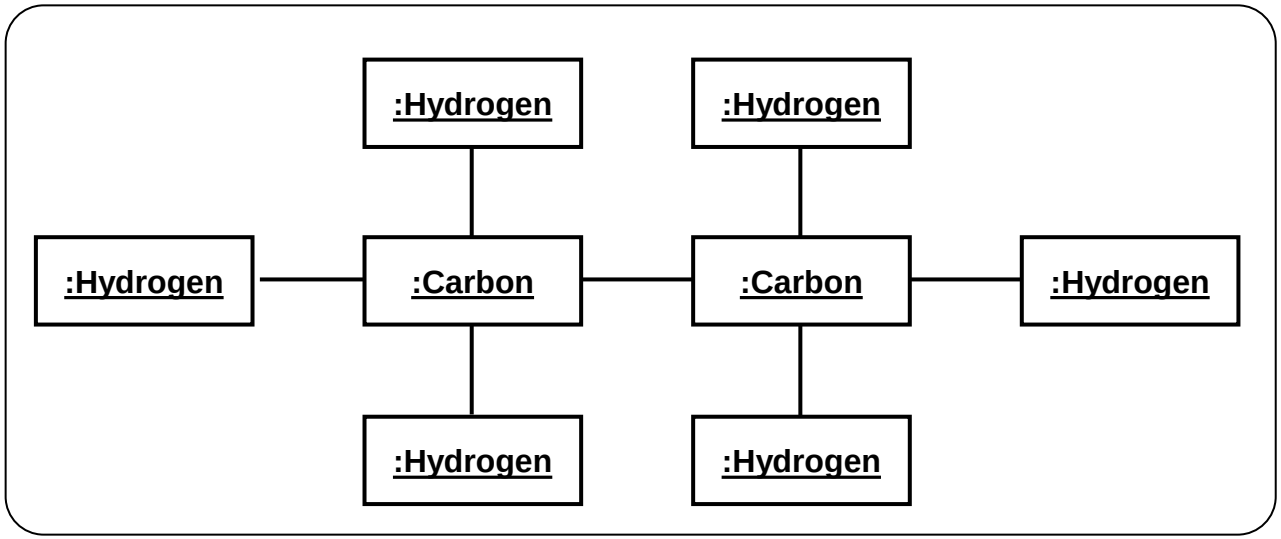


Classes and Objects

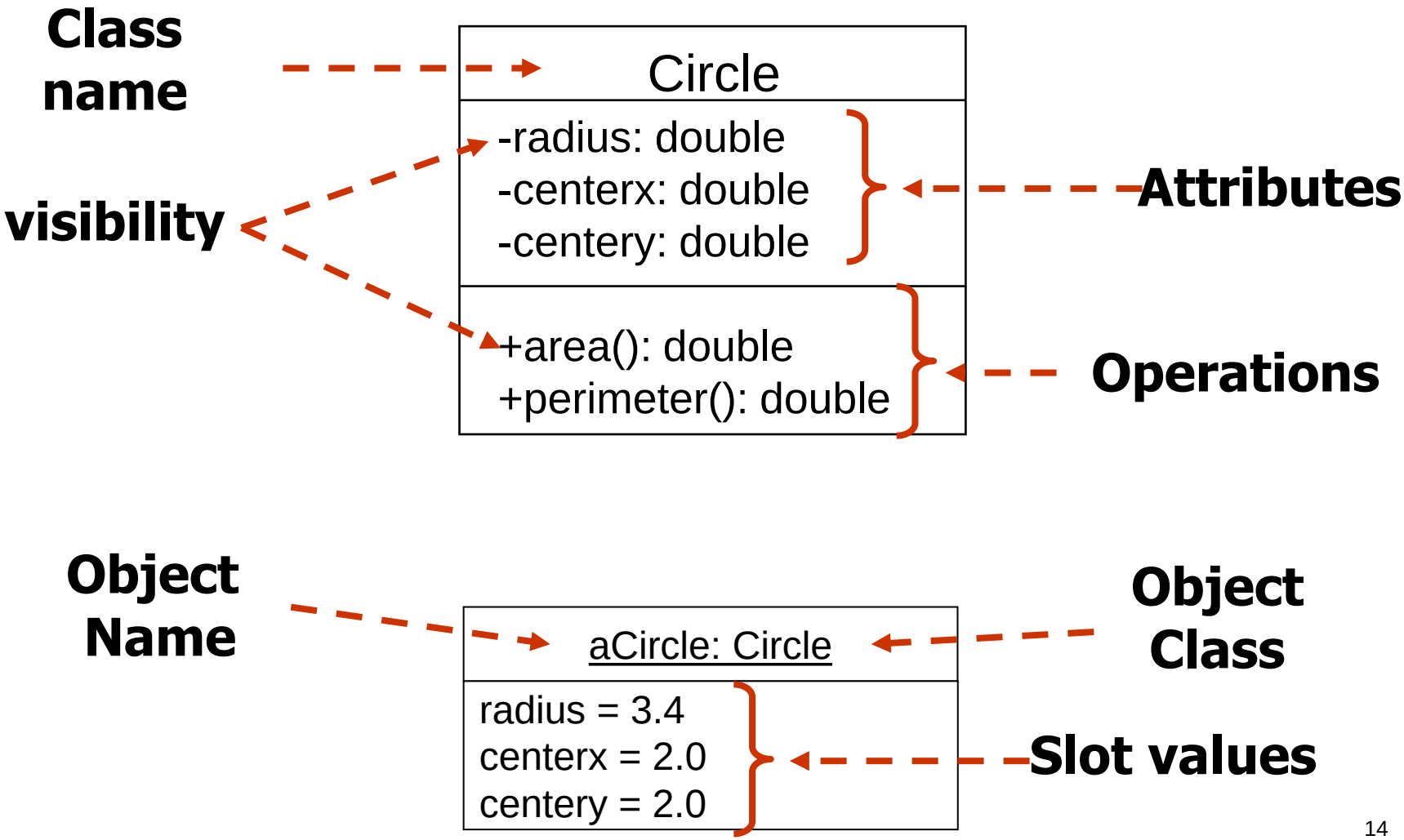
Class Diagram



Object Diagram



Classes and Objects



Classes

Attributes

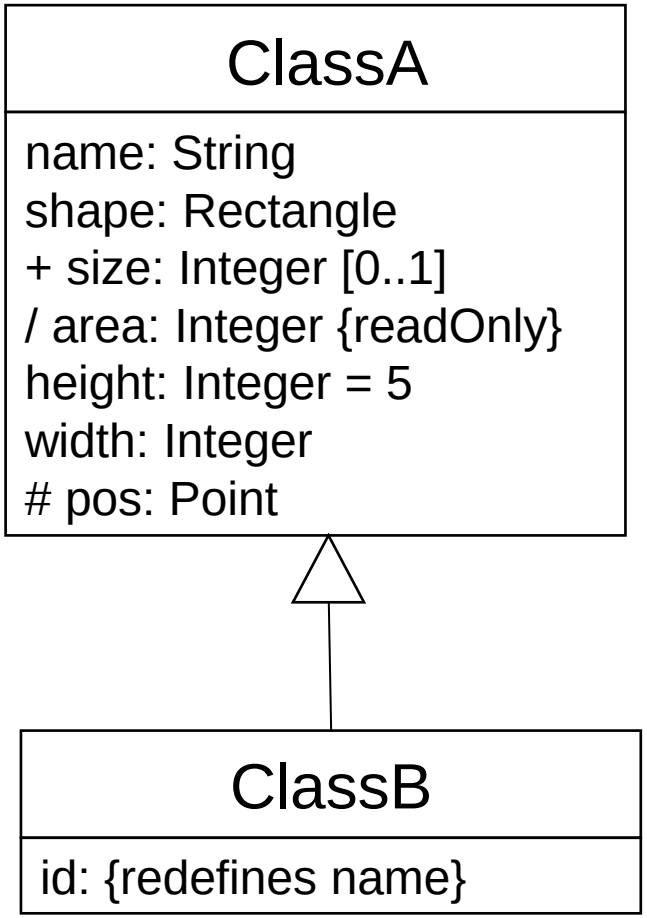
- Notation for attributes:

[*visibility***] [/]** *name* **[***: type***] [***multiplicity***] [**= *value***] [{** *(property)+* **}]**

- Visibility:
  - public(+), private(-), protected(#), package (~).
- "/" = Derived attribute.
- Multiplicity between "[ ]" and 1 by default.
- Valid properties: {readOnly}, {union}, {subsets <property-name>}, {redefines <property-name>}, {ordered}, {bag}, {seq}, {sequence}, and {composite}.
- Static attributes are <u>underlined</u>.

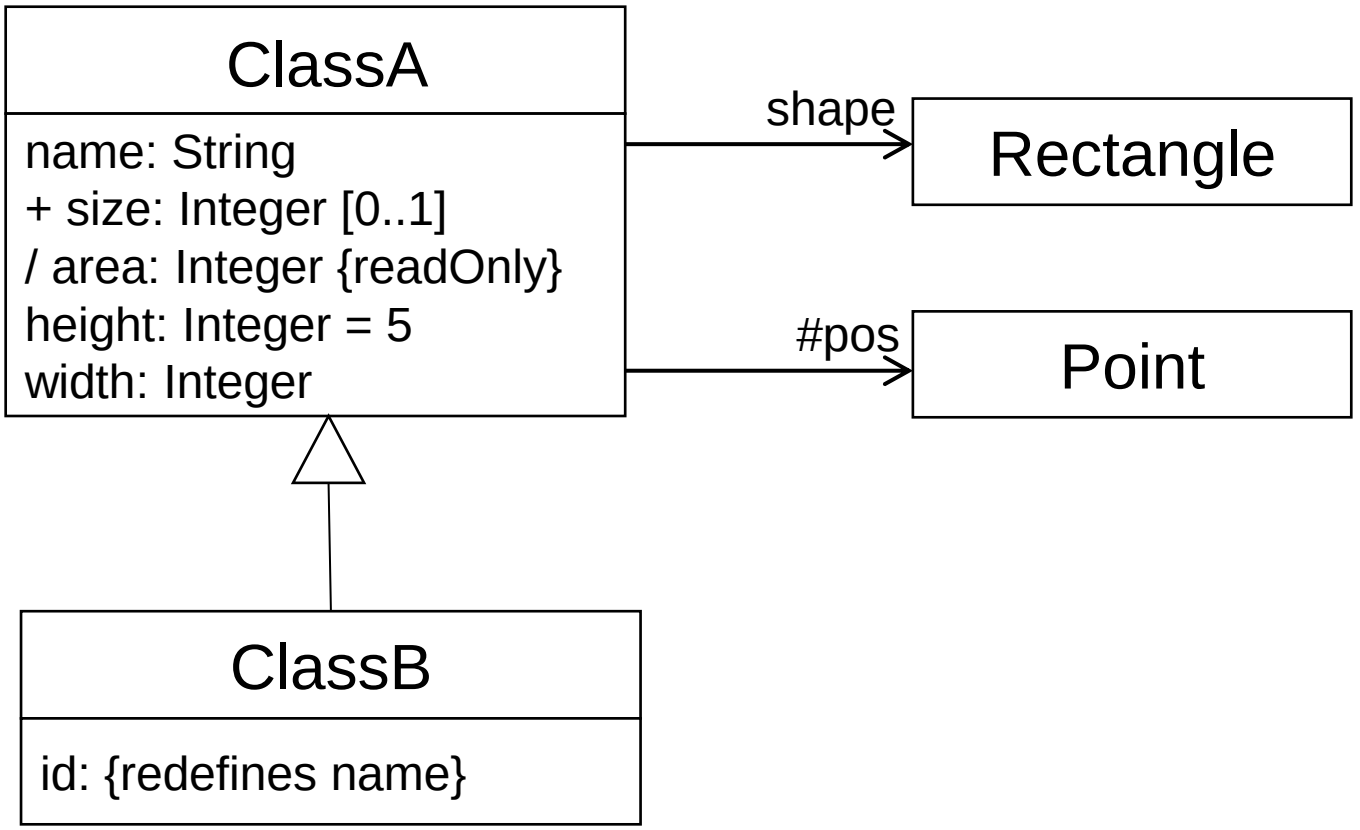
Classes

Attributes



Classes

Preferred Notation



Classes

Methods

- Notation for methods:

[visibility] name ([parameter-list]) : [type] [{property}]

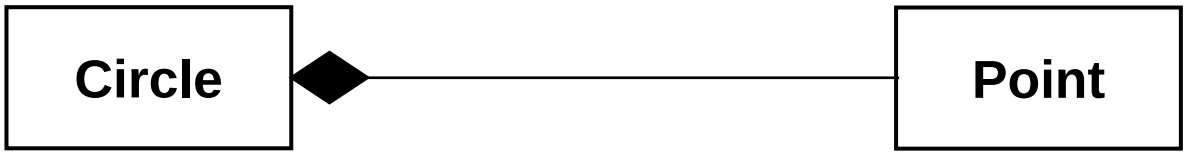
- Visibility (optional).
- Method name
- List of formal parameters, separated by commas:
  - [direction] name : type [multiplicity] = value [{property}]
- Static methods are underlined.

- Examples:

display ()
-hide ()
+createWindow (location: Coordinates, container: Container [0..1]): Window
+toString (): String

Associations: Composition

- A *circle* **contains** a *point* (as its center).
- This relation is represented as **composition**

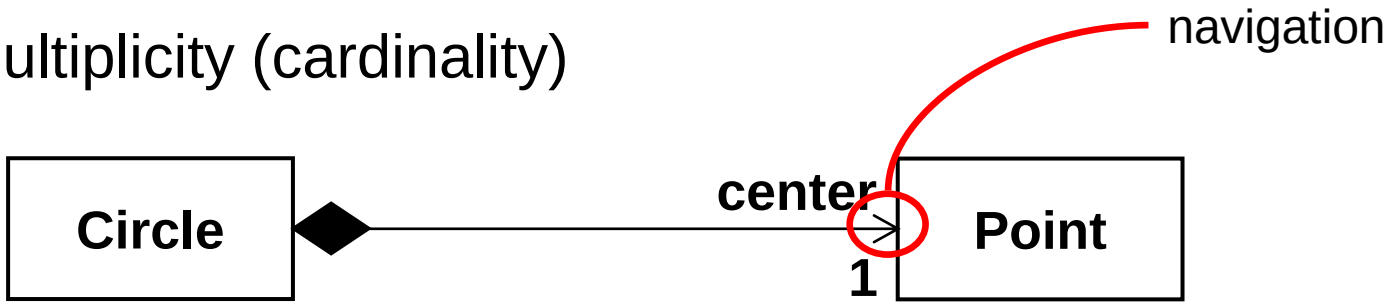


- Whole/part relation
  - The *whole* is the circle.
  - The *part* is the point.
- It is a strong relation:
  - If the circle is destroyed ot copied, so is the Point.
  - Cardinality in the whole part is 0..1 or 1.

Associations

Navigation, Roles, Cardinality

- Associations can be tagged:
  - Roles in the relation
  - Multiplicity (cardinality)



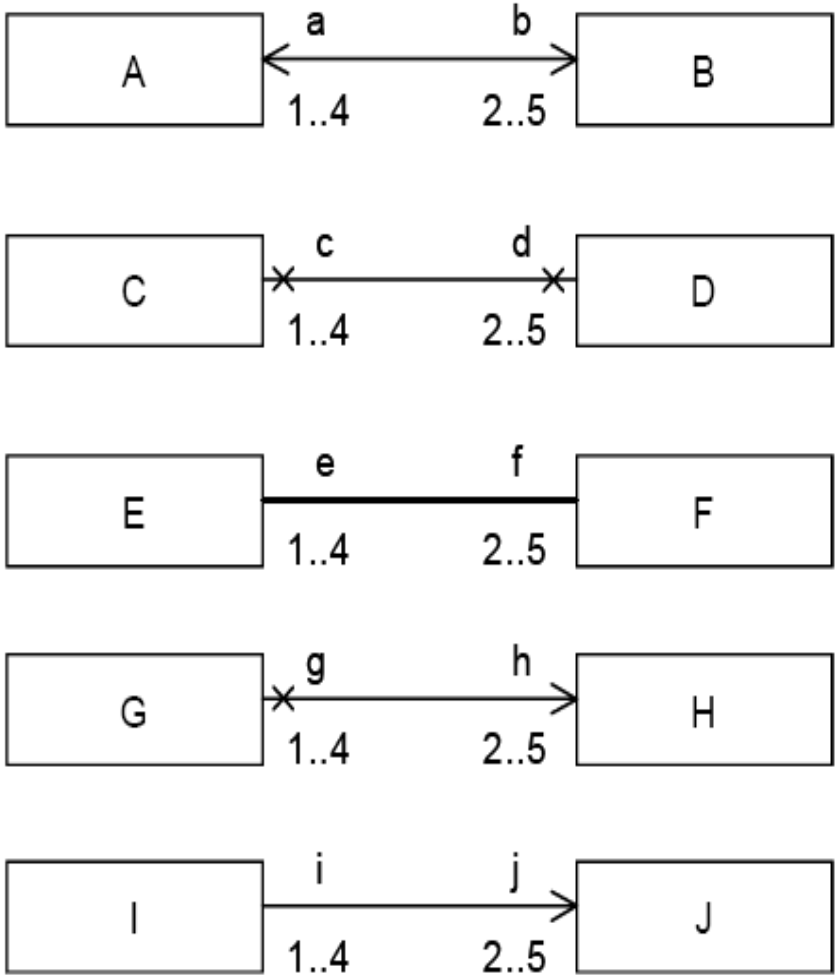
- Cardinality examples:

1..*	minimum 1, no maximum
0..*	minimum 0, no maximum
0..1	minimum 0, maximum 1
1,2,4	one, two or four
3	exactly three

- Navigation:
  - Unidirectional
  - Bidirectional
  - Not specified.
  - Not navigable (x)

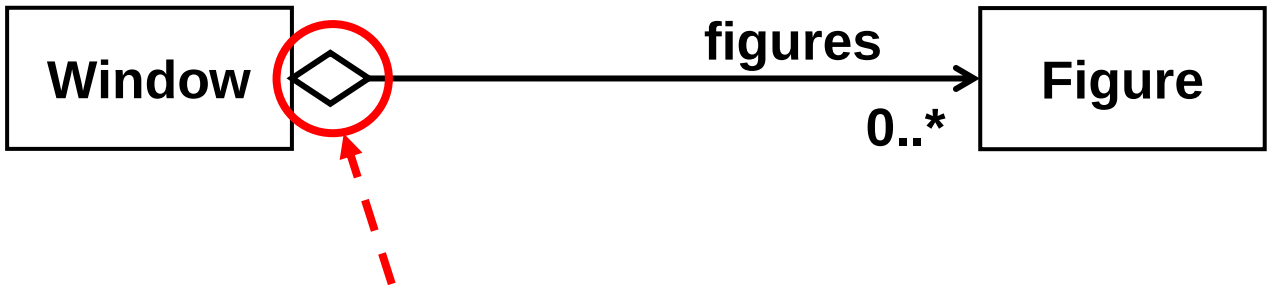
Associations

Navigation and Cardinality Examples



Associations: Aggregation

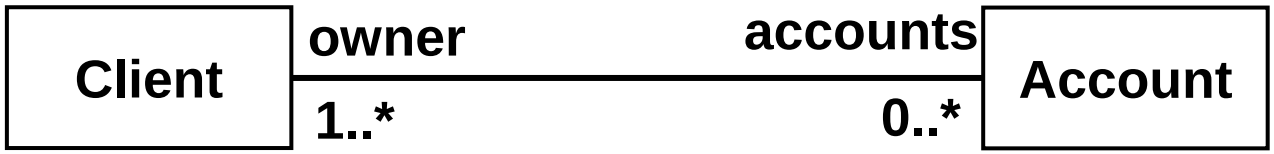
- When the whole/part relation is not so strong **Aggregation** is used

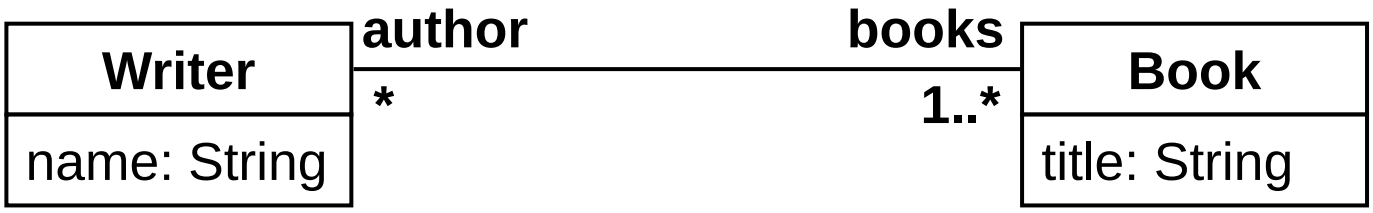


The window contains figures, but each one of them can exist without each other

Associations

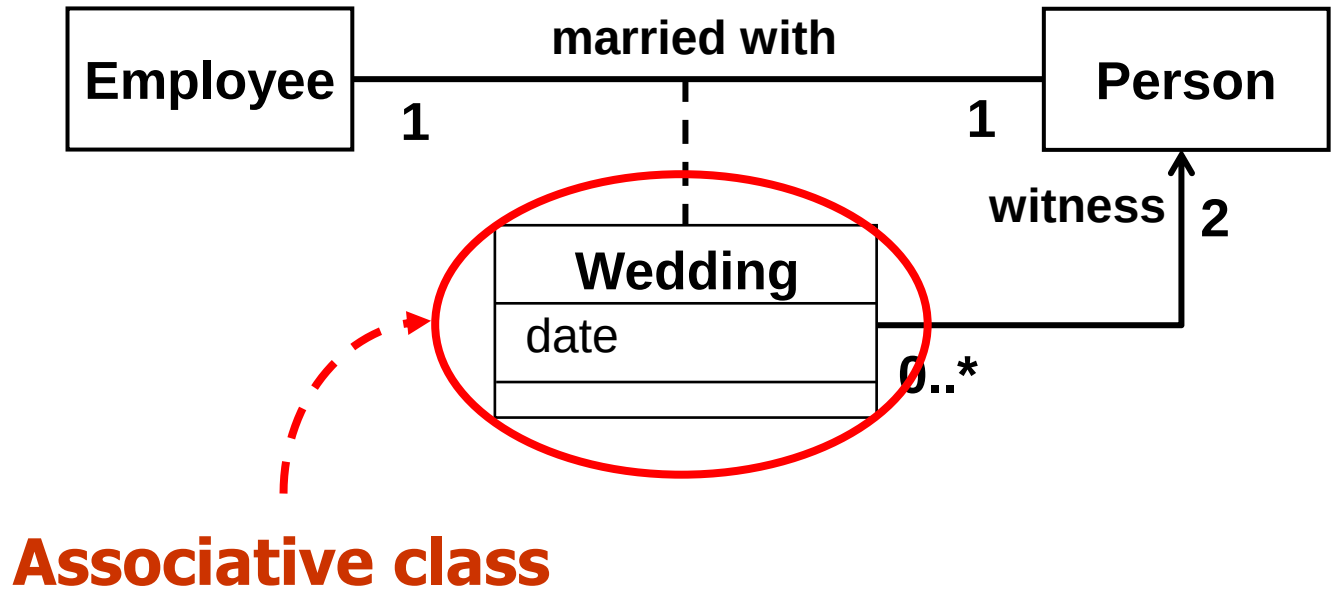
- There are relations that do not imply a whole/part relationship





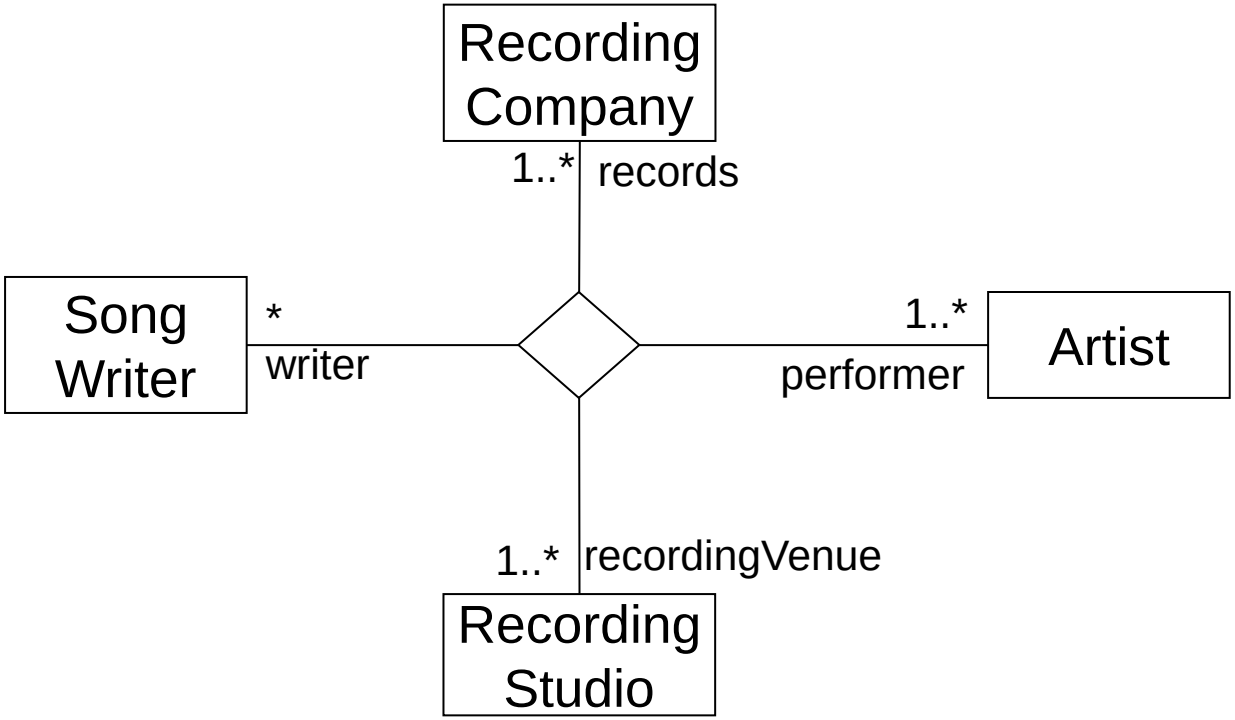
Associative Classes

- Association with attributes

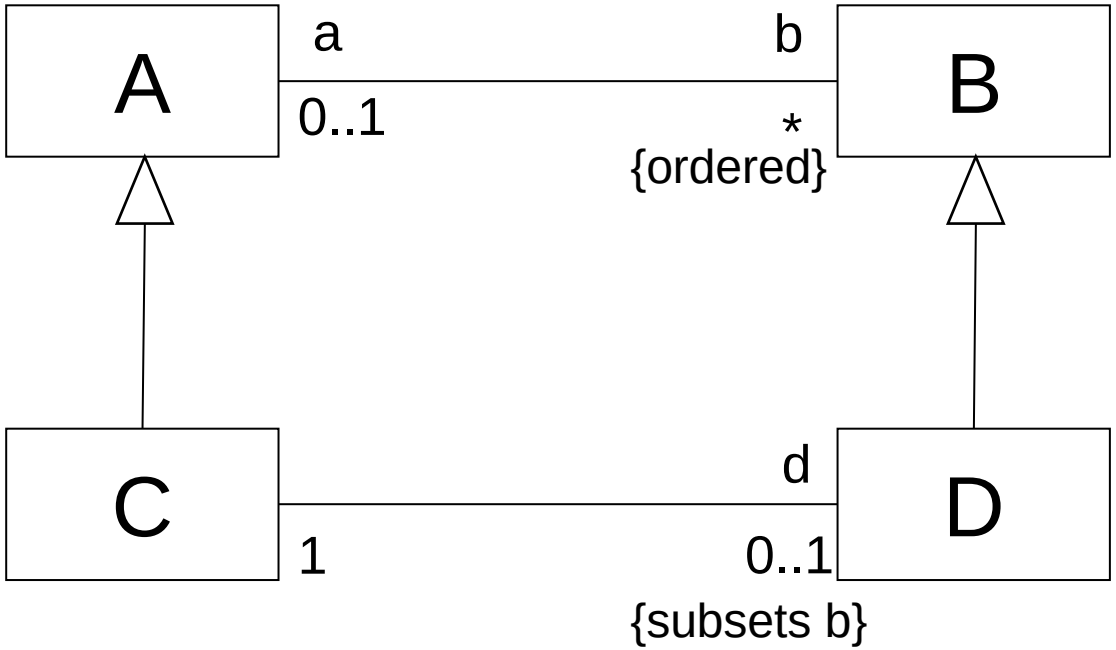


N-ary associations

- Associations between more than two classes

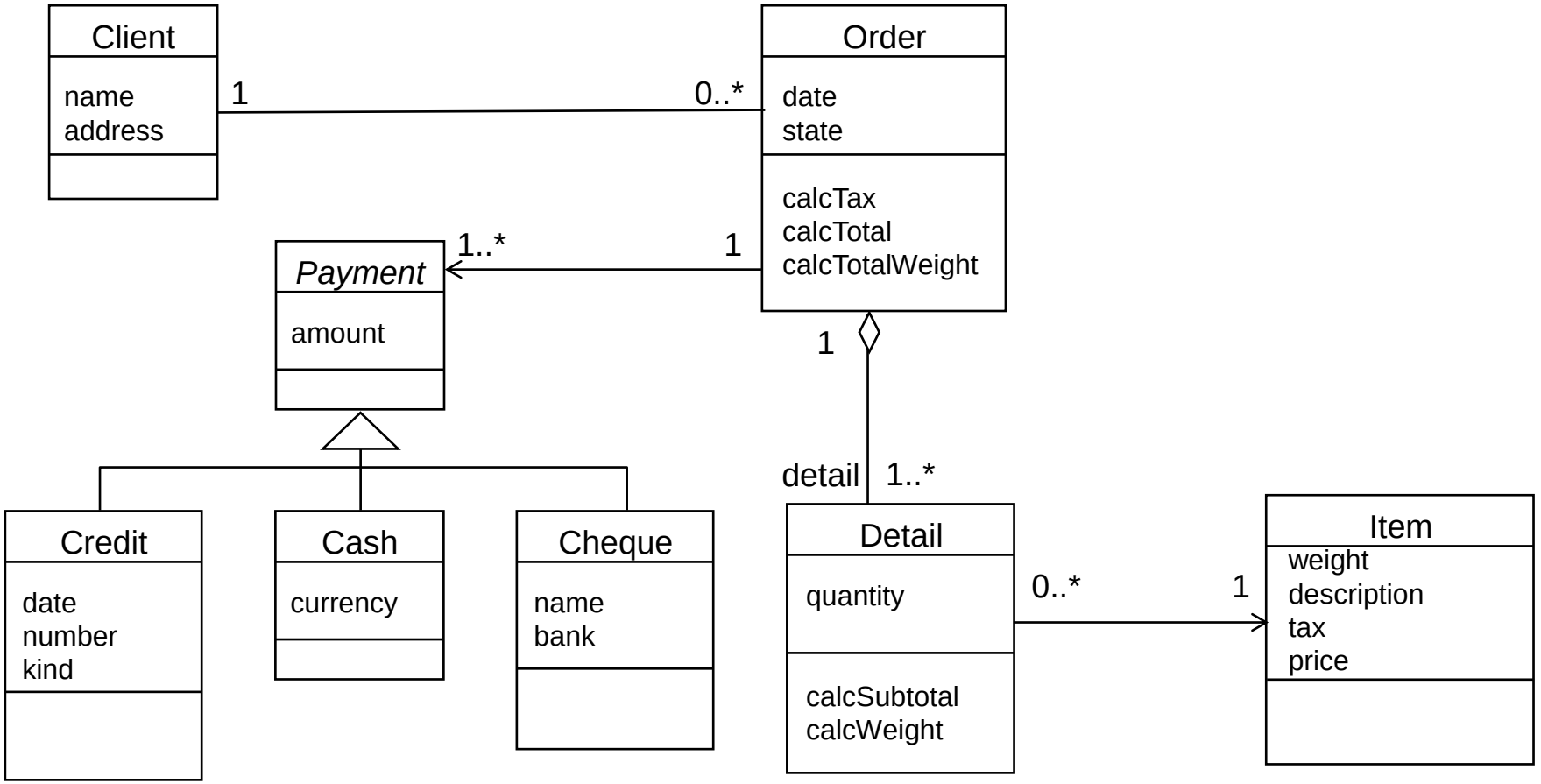


Role properties



- For an object of type C, the collection d is a subset of the collection b.

Example


Client
name
address
1
0..*
Order
date
state
calcTax
calcTotal
calcTotalWeight
Payment
amount
1..*
1
1
detail
1..*
Credit
date
number
kind
Cash
currency
Cheque
name
bank
Detail
quantity
calcSubtotal
calcWeight
0..*
1
Item
weight
description
tax
price

Classes and Objects

Style

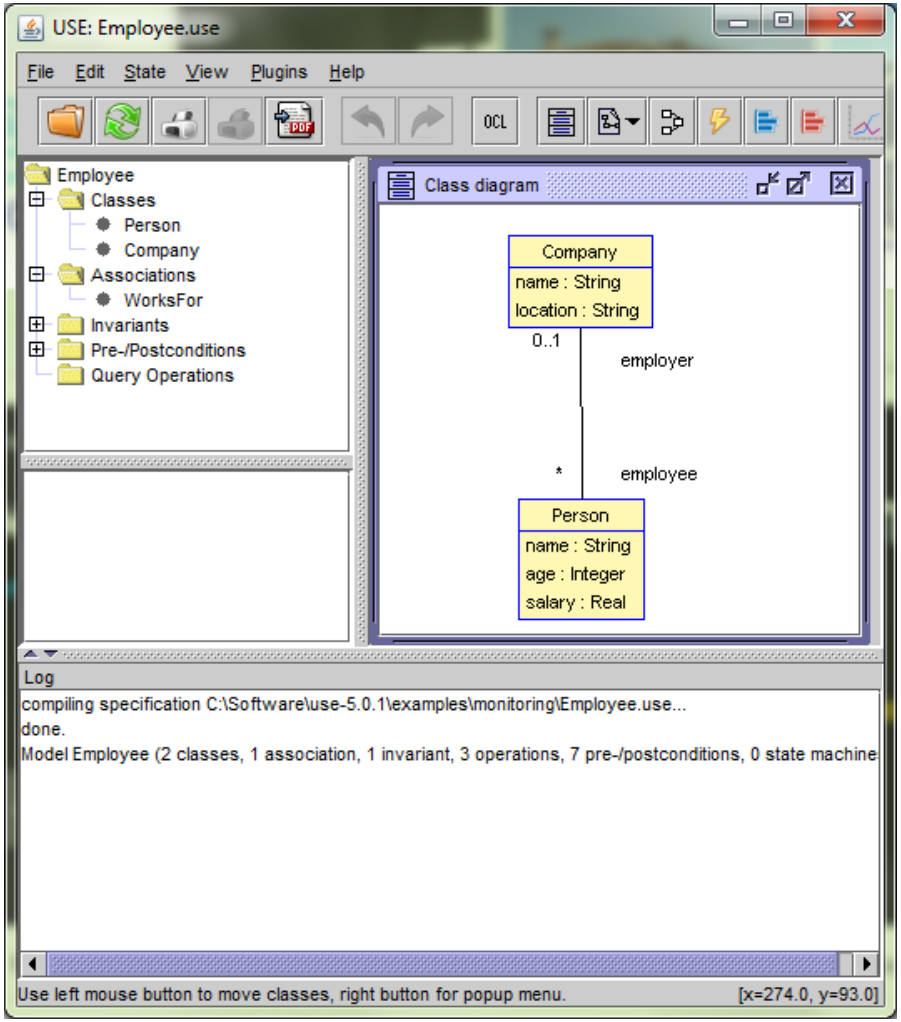
- Use attributes for primitive types and relations for objects
- Class diagrams do not normally include:
  - Constructors
  - Getters and Setters
  - Methods to handle the elements of an association or aggregation (e.g., “add/remove”)
  - …
- …because these are implementation details and do not belong to the design (higher level of abstraction).

Exercise (in class)

- A factory is made of machines of three different kinds: generators, assemblers and packagers.
- Machines are connected through conveyors, which transport parts from one machine to another.
- Conveyors can connect multiple (but at least one) incoming machines to multiple (but at least one) outgoing machines.
- Parts have a serial number.
- A conveyor can hold parts up to its maximum capacity.
- A generator cannot have incoming conveyors, and a packager does not have outgoing conveyors.
- Any machine can be operated by at most one operator (workers of the factory). Each operator operates at most one machine at the same time. At every moment, there should be at least one operator in some machine.

- Some of these requirements are difficult to capture using UML alone… we will use OCL to solve this.

USE (UML-based specification environment)

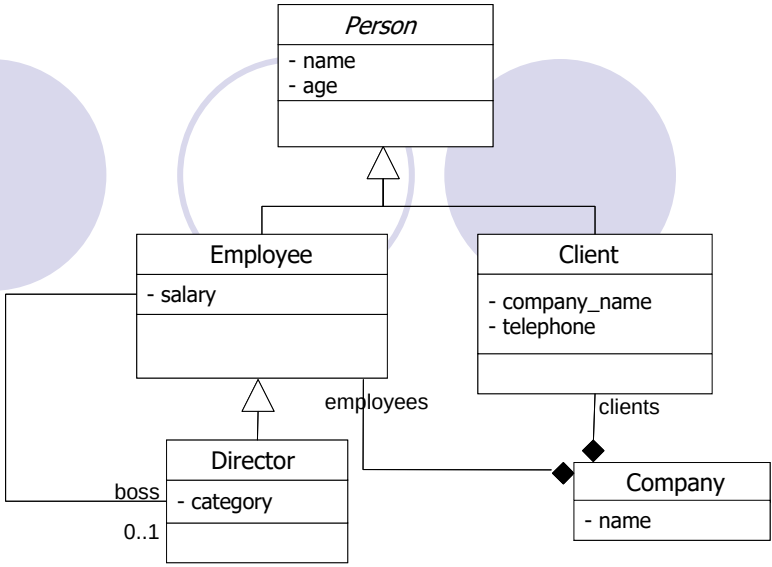


https://sourceforge.net/projects/useocl/

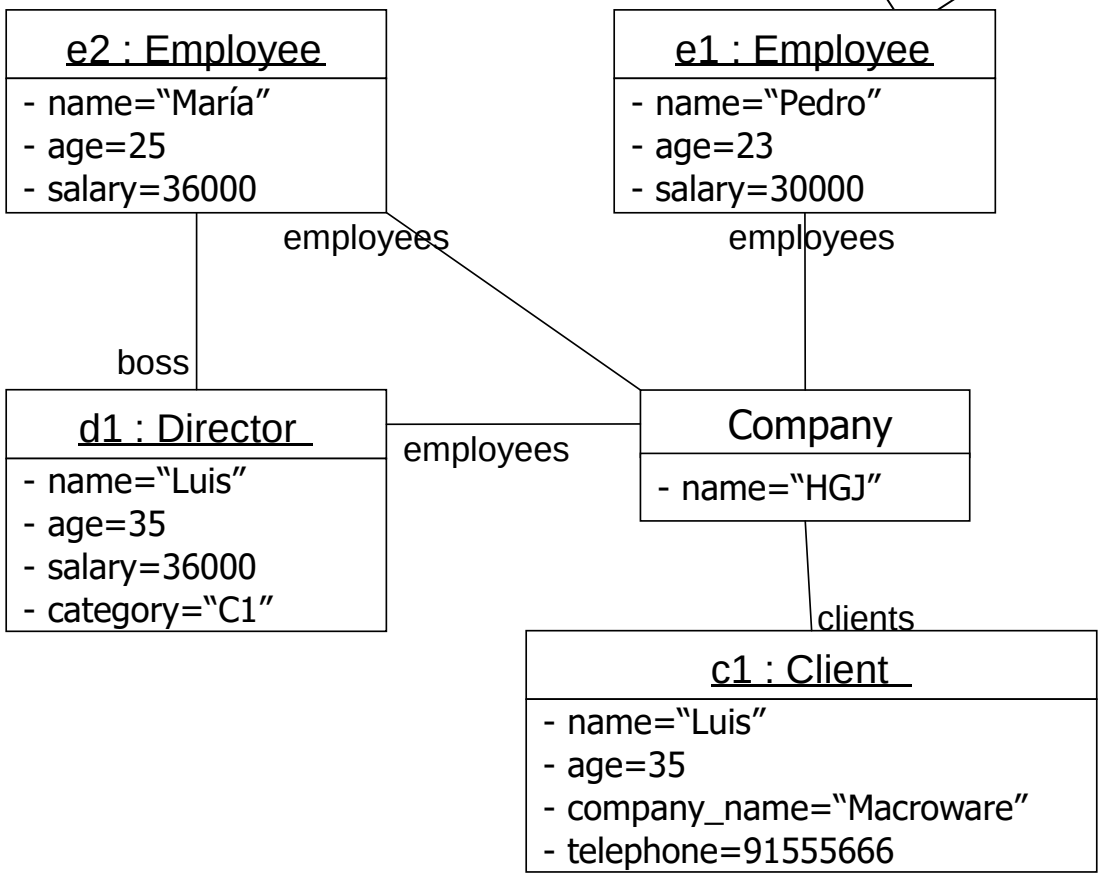
Employee.use

```
model Employee

-- classes
class Person
attributes
  name : String
  age : Integer
  salary : Real
operations
  raiseSalary(rate : Real) : Real
end

class Company
attributes
  name : String
  location : String
operations
  hire(p : Person)
  fire(p : Person)
end

-- associations
association WorksFor between
  Person[*] role employee
  Company[0..1] role employer
end
```

Objects



conforms to



Index

- Introduction.
- Class and Object Diagrams.
- **Other Diagrams.**

- OCL: The Object Constraint Language.
- Bibliography.

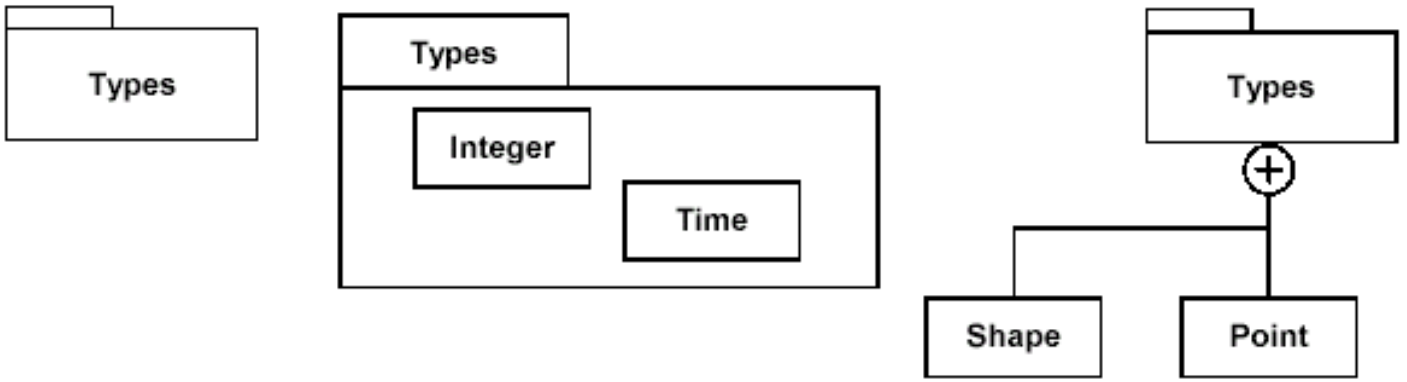
Structural Diagrams

- Classes and Objets
- **Packages**
  - Application structure.
- **Components**.
  - Modules with interfaces.
- **Composite Structures**
  - Classes with internal structure.
- **Deployment.**
  - Physical view.

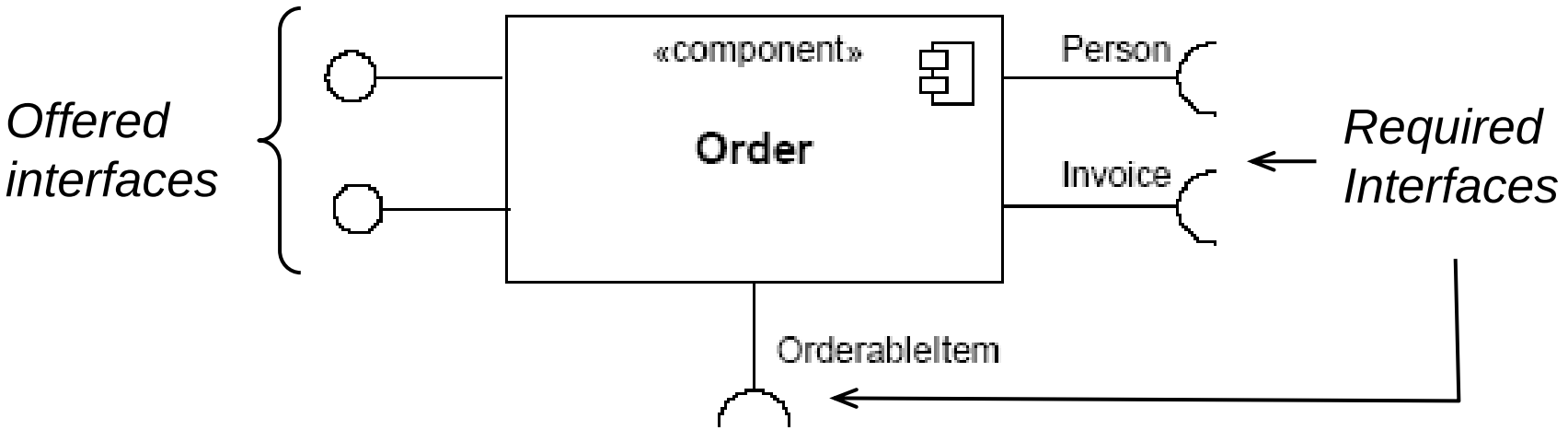
Packages

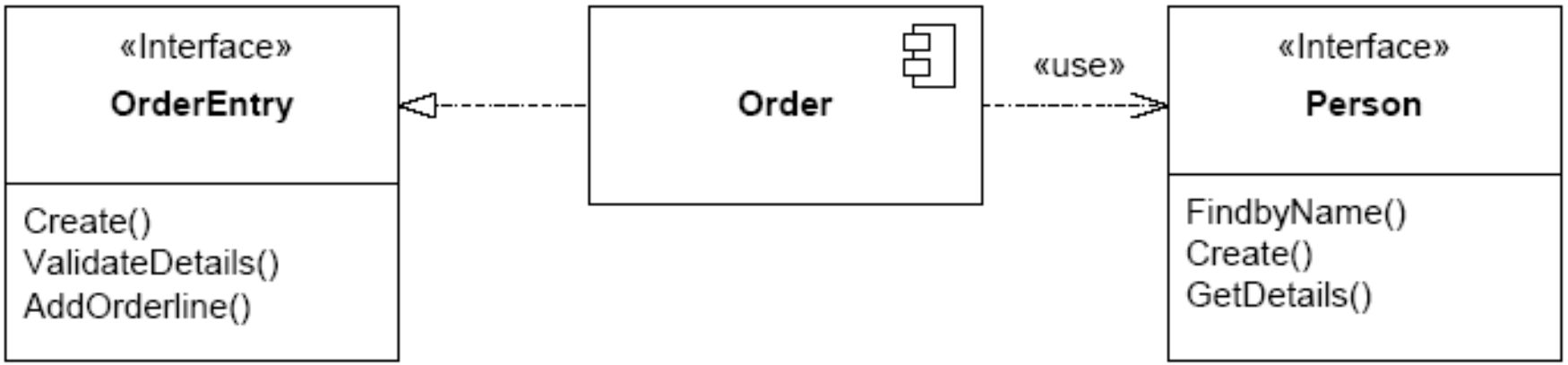
- A package is a container that groups related elements.

- Package diagrams show the high-level structure of the application.



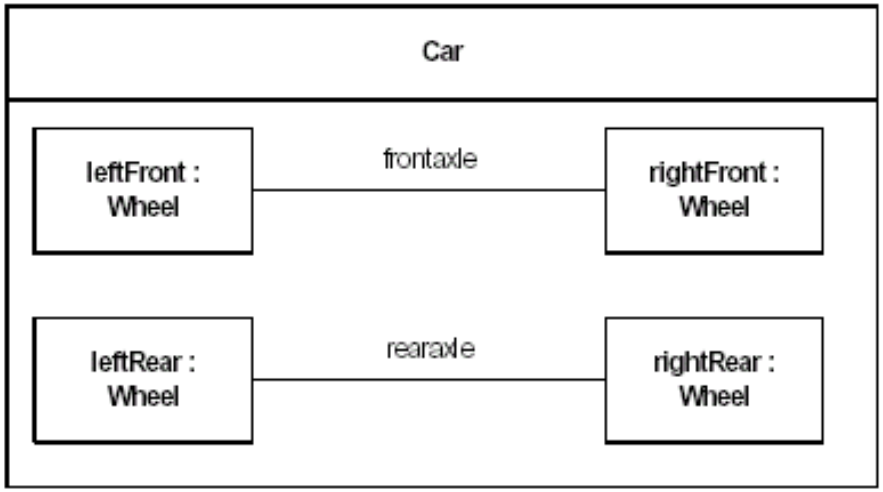
Components

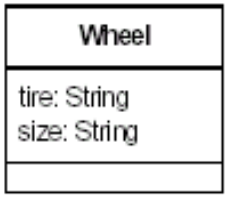

Offered interfaces
«component»
Order
Person
Invoice
Required Interfaces
OrderableItem

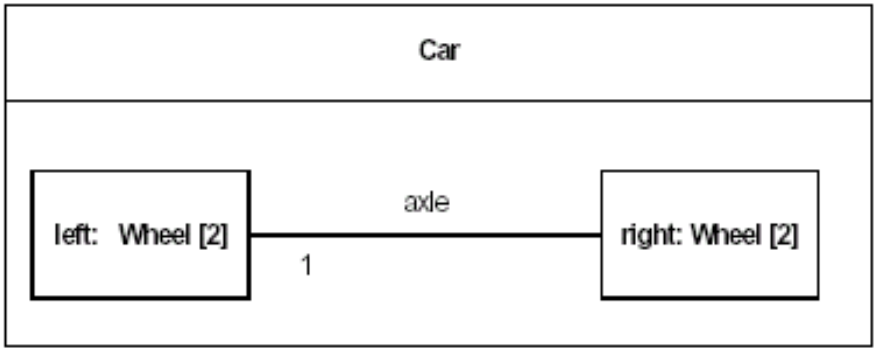

«Interface»
OrderEntry
Create()
ValidateDetails()
AddOrderline()
Order
«use»
«Interface»
Person
FindbyName()
Create()
GetDetails()

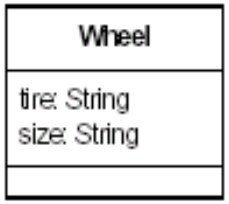
Composite Structures

- Internal Structure of a Class

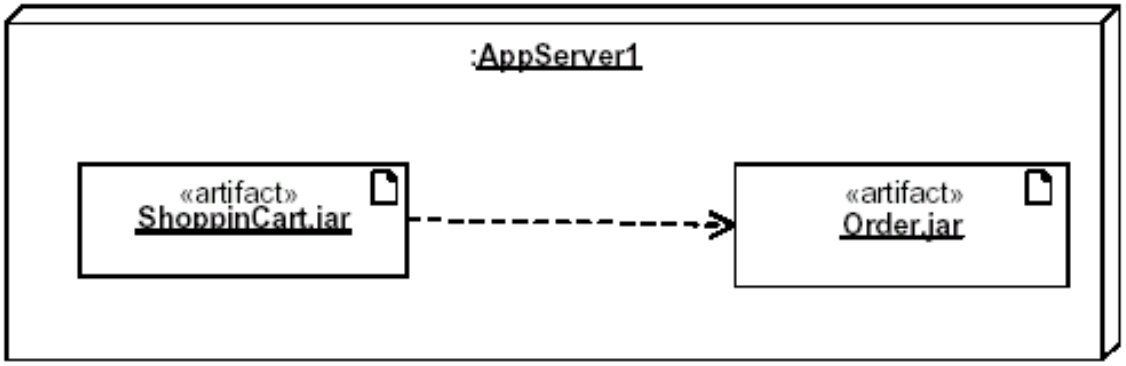


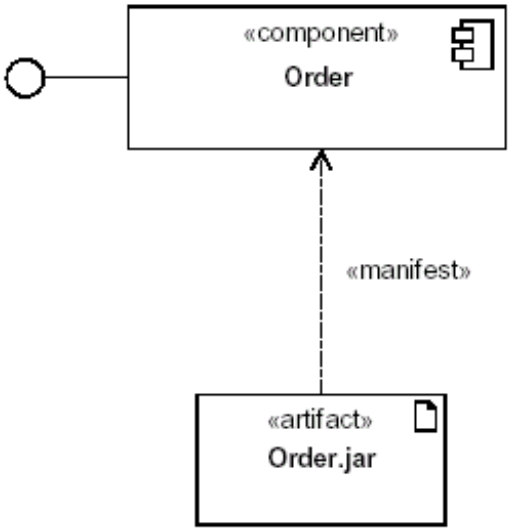


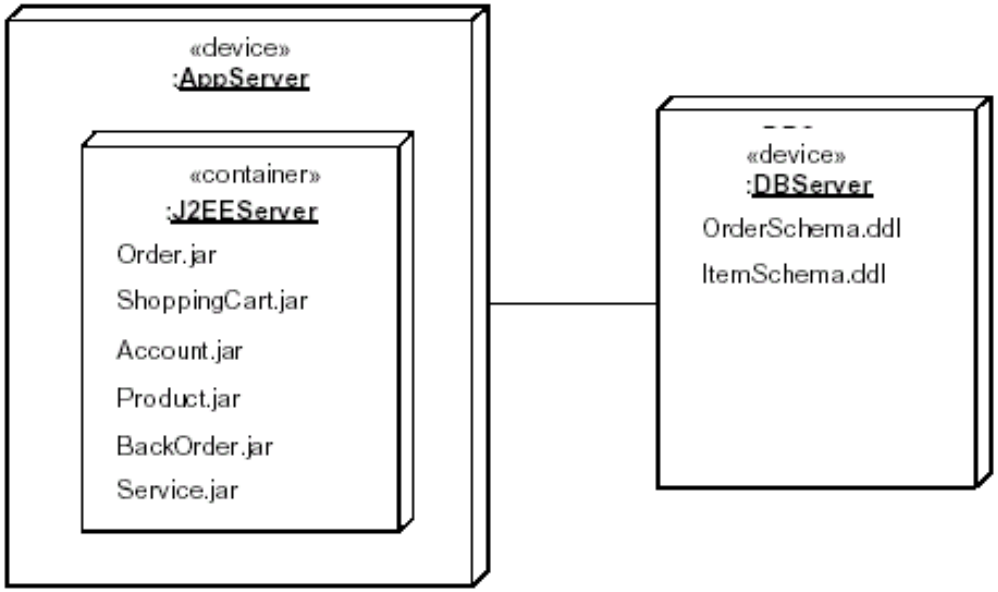




Deployment


:AppServer1
«artifact»
ShoppinCart.jar
«artifact»
Order.jar

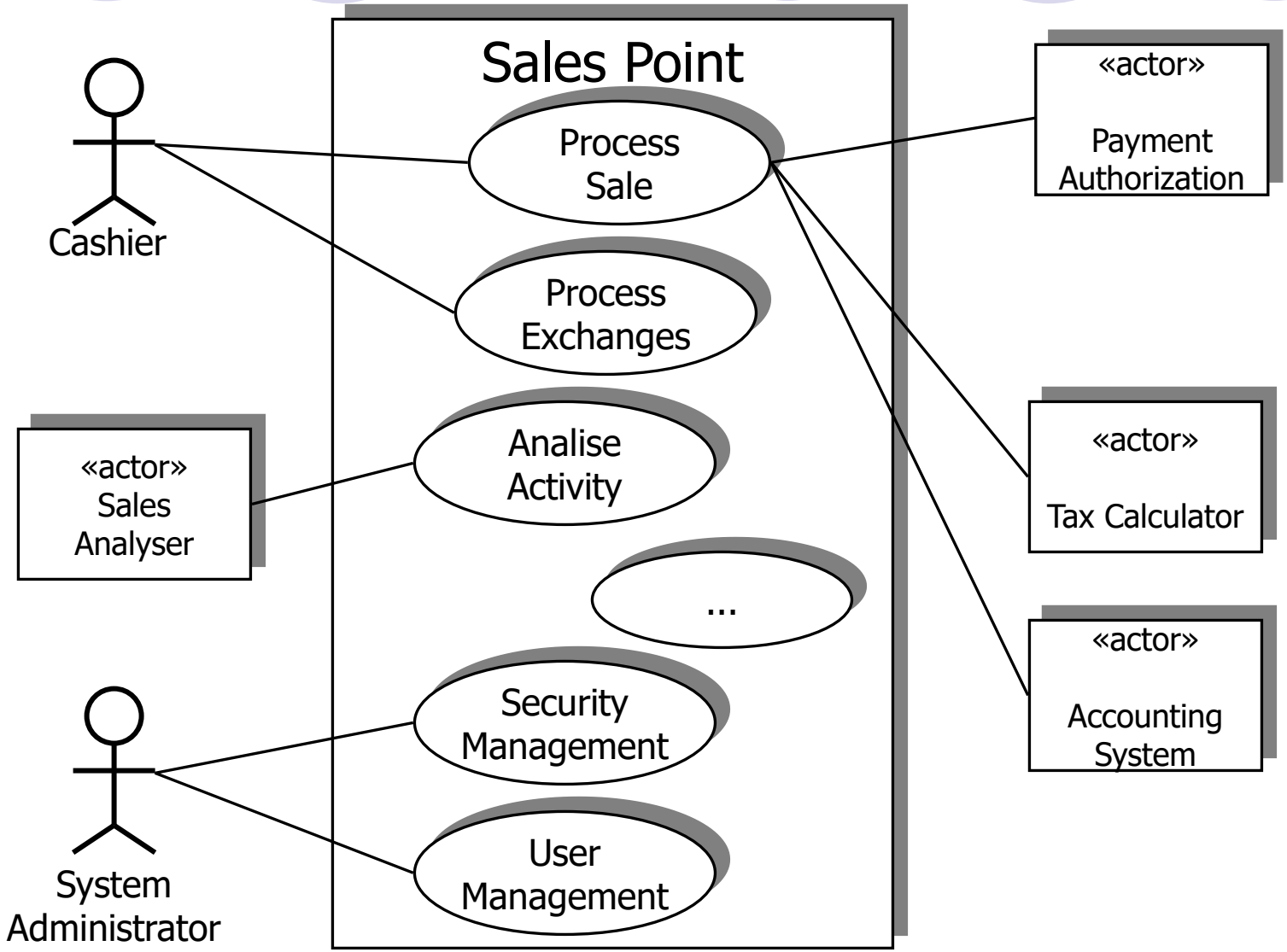

«component»
Order
«manifest»
«artifact»
Order.jar


«device»
:AppServer
«container»
:J2EEServer
Order.jar
ShoppingCart.jar
Account.jar
Product.jar
BackOrder.jar
Service.jar
«device»
:DBServer
OrderSchema.ddl
ItemSchema.ddl

Behavioural Diagrams

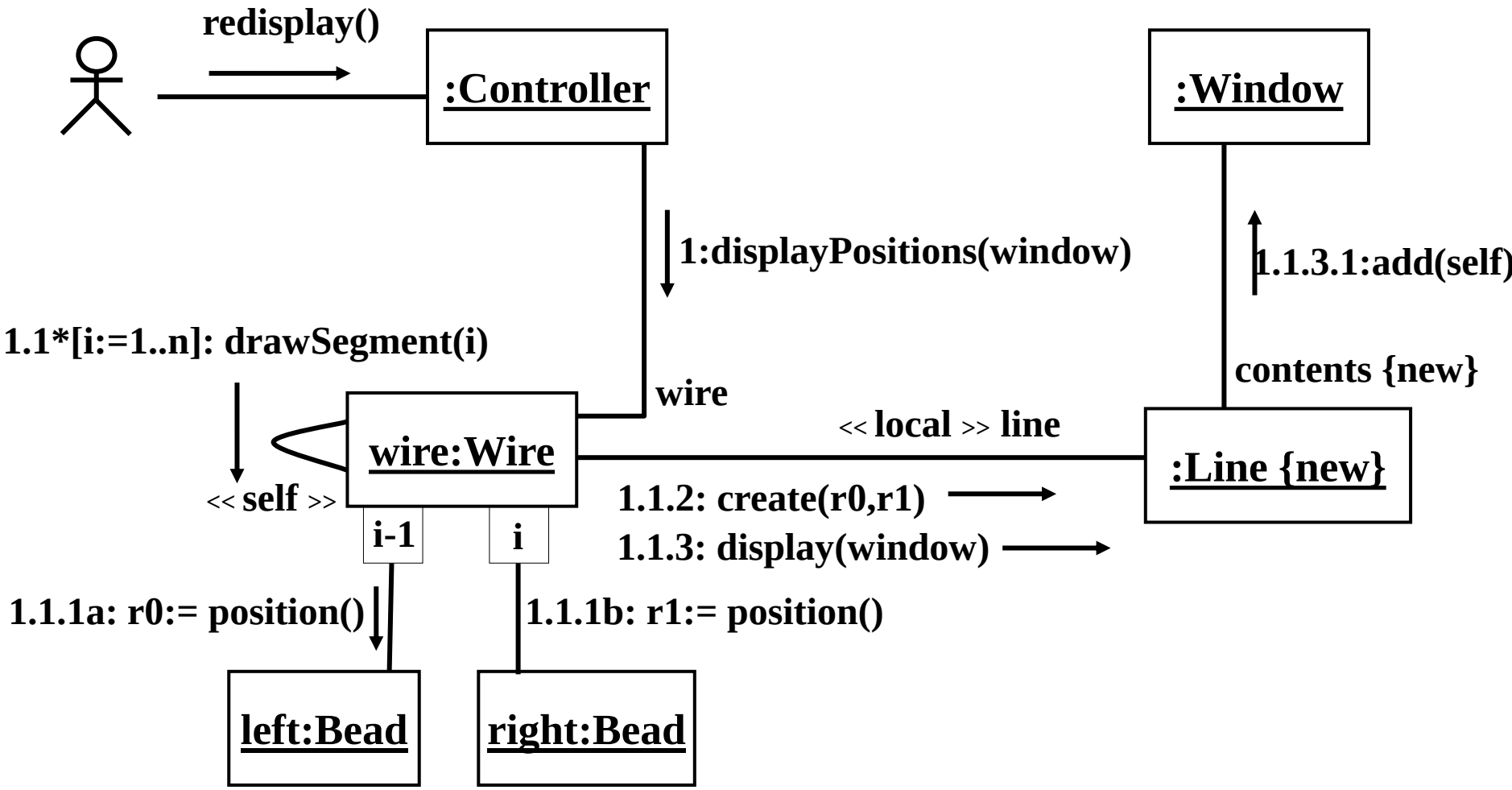
- ***Use case diagrams***
- ***Interaction diagrams***
  - Communication/Collaboration
  - Sequence
  - Interaction overview
  - Time
- ***State machines*** (*Statecharts*).
- ***Activity Diagrams.***

Use Cases

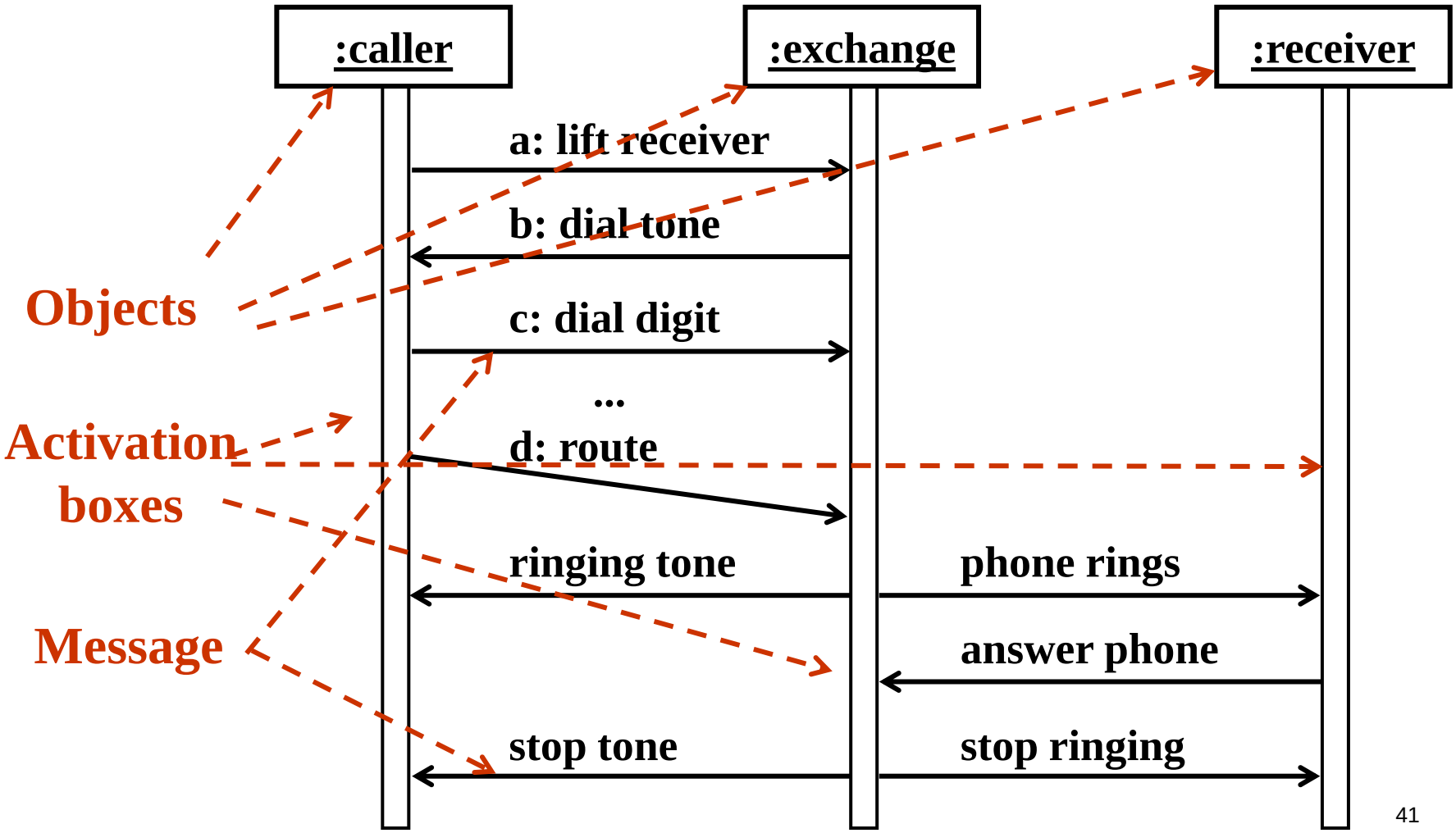

Sales Point
Process Sale
Process Exchanges
Analise Activity
...
Security Management
User Management
Cashier
«actor» Sales Analyser
System Administrator
«actor» Payment Authorization
«actor» Tax Calculator
«actor» Accounting System

Collaboration Diagram

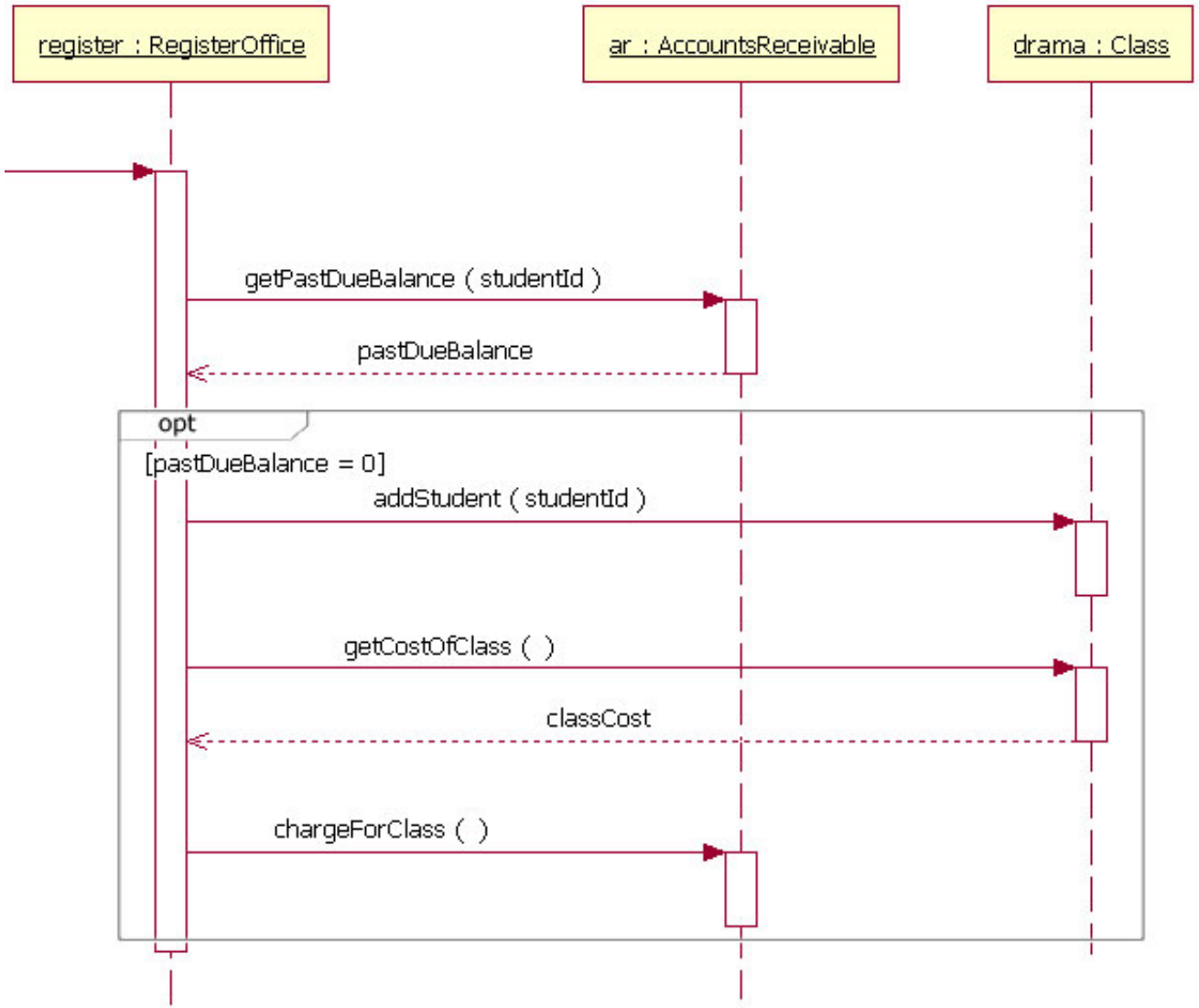
Example.



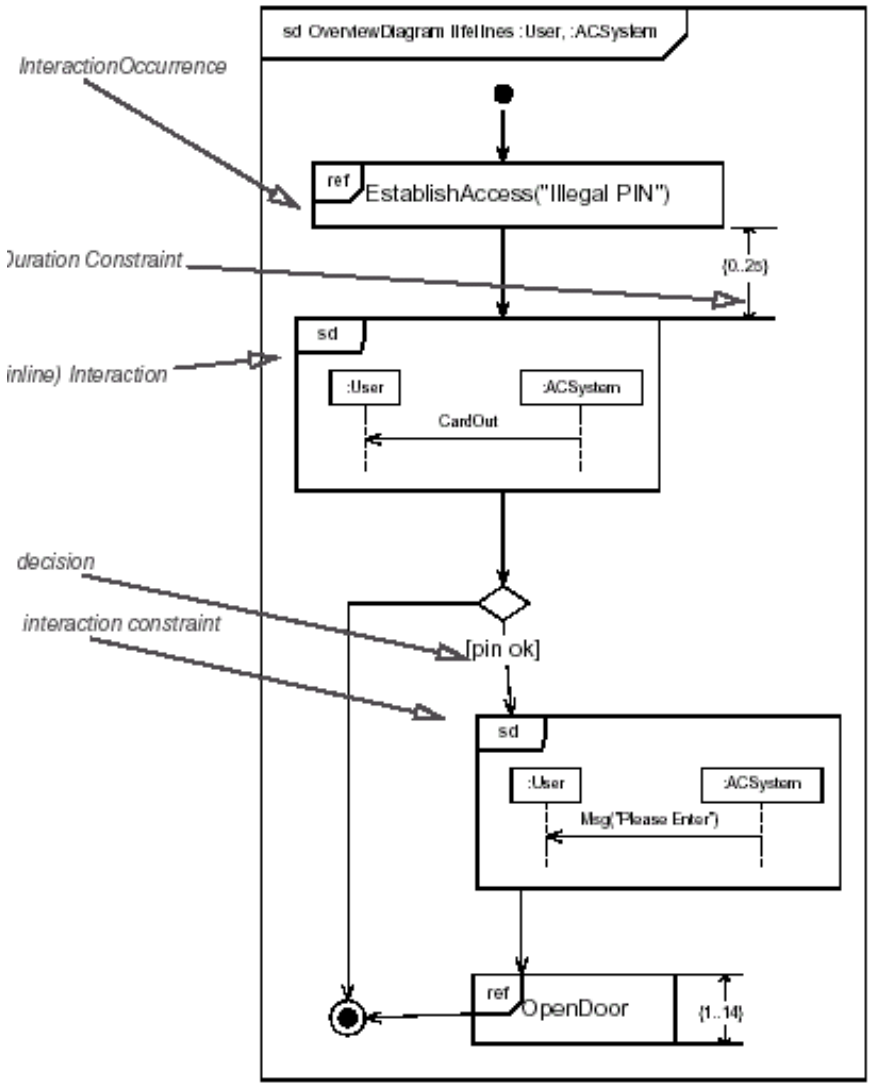
Sequence Diagrams



Sequence Diagrams



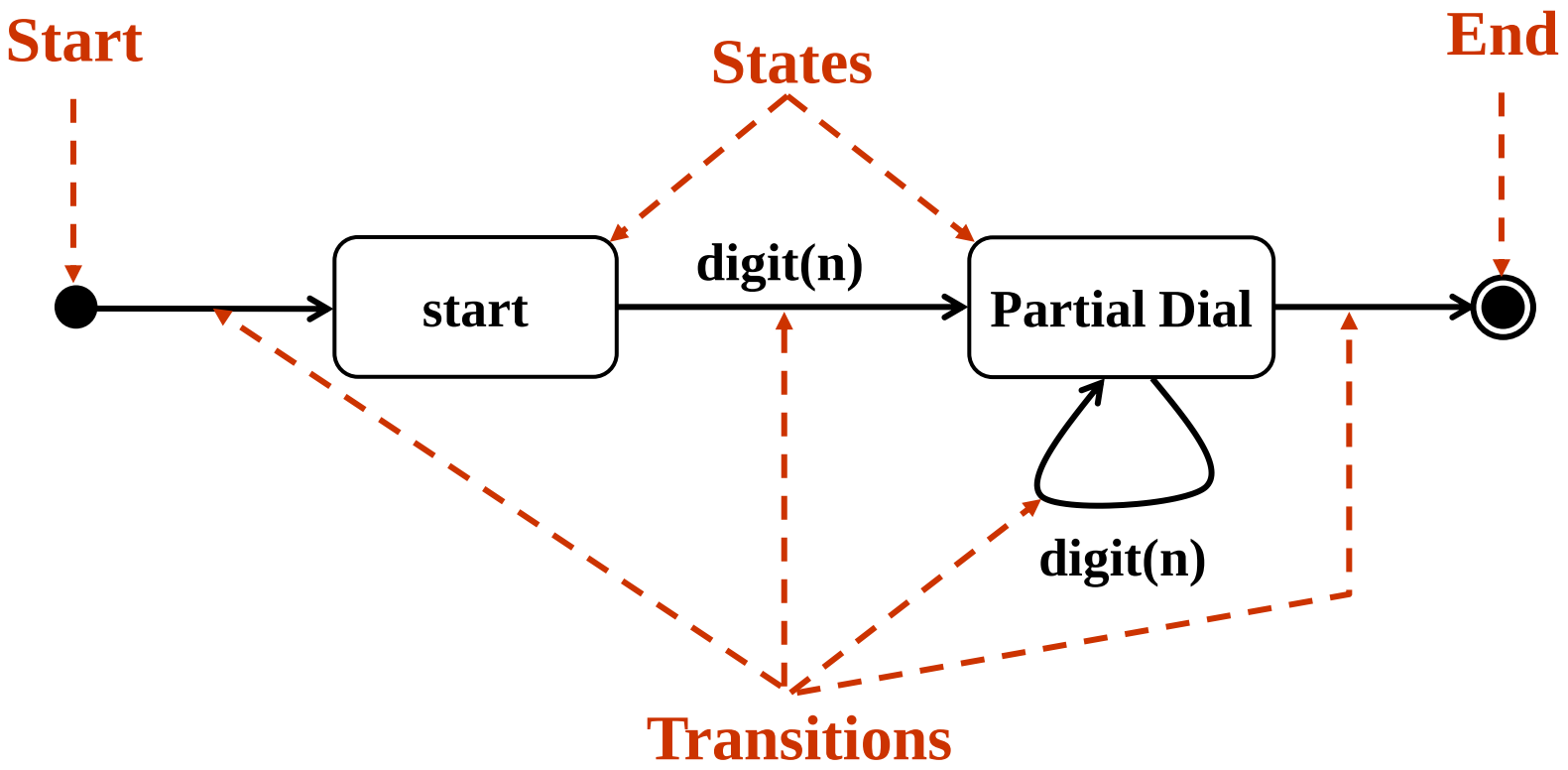
Interaction Overview Diagram



State Machines

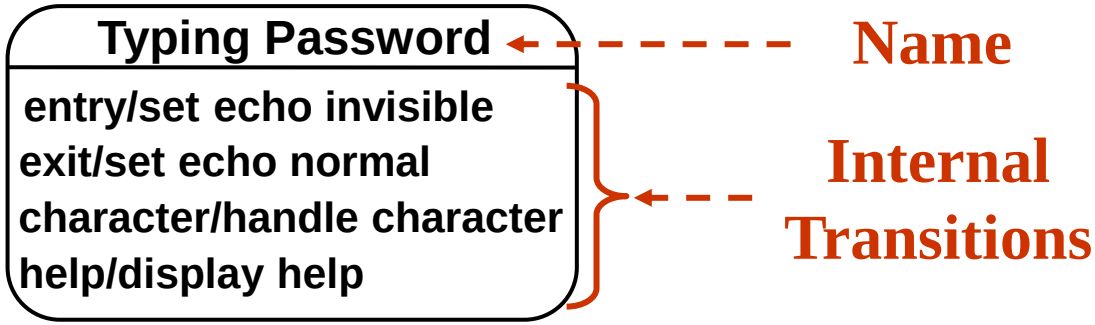
- "*Statecharts*" [Harel]
- Describe the behaviour of entities (e.g., objects).
- Specify reaction upon events.
- Describe the possible state sequences and actions that entities can go through.

State Machines

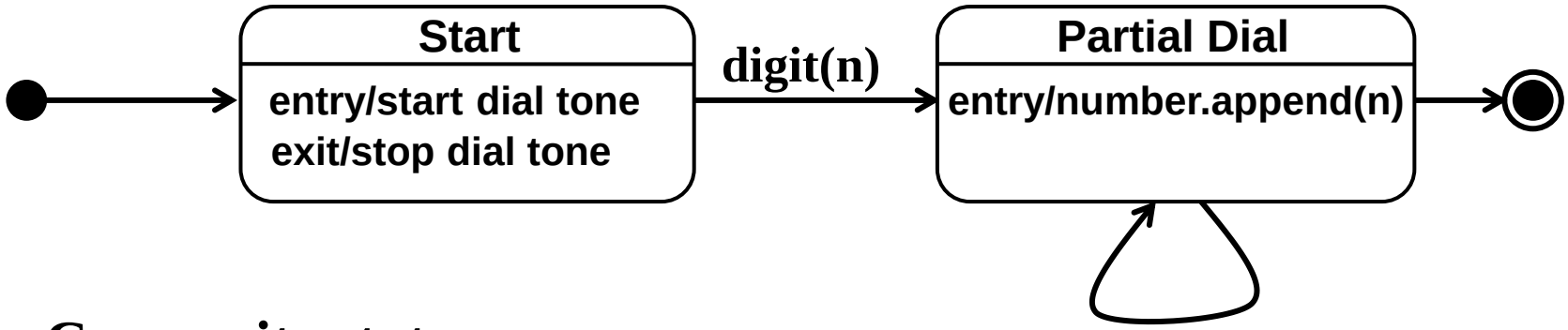


State Machines

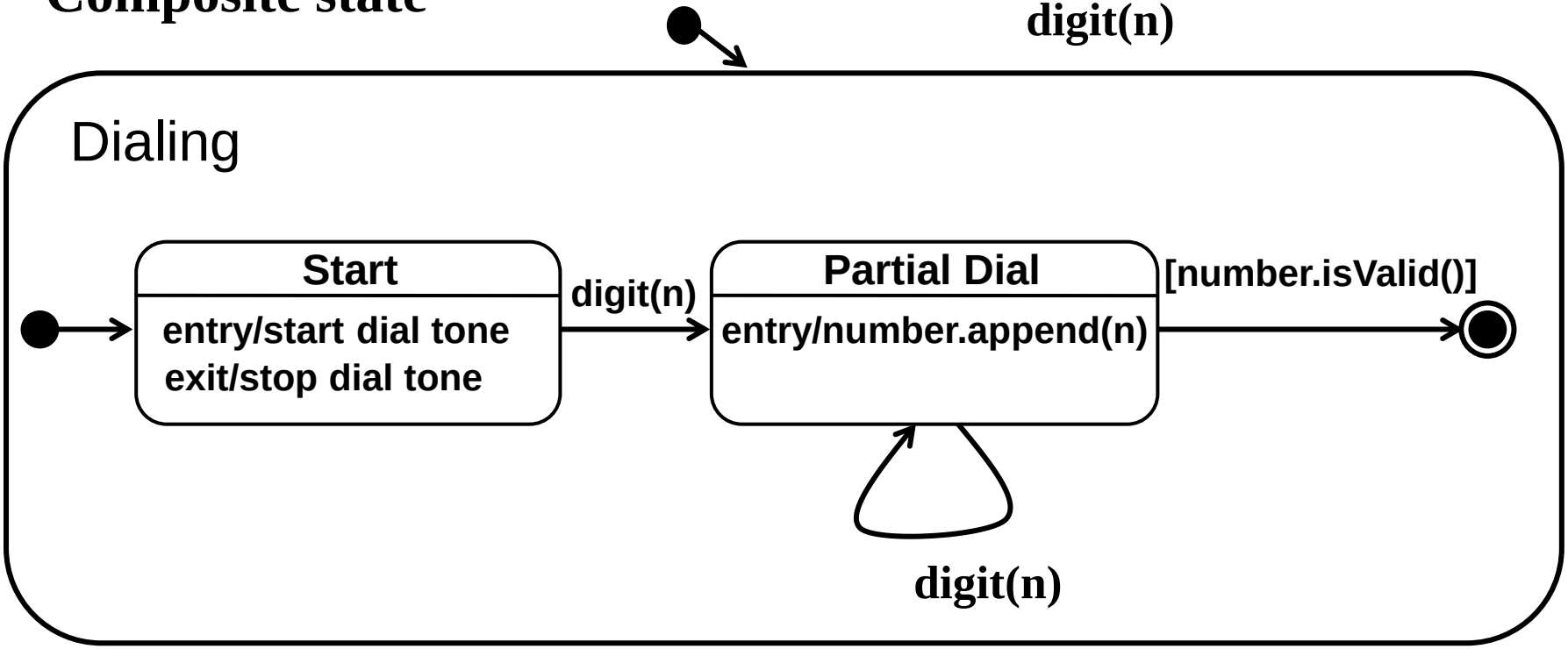
- A state has:
  - Name
  - Internal transitions: List of actions to be executed in this state (entry/exit/do)
- Example:



State Machines

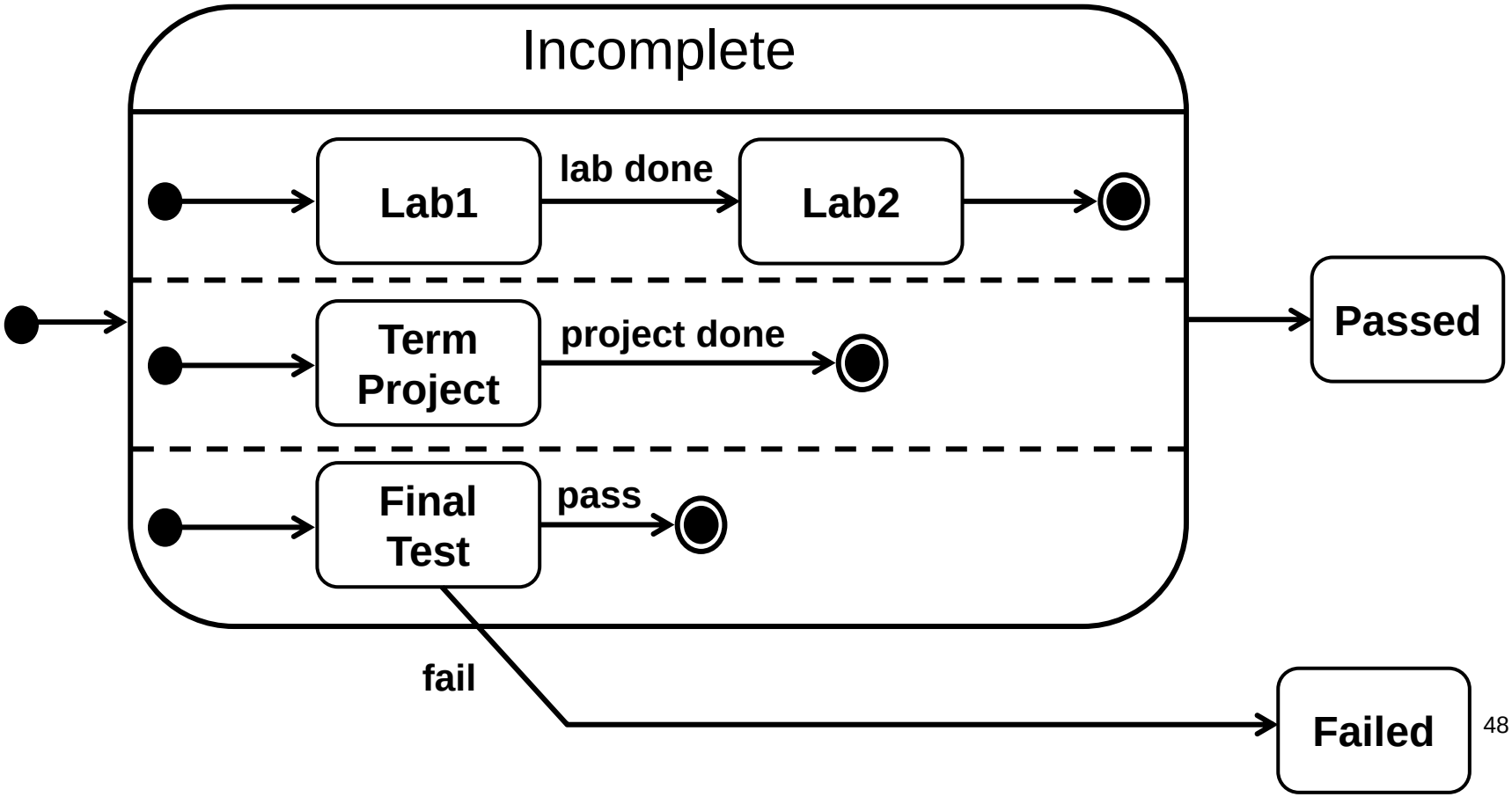


Composite state

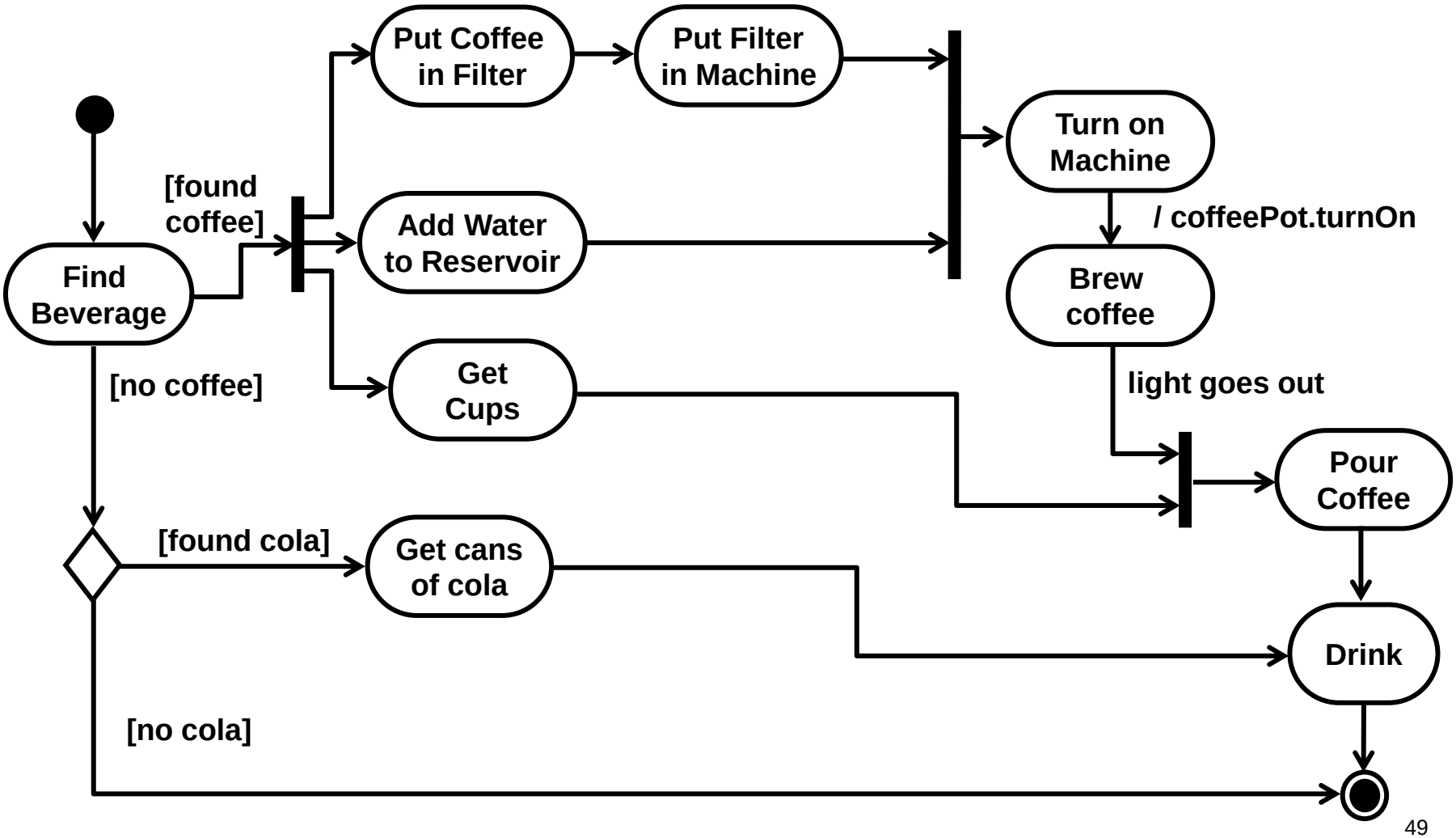


State Machines

Orthogonal Components

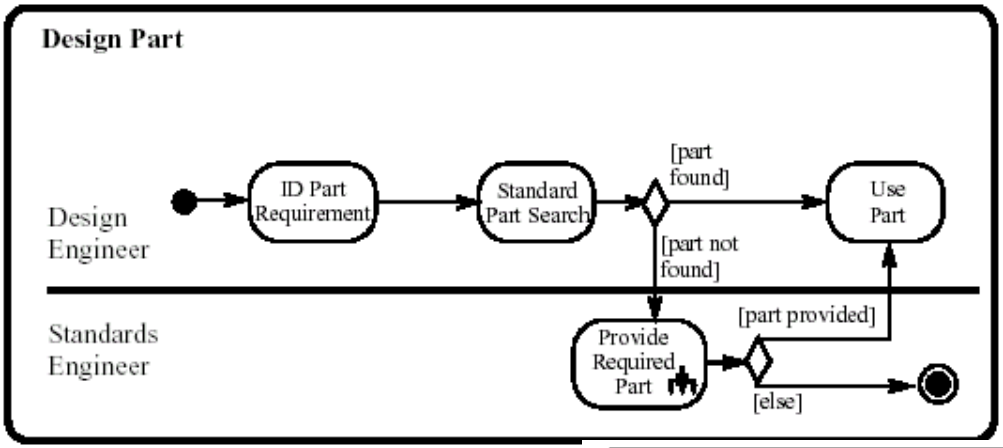


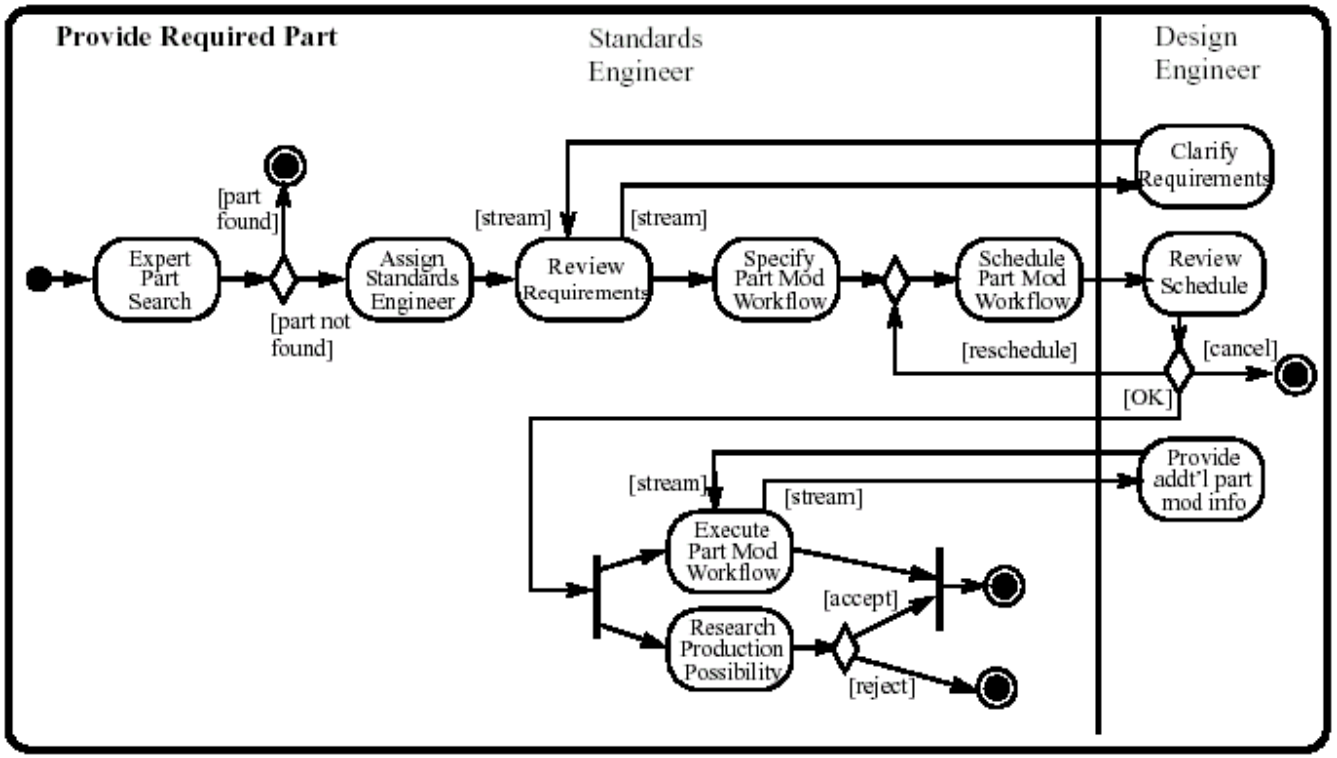
Activity Diagrams



Activity Diagrams

Swimlanes





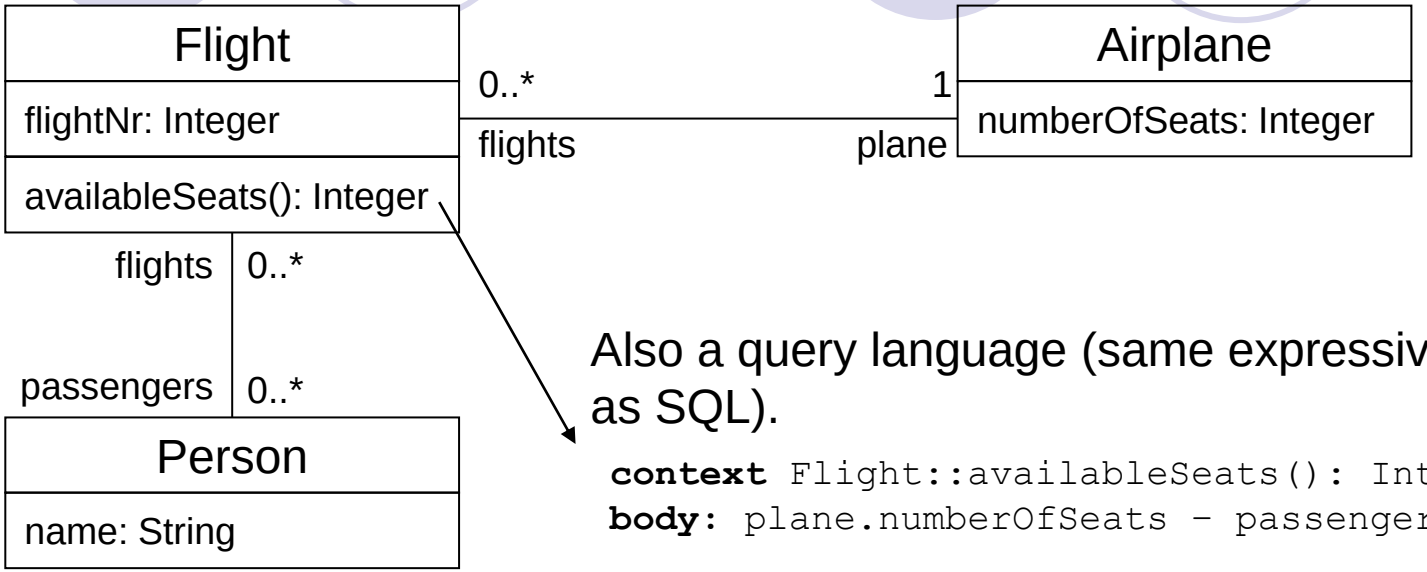
Index

- Introduction.
- Class and Object Diagrams.
- Other Diagrams.
- **OCL: The Object Constraint Language.**
- Bibliography.

OCL: Object Constraint Language

- Constraint language to express additional conditions that we cannot express diagramatically
- Combine diagrams and textual specifications
- Language based on first order predicate logic and set theory
- Useful for MDE: precise models (instead of annotations in natural language)
- It is mostly used together with class diagrams

Examples



Also a query language (same expressive power as SQL).

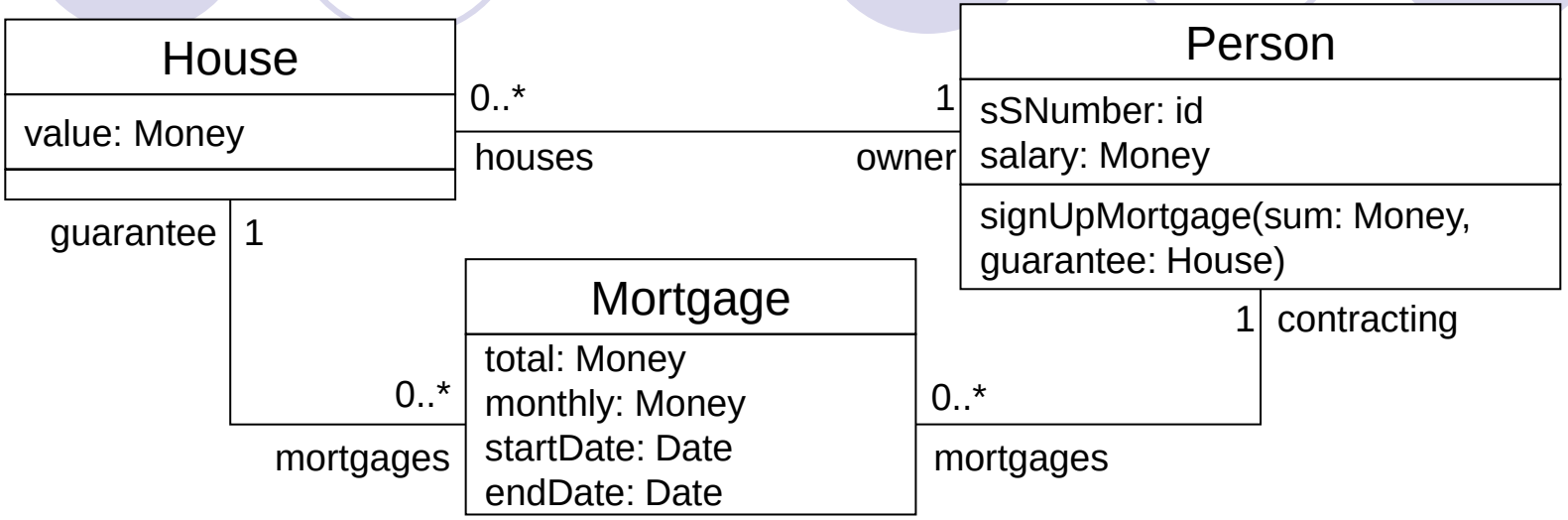
```
context Flight::availableSeats(): Integer
body: plane.numberOfSeats – passengers->size()
```

- How do we express the fact that no flight can have more passengers than airplane seats?

- OCL constraint:

```
context Flight
inv: passengers->size() <= plane.numberOfSeats
```

Examples



Additional Rules:

1. A person may hold a mortage on a house only if he is the owner.
2. The start date of each mortgage should be before the end date.
3. The social security number of each person is unique.
4. It is only allowed to sign up a new mortgage if the person salary is enough.
5. It is only allowed to sign up a new mortgage if the value of the guarantee house is enough.

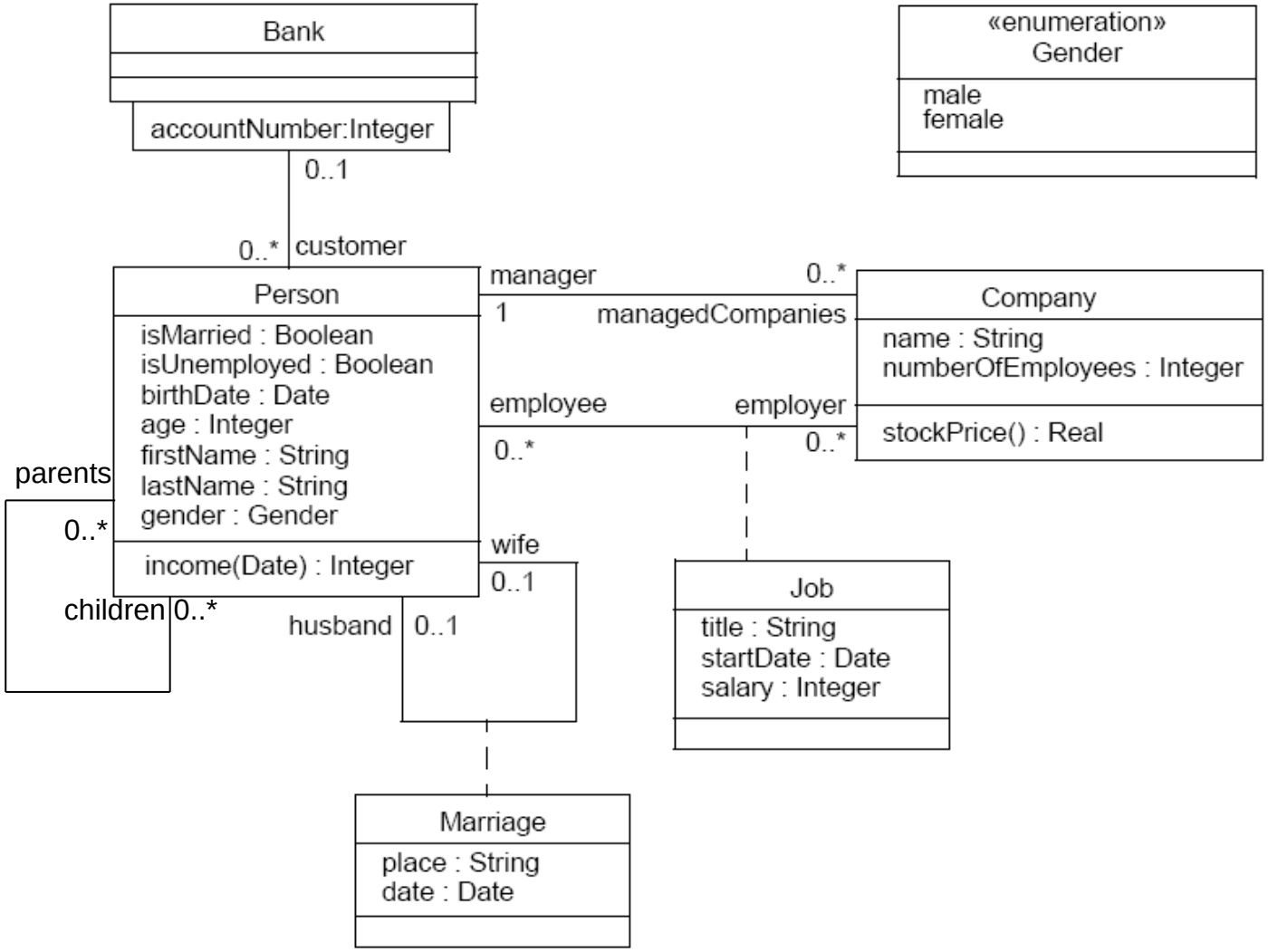
Examples

OCL constraints are written in the context of a specific object.

```
context Mortgage
inv: guarantee.owner = contracting

context Mortgage
inv: startDate < endDate

context Person
inv: Person::allInstances()->isUnique(sSNumber)

context Person::signUpMortgage(sum: Money, guarantee: House)
pre: self.mortgages.monthly->sum()+sum <= self.salary * 0.70

context Person::signUpMortgage(sum: Money, guarantee: House)
pre: guarantee.value >= guarantee.mortgages.total->sum()
```

self references the instance in the context.

Running Example



Basic data types

Type	Value	Operations
		e1 = e2, e1 <> e2
Integer	1, -5, 34	i+i2, i-i2, i*i2, i.div(i2), i.mod(i), i.abs( ), i.max(i2), i.min(i2), <, >, <=, >=
Real	1.5, 1.34, ...	r+r2, r-r2, r*r2, r/r2, r.floor( ), r.round( ), r.max(r2), r.min(r2), <, >, <=, >=
Boolean	true, false	not b, b and b2, b or b2, b xor b2, b implies b2
String	'' a string''	s.size(), s.concat(s2), s.substring(i1,i2), s.toUpper(), s.toLower(), s.toInteger(), s.toReal()
Enumeration	Day::Monday, Day::Tuesday,	=, <>
TupleType(x : T1, y : T2, **z** : T3 **)**	Tuple { y : T2 = …, x = … , z = … }	t.x t.y t.z

(from Charles André OCL cheat sheet at: http://www-sop.inria.fr/members/Charles.Andre/CAdoc/ESINSA/UMLOCL-memo.pdf)

Collections

- Set, OrderedSet, Bag, Sequence.

- Loop operations with collections:

select(expr): Selects the elements making expr true

collection->select(logic-expression)
collection->select(v | logic-expression-with-v)
collection->select(v : Type | logic-expression-with-v)

```
context Company inv:
self.employee->select(age < 25)->notEmpty()
```

```
context Company inv:
self.employee->select(gender=female)->notEmpty()
```

Collections

collect(expr): returns the collection resulting from evaluating expr on every element of the original collection.

```
self.employee->collect( birthDate )->asSet()
self.employee.birthDate->asSet()                -- shortcut
```

forAll(expr): returns true if expr evaluates to true on every element of the collection

```
collection->forAll( logic-expression)
collection->forAll( v | logic-expression-with-v)
collection->forAll( v : Type | logic-expression-with-v )
```

```
context Company
inv: self.employee->forAll( isUnemployed = false )
inv: self.employee->forAll( p | p.isUnemployed = false )
inv: self.employee->forAll( p : Person | p.isUnemployed = false )
```

Collections

exists(expr): returns true if there is at least one element for which expr evaluates to true

collection->exists(logic-expression)
collection->exists(v | logic-expression-with-v)
collection->exists(v : Type | logic-expression-with-v)

context Company
inv: self.employee->exists(age > 50)
inv: self.employee->exists(p | p.age > 50)
inv: self.employee->exists(p : Person | p.age > 50)

iterate(…): iterates over all elements of a collection

collection->iterate(elem : Type; acc : Type = <expresion> | logic-expression-with-elem-and-acc)

collection->collect(x : T | x.property)
collection->iterate(x : T; acc : T2 = Bag{} | acc->including(x.property))

Adds one element to a collection

Collections

- Other operations (I)
  - source->any(iterator|body)
    - source->select(iterator|body)->asSequence()->first()
  - source->collectNested(iterators|body)
    - Bag resulting from applying body to every element of source (results are not flattened).
  - source->isUnique(iterators|body)
    - True if body evaluates to a different value for every element in source.
  - source->isEmpty() / source->notEmpty()
    - True if source is/is not empty
  - source->one(expr)
    - Returns true if there is exactly one element of source satisfying expr.
    - **context** Company **inv**: employee->one(e | self.manager=e)
  - source->reject(expr)
    - Returns a collection with the elements from source that do not satisfy the condition.
    - **context** Company **inv**: employee->reject(e | e.age>=18)->isEmpty()
  - source->sortedBy(expr)
    - Sorts source, yielding an OrderedSet

Collections

- Other operations (I)
  - source->collectNested(iterators|body)
    - Bag resulting from applying body to every element of source (results are not flattened)
    - Assume a model with a Company that has two employees such as person1.children = {James, Jane} and person2.children = {John}.

(in context Company)
self.employees->collectNested(children.firstName)

→ Result:

Sequence{Sequence{James, Jane}, Sequence{John}}

Collections (inclusion/exclusion)

- Other operations (II)
  - source->includes(obj)
    - Returns true if collection source includes the object obj

Expression	Result
Sequence{2.3}->includes(2.1)	false
Sequence{2.0}->includes(2)	true

  - source->excludes(obj)
    - Returns true if collection source does not include the object obj
  - source->includesAll(collection)
    - Returns true if collection source includes all the objects in collection

Expression	Result
Sequence{2.3, 5.2, 'a', 3, null}->includesAll(Set{3, null})	true
Sequence{2.3, 5.2, 'a', 3}->includesAll(Set{3, null})	false

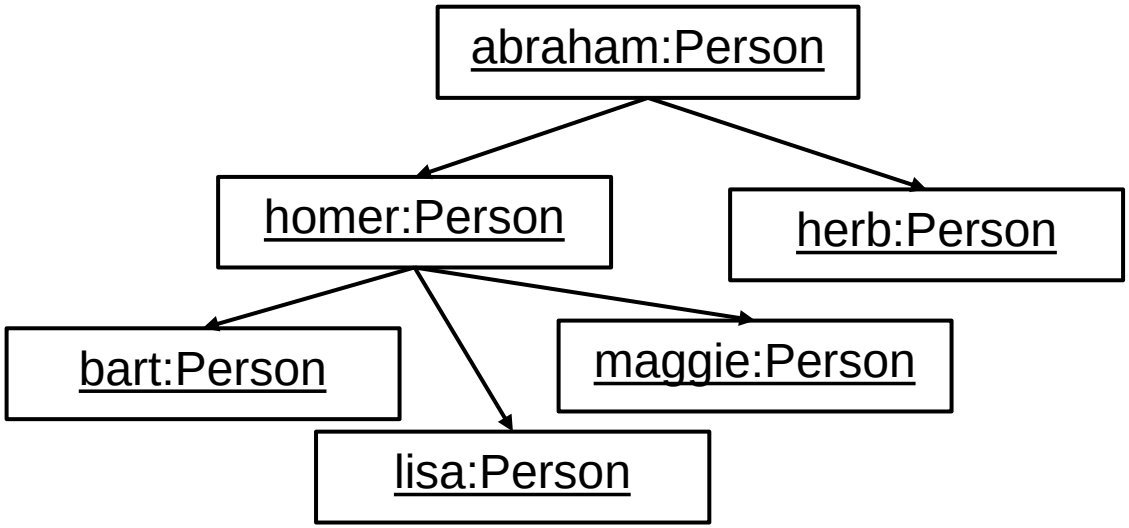
  - source->excludesAll(collection)
    - Returns true if collection source excludes all the objects in collection

Closure

- Iterates and increments a collection through a reference
- Calculates the transitive closure

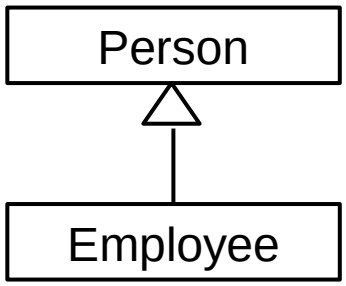
Set{abraham}->**closure**(children) ≡

Set{abraham, homer, herb, bart, lisa, maggie}



Operations for any type (OclAny)

- oclIsKindOf(Classifier typespec) : Boolean
  - Returns **true** if the type of *self* corresponds to the type or supertype of typespec, **false** otherwise.
  - This operation allows users to check the class hierarchy of *self* much like would an **instanceof** Java.



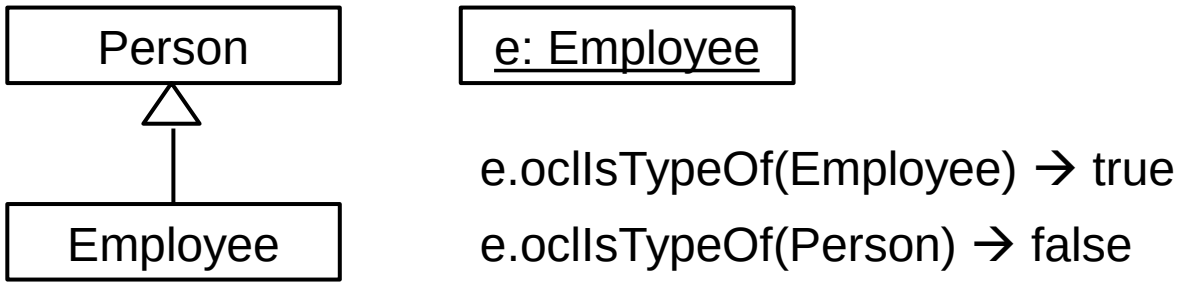
e: Employee

e.oclIsKindOf(Employee) → true

e.oclIsKindOf(Person) → true

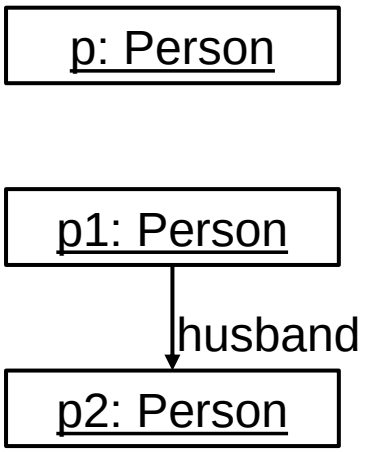
Operations for any type (OclAny)

- oclIsTypeOf(Classifier typespec) : Boolean
  - Returns **true** if the type of *self* is the same as typespec, or **false** otherwise.
  - This operation allows users to check the exact class type of *self*.



Operations for any type (OclAny)

- oclIsUndefined () : Boolean
  - Returns **true** if *self* is **null** (or invalid)



p.husband.oclIsUndefined() → true

p1.husband.oclIsUndefined() → false

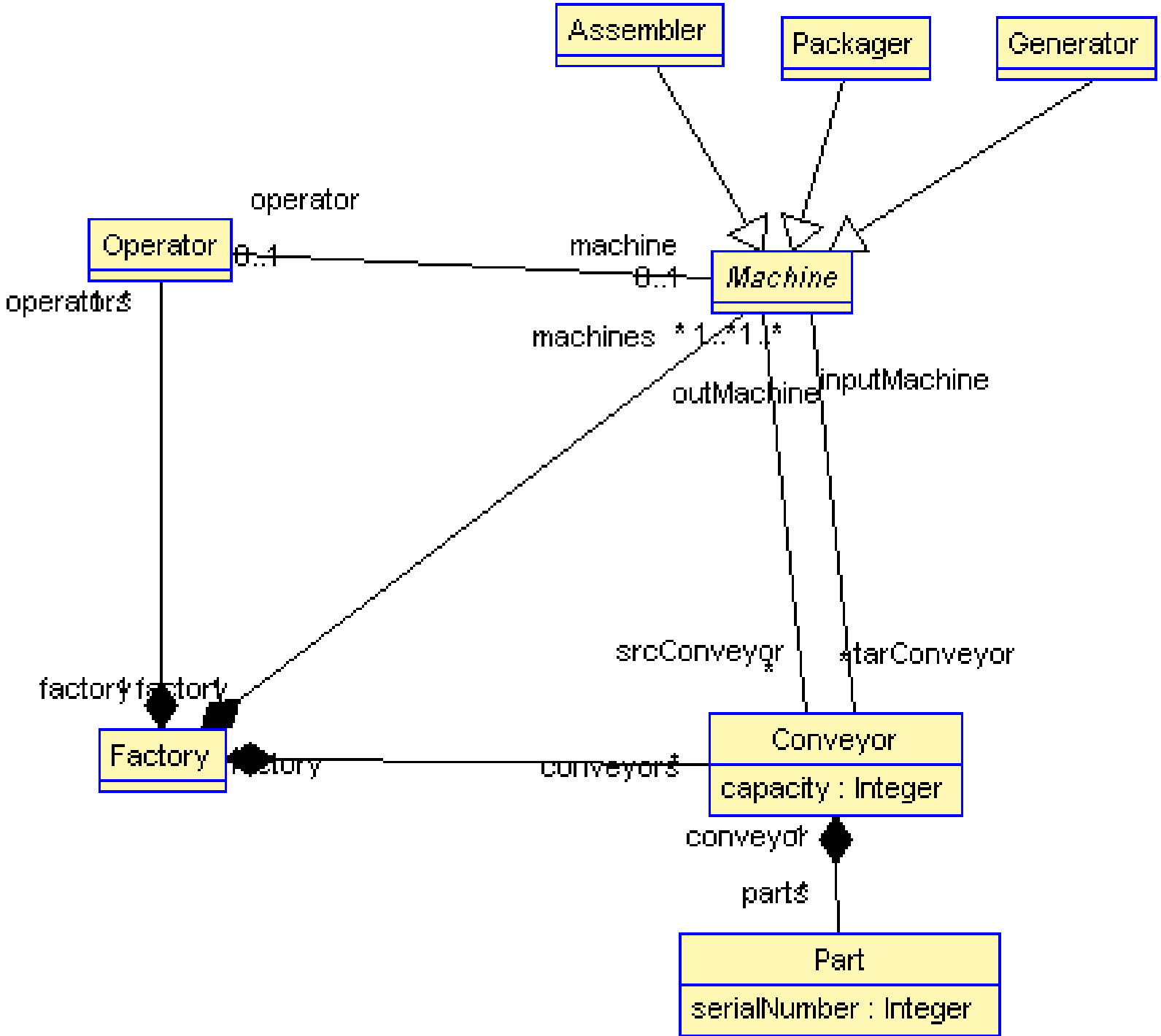
For more details on OCL operations see:
https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference

Exercise (in class)

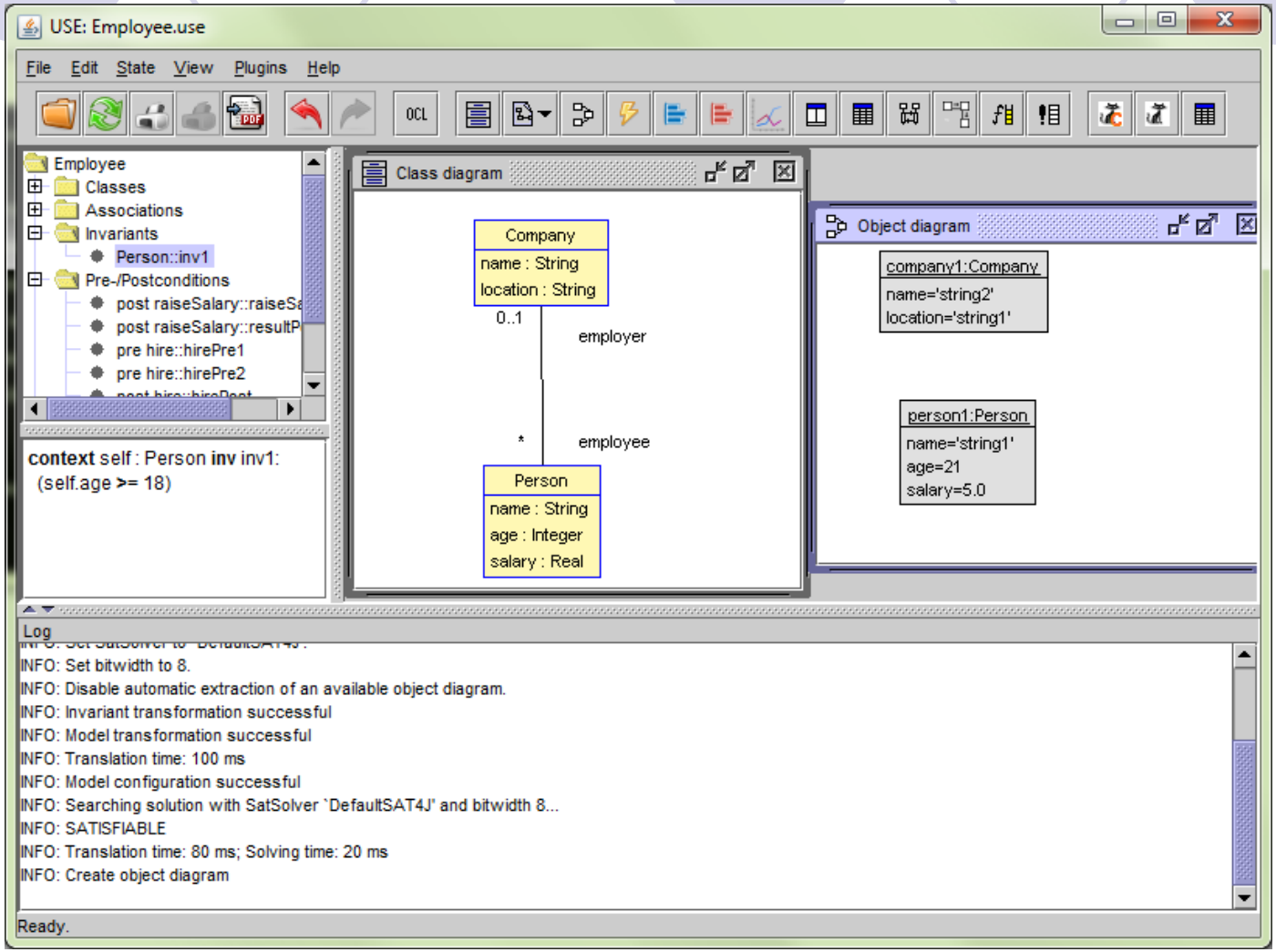
- A factory is made of machines of three different kinds: generators, assemblers and packagers.
- Machines are connected through conveyors, which transport parts from one machine to another.
- Conveyors can connect multiple (but at least one) incoming machines to multiple (but at least one) outgoing machines.
- Parts have a serial number.

- A conveyor can hold parts up to its maximum capacity.
- A generator cannot have incoming conveyors, and a packager does not have outgoing conveyors.
- Any machine can be operated by at most one operator (workers of the factory). Each operator operates at most one machine at the same time. At every moment, there should be at least one operator in any of the machines.

- Conveyors should have a positive maximum capacity


Assembler
Packager
Generator
operator
Operator
0..1
machine
0..1
Machine
operators
machines
* 1..*
1..*
outMachine
inputMachine
srcConveyor
tarConveyor
factory
factory
Factory
factory
conveyors
Conveyor
capacity : Integer
conveyor
parts
Part
serialNumber : Integer

Demo: USE/USE Validator



- Remember to download the USE validator plugin

Validating your constraints

- ***Model finders***: Class diagrams + OCL constraints as a constraint satisfaction problem
  - USE Validator, UMLtoCSP
  - Alloy (http://alloytools.org/, uses a different specification language)
- The problem is satisfiable if an object diagram exists satisfying all constraints
- If no solution exists:
  - The search bounds might be too narrow
  - There are conflicting constraints
- Small scope hypothesis: "*a high proportion of bugs can be found by testing the program for all test inputs within some small scope*" [JD96]

[JD96] D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Soft. Eng.*, 22(7), 1996.

Exercise [1/2] (home work)

A simple reviewing system for scientific conferences

- The system can handle several conferences at a time, and accept the registration of several users.
- Each conference has one or two *chairs*.
- Authors submit papers to a given conference. One of the authors is the “corresponding” author, which receives notifications about the paper.
- Each paper is reviewed by 3 reviewers.
- On the basis of the review, the chairs decide accepting or rejecting the papers. For this purpose, they send a notification to the corresponding authors of each paper.
- Papers are described by a title, authors, abstract and body.

Exercise [2/2]

A simple reviewing system for scientific conferences

- Authors, reviewers and chairs are described by a name, affiliation and e-mail, and all should be registered users. Also, we need to distinguish whether they are students.
- Reviews are made of a score (from 0 to 5) and comments.
- Reviewers can submit articles, but are not allowed to review their own papers. Conference chairs cannot submit papers to the conference they chair.
- Students cannot be reviewers or chair any conference.

Make a class diagram, and include OCL constraints if needed.
Hand in: October 5th (by e-mail to Juan.deLara@uam.es).
In USE format. USE is a UML tool available at:
http://sourceforge.net/projects/useocl/
Remember to validate all your constraints!!

Index

- Introduction.
- Class and Object Diagrams.
- Other Diagrams.
- OCL: The Object Constraint Language.
- **Bibliography.**

Bibliography: UML and OCL

- **UML**
  - OMG web site on UML: http://www.uml.org
  - UML 2.0 in a Nutshellby Dan Pilone; Neil Pitman. O'Reilly. June 2005
  - Perdita Stevens, Rob Pooley. "*Utilización de UML en Ingeniería del Software con Objetos y Componentes*". Addison Wesley, 2002.
  - Grady Booch, James Rumbaugh, Ivar Jacobson. "*The Unified Modeling Language User Guide*". Addison-Wesley, 1999.
  - UML 2 Toolkit. Eriksson, H. E., Penker, M., Lyons, B., Fado, D. OMG Press, Wiley. 2004.
  - Craig Larman. "*Applying UML and Patterns*". Prentice Hall. 2002.

- **OCL**
  - The Object Constraint Language 2nd Edition. Getting your Models Ready for MDA. Warmer, Kleppe. Addison-Wesley. 2003.
  - OCL specification: https://www.omg.org/spec/OCL

Bibliography: Statecharts

- Harel D. *"On Visual Formalisms".* Communications of the ACM. Vol 31, No. 5. Pp.: 514-530. Mayo 1988.

- Harel D., Naamad A. *"The STATEMATE Semantics of Statecharts"*. ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, Oct. 1996, pp.: 293-233.

- David Harel and Eran Gery. Executable object modeling with statecharts. IEEE Computer, pages 31-42, 1997.