AOT LAB

Agent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma

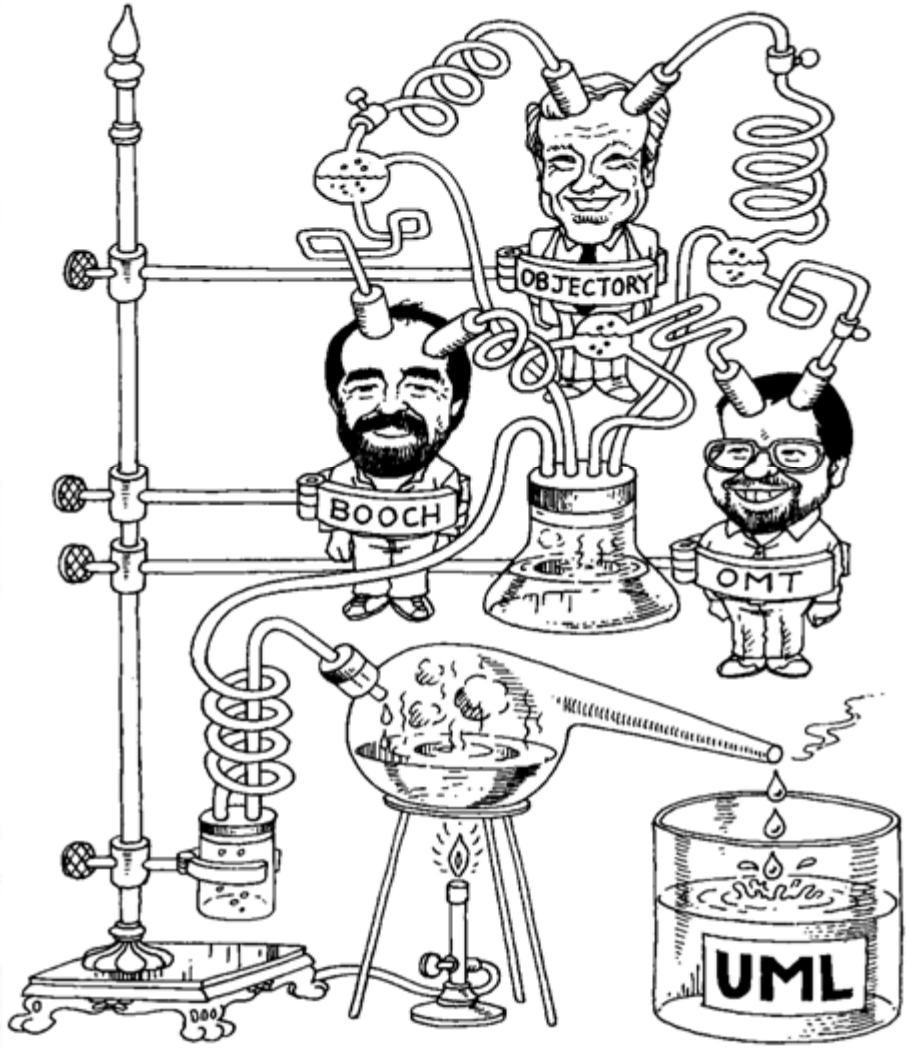
Advanced Software Engineering

Unified Modeling Language

Prof. Agostino Poggi

AOT LAB

What is UML?



The Unified Modeling Language (UML) is the standard language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system

What is UML?

- Is a language and notation system used to specify, construct, visualize and document models of software systems

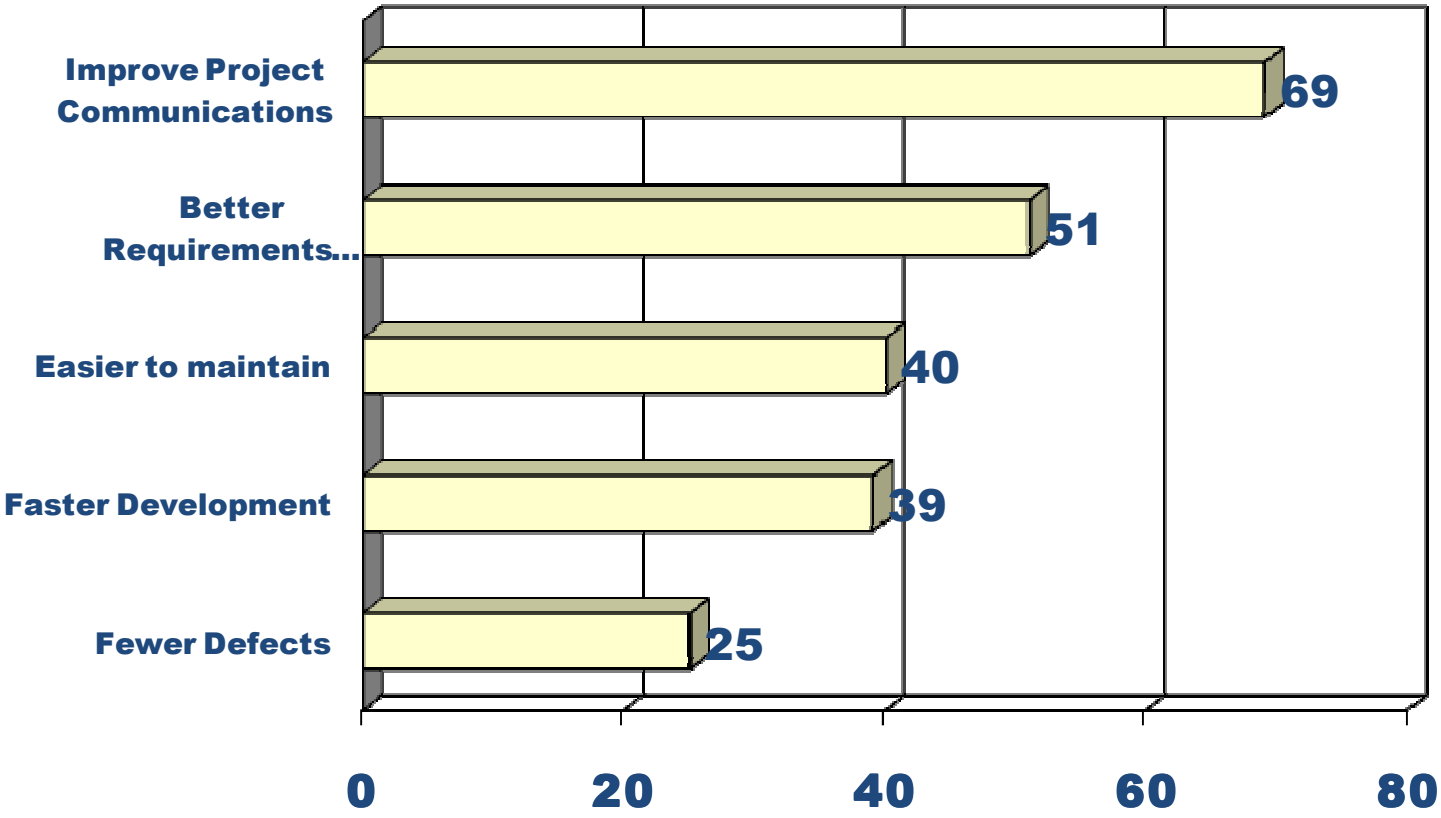
- Is not a methodology (which considers the specific framework and conditions of an application domain, the organizational environment and many other things)

Why Use UML?

- Provides multiple diagrams for capturing different architectural views
- Is a standard language for visualizing, specifying, constructing, and documenting software systems
- Tool support and interoperability improves in time, as UML, OCL, and XMI are still relatively young standards

AOT LAB

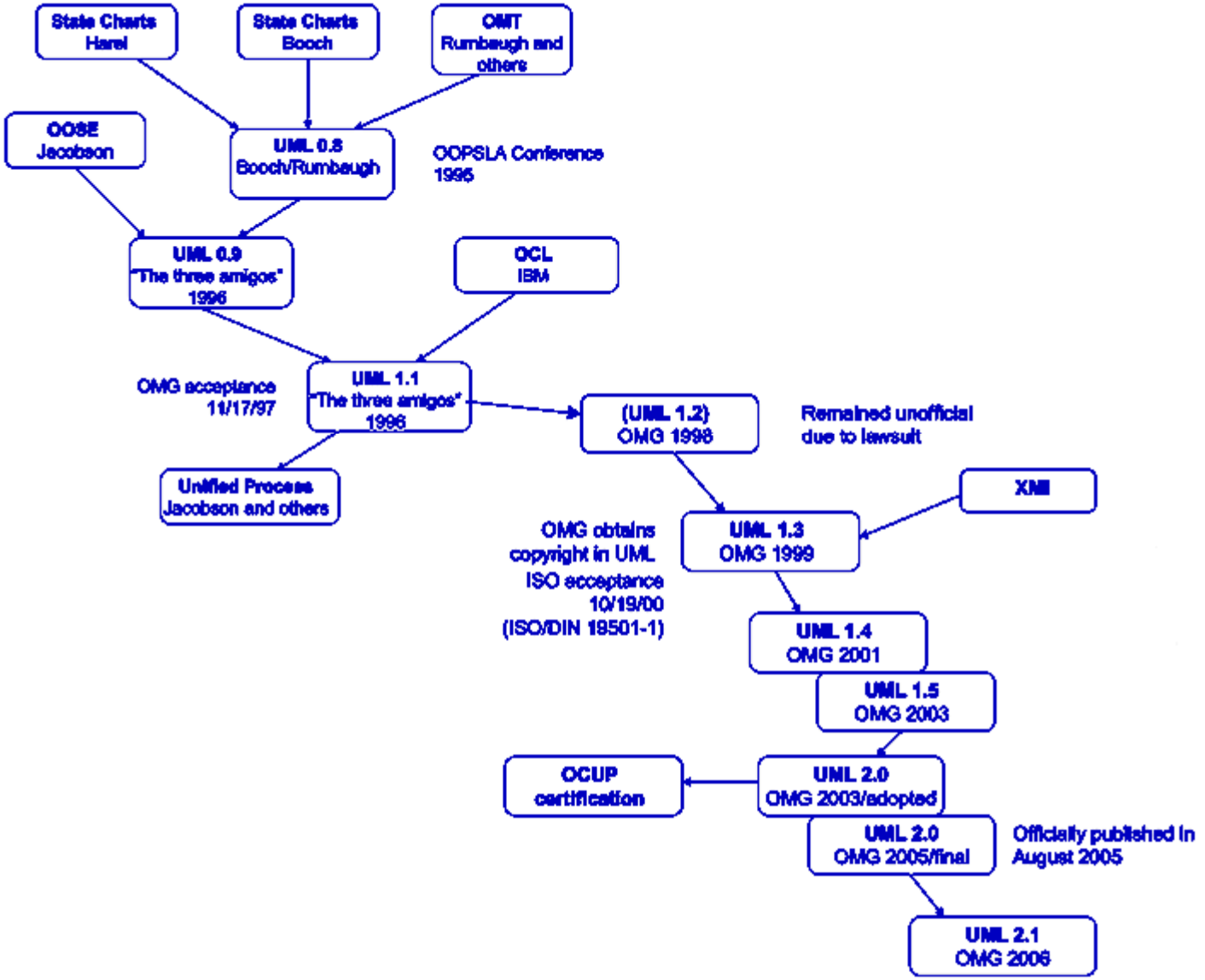
Why Use UML?



Source: BZ Research, August 2004

AOT LAB

UML History



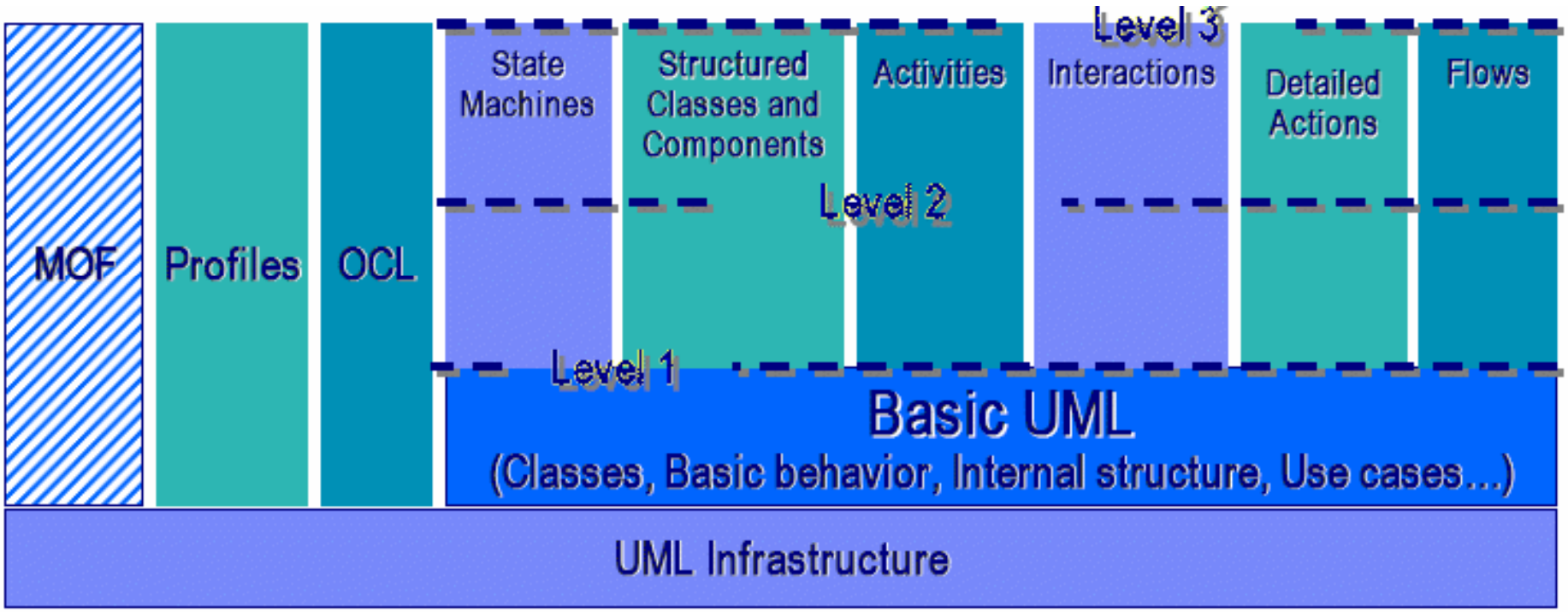
UML Notation Goals

- Simple because it requires only a few concepts and symbols
- Expressive because it is applicable to a wide spectrum of systems and life cycle methods
- Useful because it focuses only upon those necessary elements to software engineering
- Consistent because the same concept and symbol should be applied in the same fashion throughout
- Extensible because users and tool builders should have some freedom to extend the notation

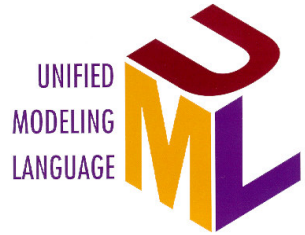
UML Specifications

- **Infrastructure**, the foundational language constructs
- **Superstructure**, the user level constructs
- **Object Constraint Language** (**OCL**), the formal language used to describe expressions on UML models
- **Diagram interchange,** the means enabling a smooth and seamless exchange of documents compliant to the UML standard

UML Architecture



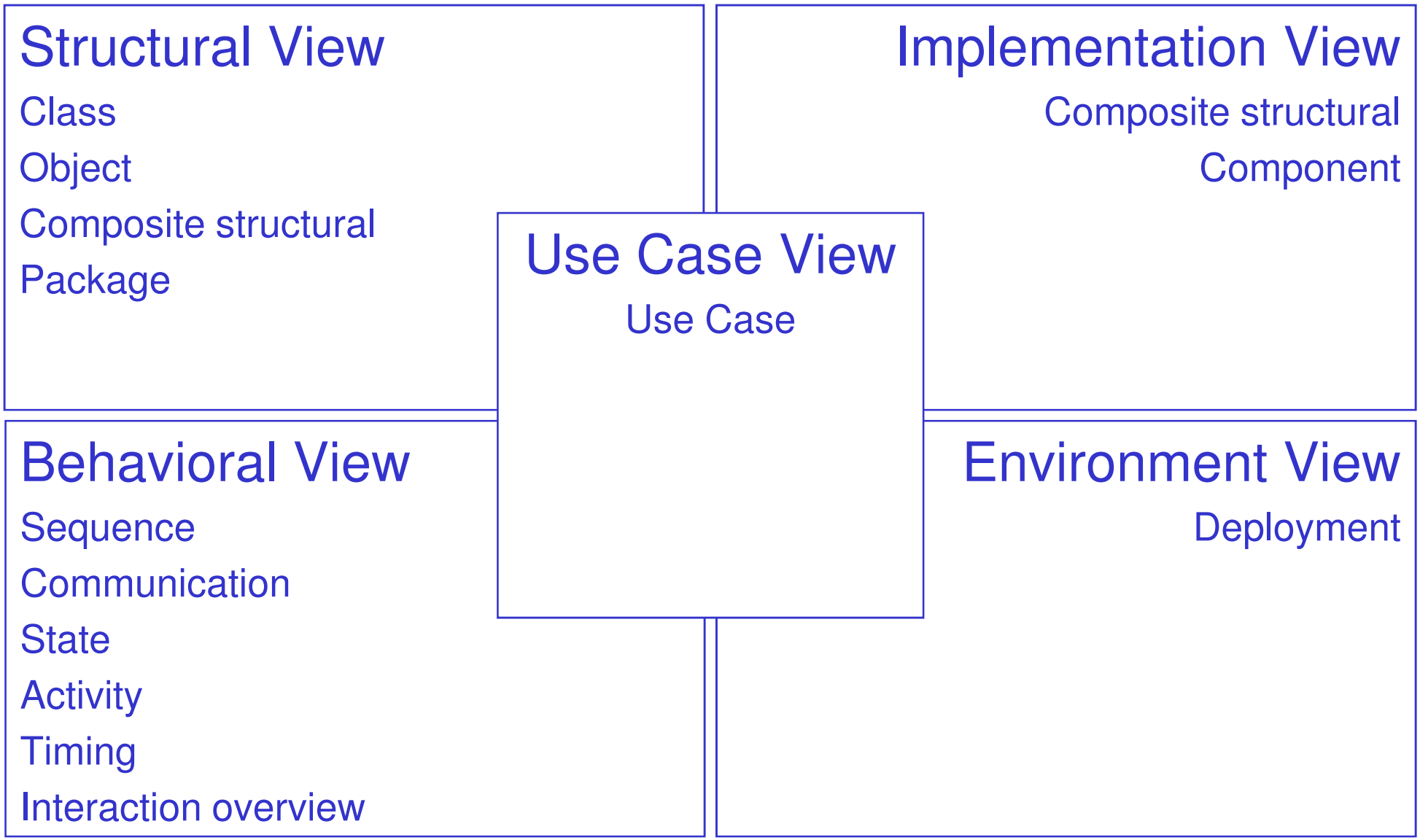
AOT LAB



UML Diagrams

AOT LAB

UML Diagram Types



Use Case View

- The most important architectural view
- Describes use cases that provide value for the users
- Essential use cases are used as proof of concept for implementation architecture
- Use cases may be visualized in UML **use case** diagrams
- Each use case may have multiple possible scenarios
- Use case scenarios could be described:
  - Using textual descriptions
  - Graphically, using UML **activity** diagrams

Structural View

- Represents structural elements for implementing solution for defined requirements and defines:
  - Object-oriented analysis and design elements
  - Domain and solution vocabulary
  - System decomposition into layers and subsystems
  - Interfaces of the system and its components
- Is represented by static UML diagrams:
  - **Class** diagrams in multiple abstraction levels
  - **Object** diagrams
  - **Composite structural** diagrams
  - **Package** diagrams

Behavioral View

- Represents dynamic interaction between system components for implementing requirements
- Shows distribution of responsibilities and allows to identify interaction and coupling bottlenecks
- A means for discussing non-functional requirements: performance, maintenance, …
- Is represented by dynamic UML diagrams:
  - **Sequence** diagrams
  - **Communication** diagrams
  - **Activity** diagrams
  - **State** diagrams
  - **Interaction overview** diagrams
  - **Timing** diagrams

Implementation View

- Describes implementation artifacts of logical subsystems defined in structural view
- May include intermediate artifacts used in system construction (code files, libraries, data files, …)
- Defines dependencies between implementation components and their connections by required and provided interfaces
- Is represented by these UML diagrams:
  - **Component** diagrams
  - **Composite structural** diagrams

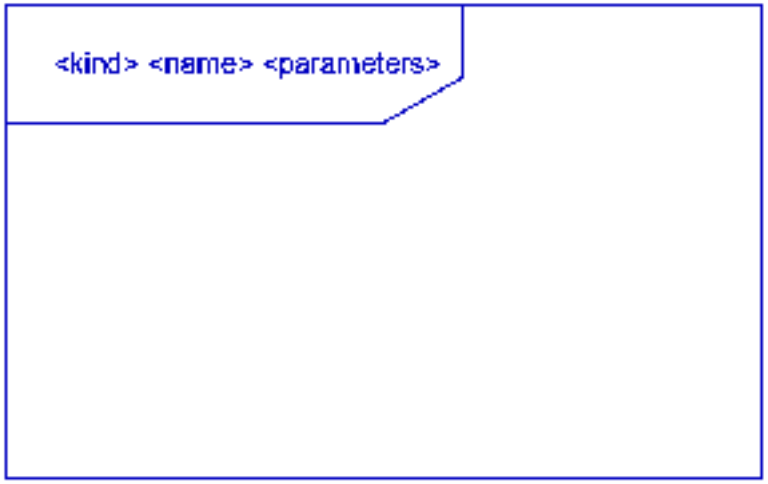
Environment View

- Represents system hardware topology
- Defines how software components are deployed on hardware nodes
- Useful for analyzing non-functional requirements: reliability, scalability, security, …
- Provides information for system installation and configuration
- Is represented by the UML **deployment** diagram

AOT LAB

Diagram Frame

The heading is a string contained in a name tag which is a rectangle with cut off corners in the upper left hand corner of the frame

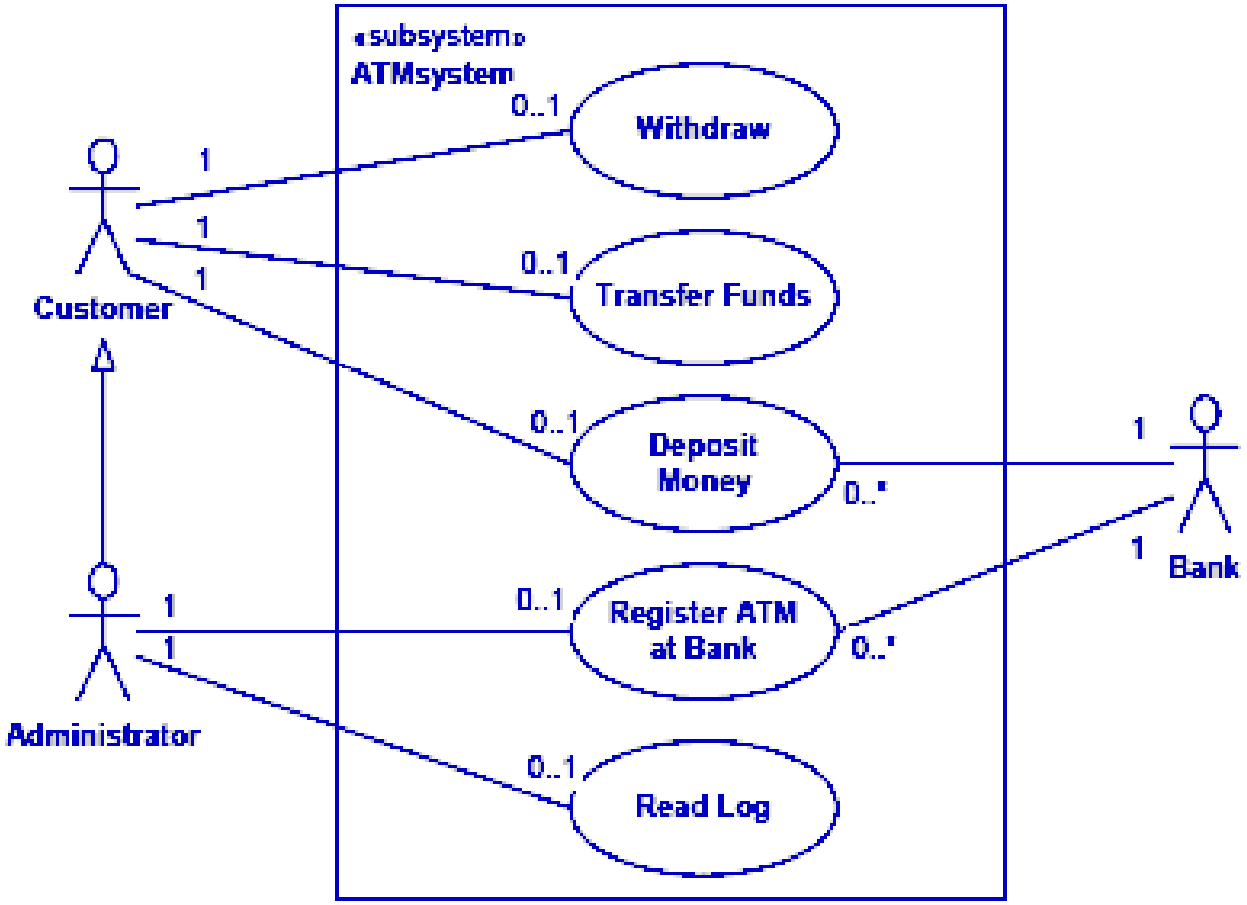


Each diagram has a frame, a content area and a heading

The frame is a rectangle and is used to denote a border

AOT LAB

Use Case Diagram



Use Case Diagram

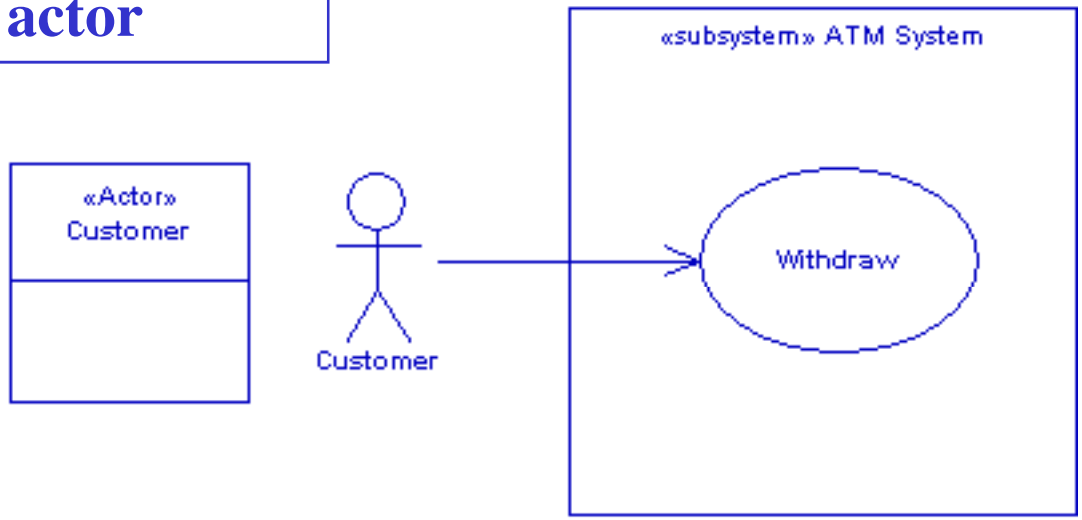
- A use case describes the proposed functionality of a system
- A use case represents a discrete unit of interaction between a user (human or machine) and the system
- This interaction is a single unit of meaningful work, that may be include a complex interaction between parts

AOT LAB

Actor, System Boundary and Use Case

A user of the system is identified with the name of **actor**

The **system boundary** usual divides what is inside or outside the system (use cases from actors)

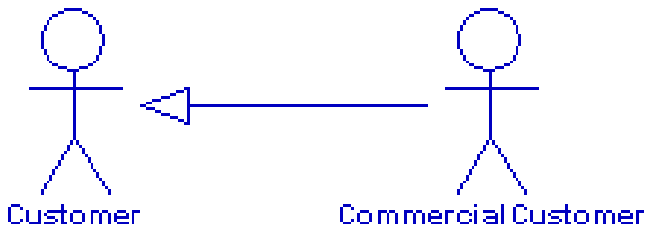


An actor can be also represented by a class rectangle with the «actor» keyword

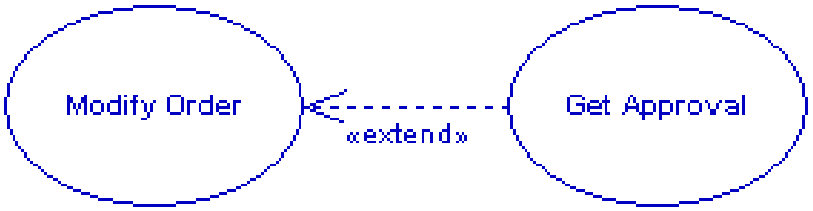
A single unit of meaningful work related to a functionality of the system is identified with the name of **use case**

AOT LAB

Generalization and Composition

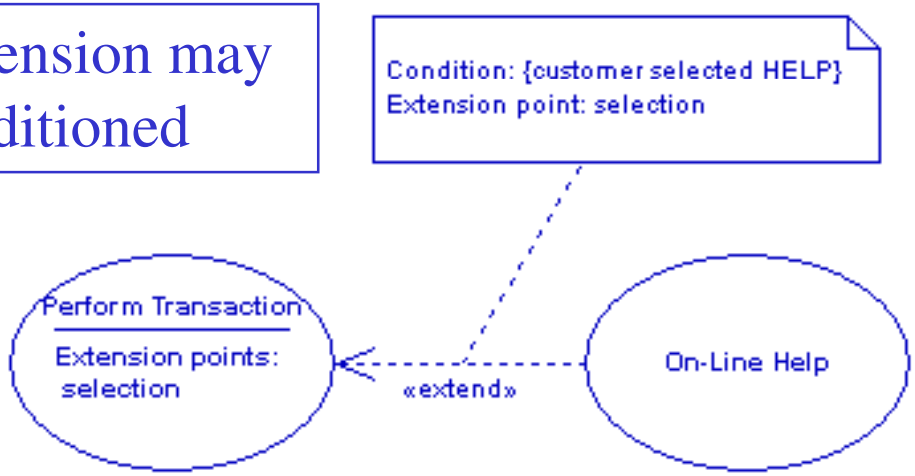


An actor can generalize another actor

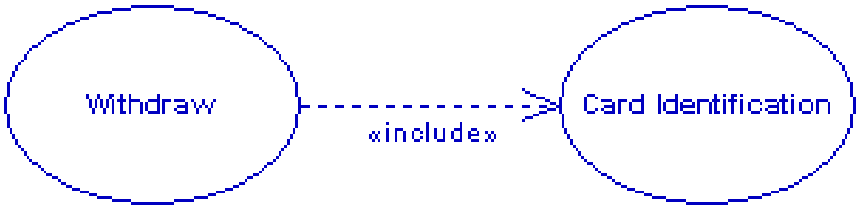


A use case may be used to extend the behavior of another use case

An extension may be conditioned



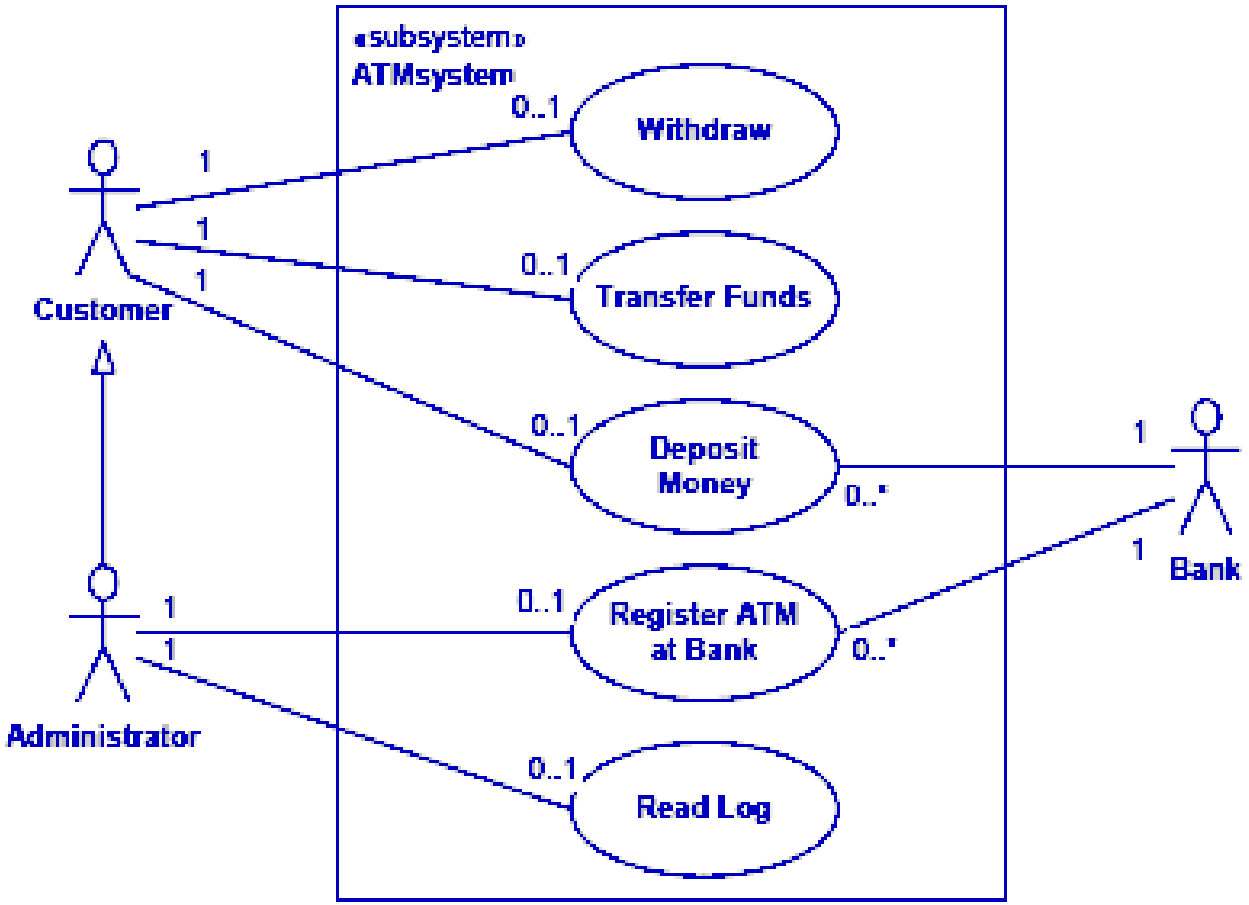
A use case may contain the functionality of another use case as part of their normal processing



An extension point indicates where an extending use case is added

Multiplicity

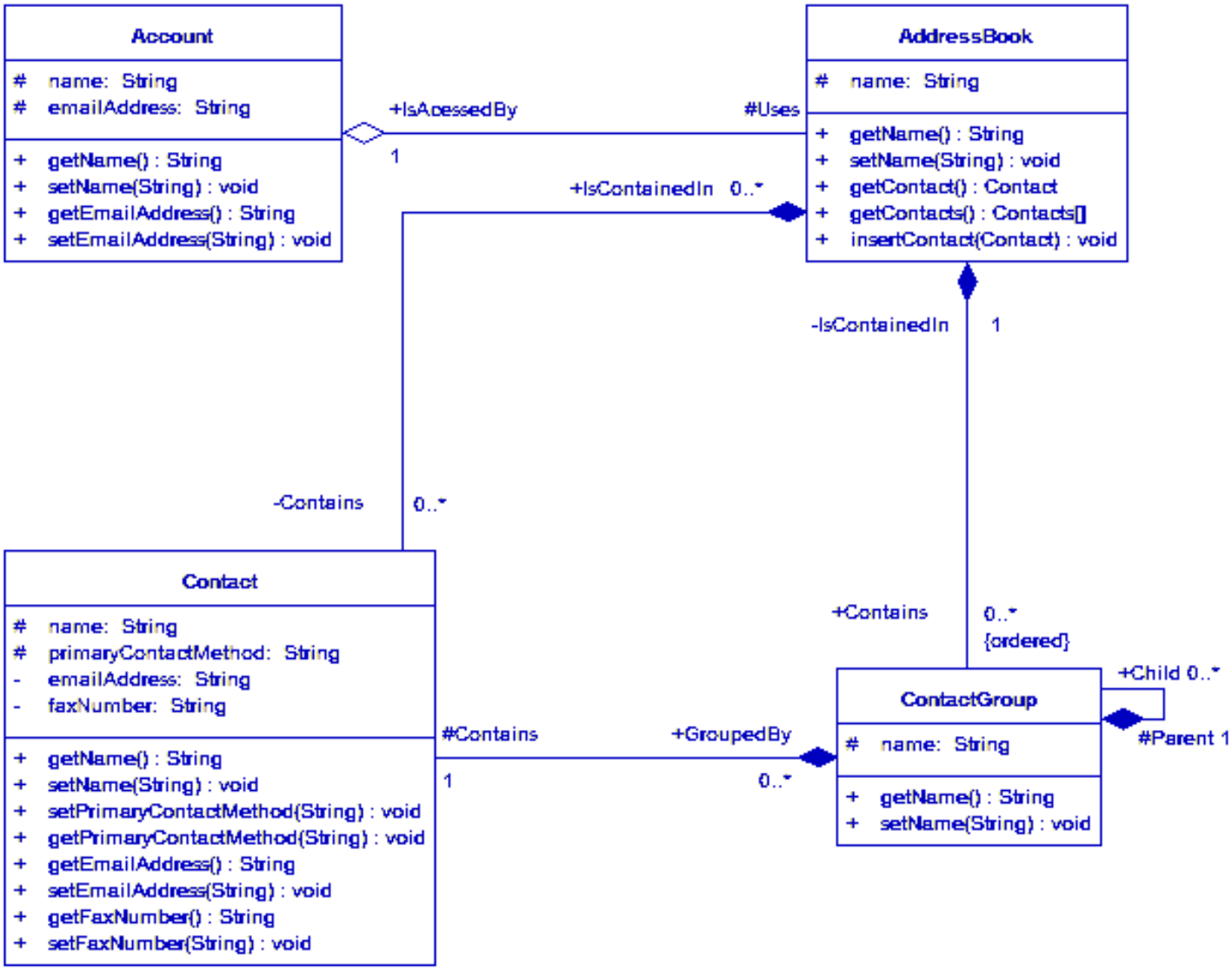
A use case diagram can contain several use cases and actors



The *uses* connector can optionally have multiplicity values at each end

AOT LAB

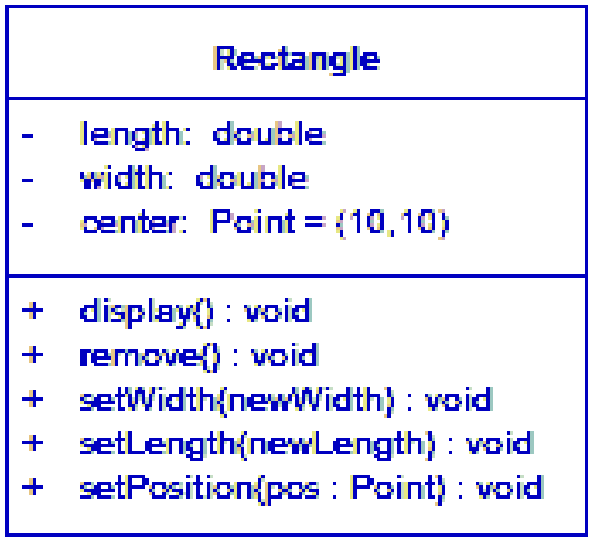
Class Diagram



Class Diagram

- A class diagram shows the building blocks of an object-orientated system
- Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations
- Class diagrams are most useful in illustrating relationships between classes and interfaces
- Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively

Class

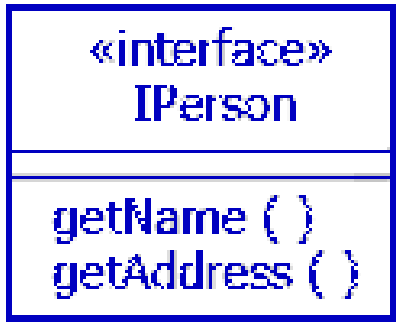


A **class** is represented by a rectangle which shows the name of the class and optionally the name of its operations and attributes. Compartments are used to divide the class name, attributes and operations

The symbol that precedes the attribute, or operation name, indicates the visibility of the element

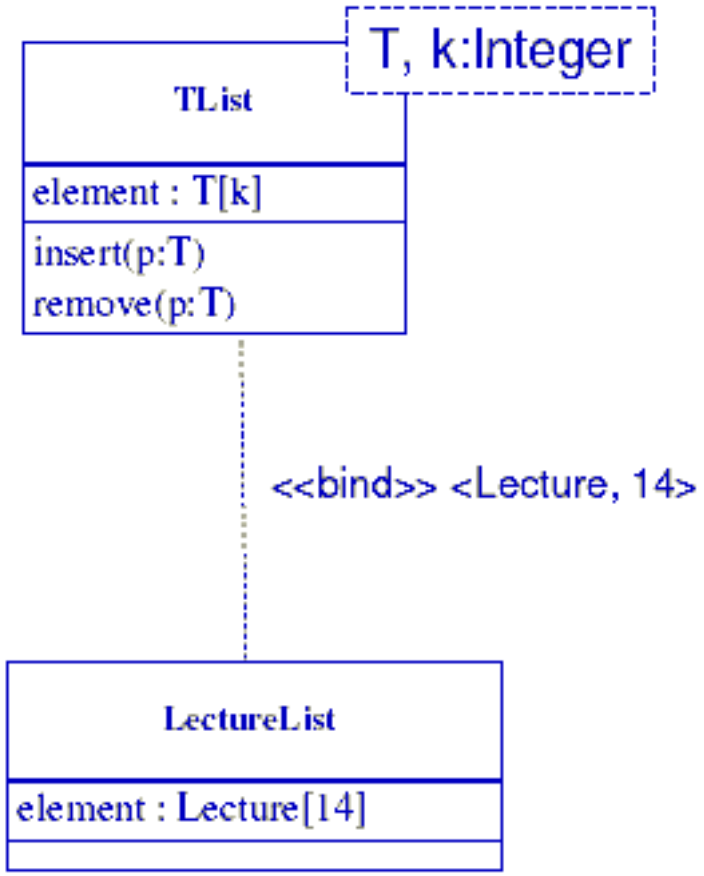
+ is public visibility
– is private visibility
is protected visibility
~ is package visibility

Interface and Template

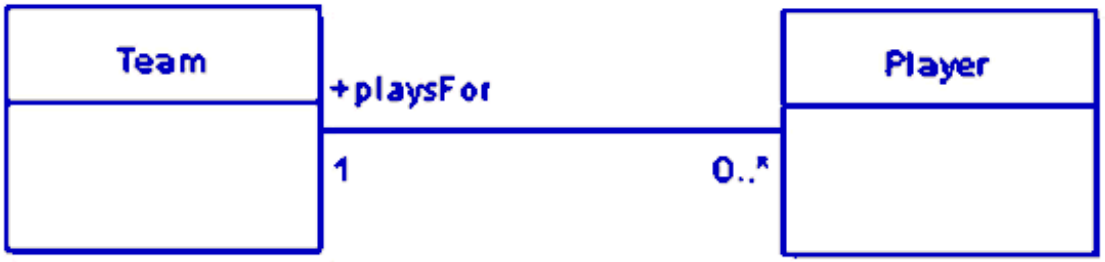


An **interface** is a specification of behavior that implementers agree to meet, that is, a contract

A **template** defines a pattern whose parameters represent types and can be applied to classes, packages, operations



Association



An **association** implies two model elements have a relationship that usually is implemented as an instance variable in one class

An association is represented by a connector that may include named roles at each end, cardinality, direction and constraints

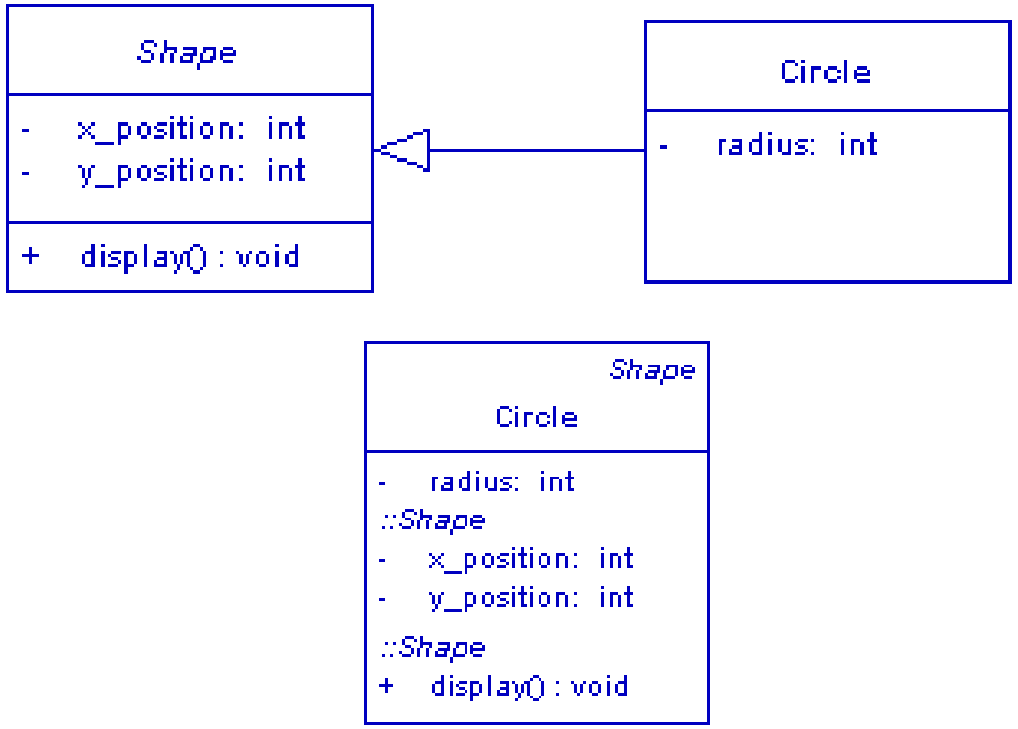
For more than two elements, a diamond representation toolbox element can be used as well

AOT LAB

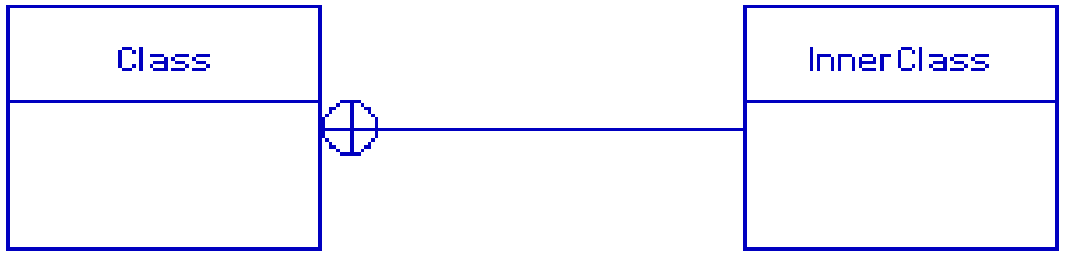
Generalization and Nesting

A **generalization** is used to indicate inheritance.

Drawn from the specific class to a general class, the generalize implication is that the source inherits the target's characteristics

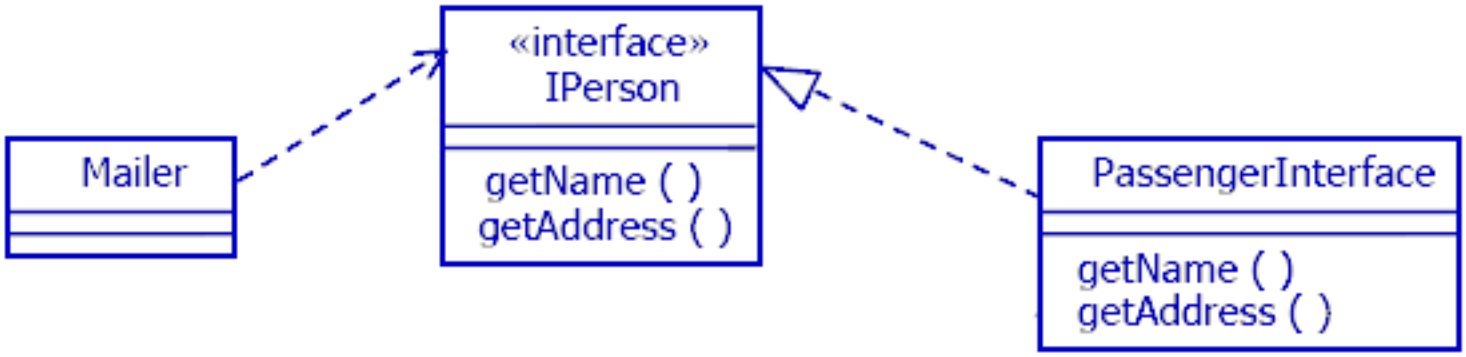


A **nesting** is connector that shows the source element is nested within the target element



AOT LAB

Dependency and Realization

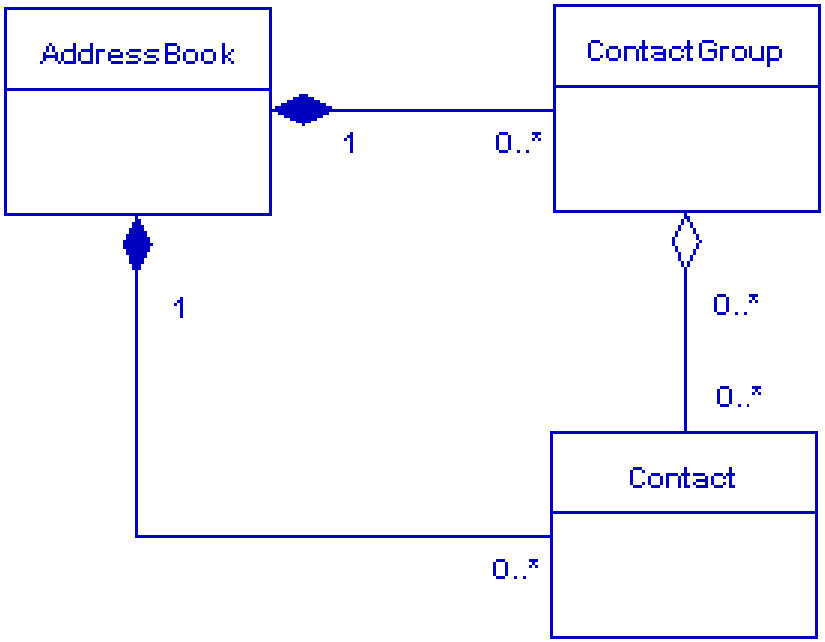


A **dependency** is a weaker form of relationship showing a relationship between a client and a supplier

A **realization** is a relationship between a specification and its implementation

AOT LAB

Aggregation and Composition



An **aggregation** is used to depict elements which are made up of smaller components

Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class

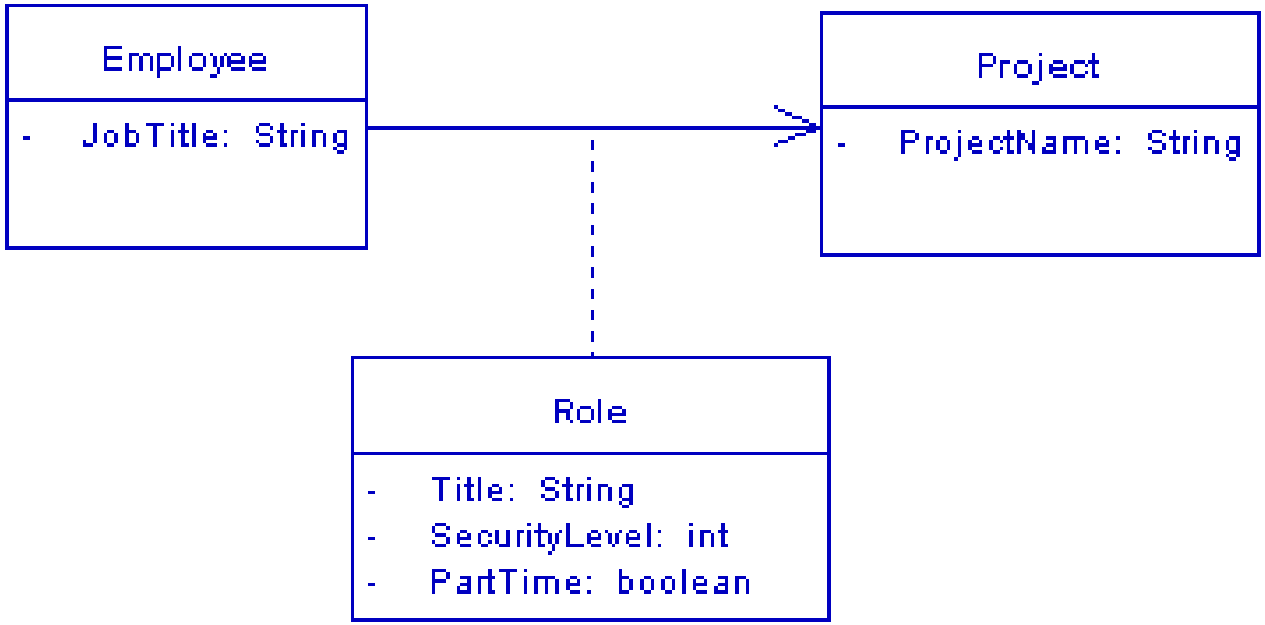
A **composition** is a stronger form of aggregation that is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time

If the container is deleted, usually all of its parts are deleted with it, but a part can be individually removed without having to delete the container

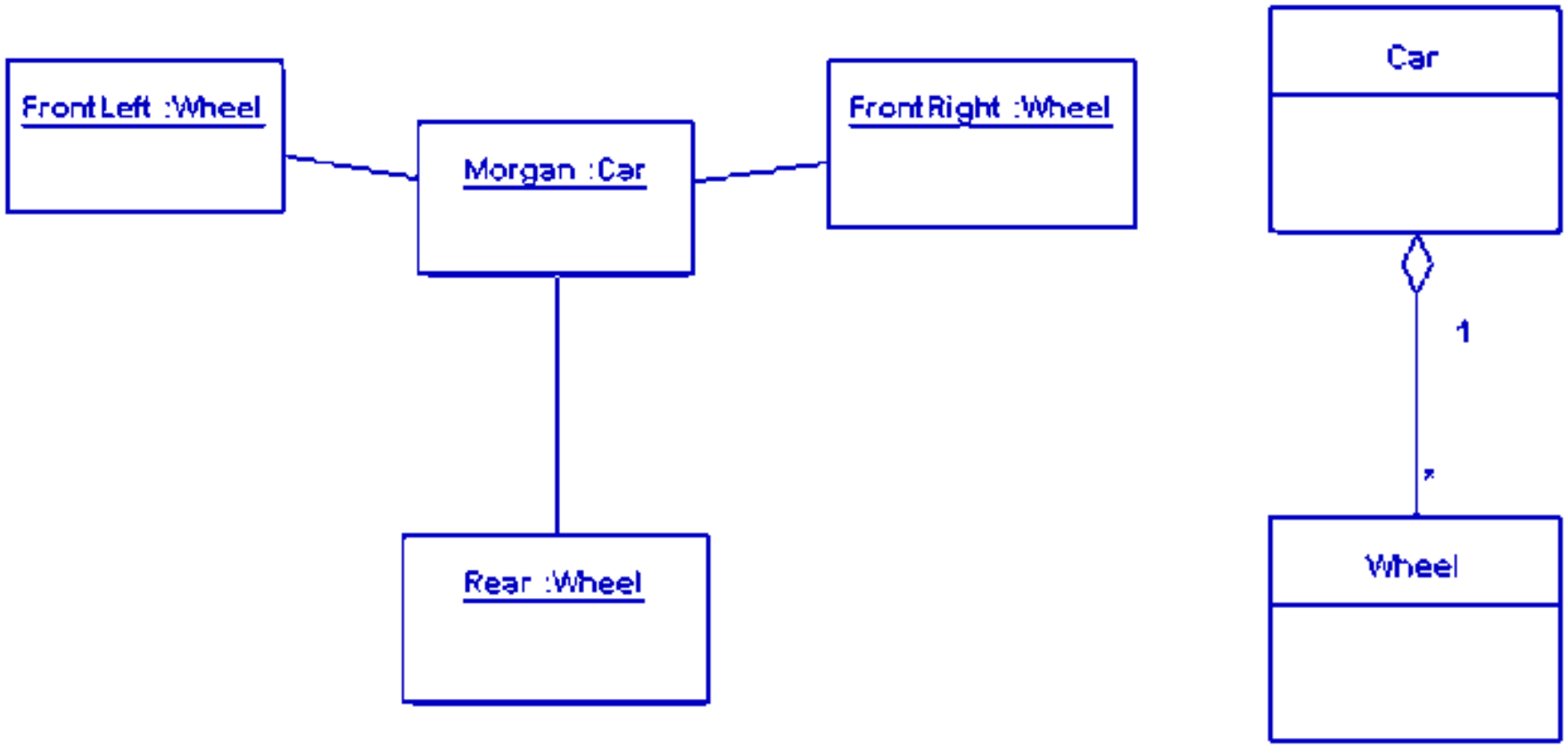
Compositions are transitive, asymmetric relationships and can be recursive

Association Class

An **association class** is a construct that allows an association connection to be defined with a set of operations and attributes



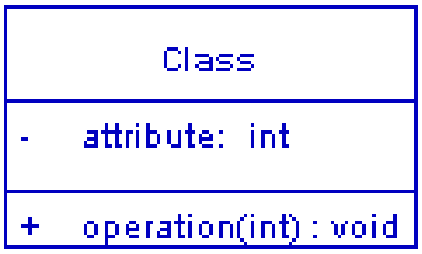
Object Diagram

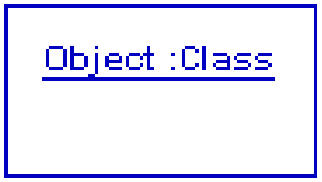


Object Diagram

- An object diagram describes the static structure of a system at a particular time and may be considered a special case of a class diagram
- Whereas a class model describes all possible situations, an object model describes a particular situation
- Object diagrams are useful in understanding and validating the corresponding class diagrams

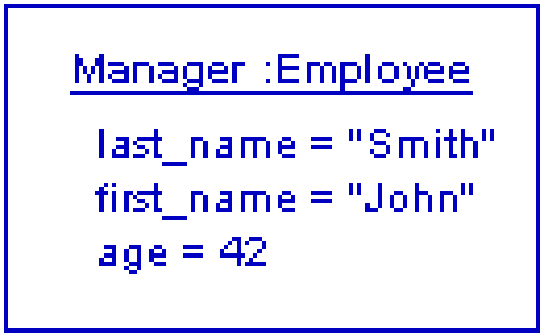
Object





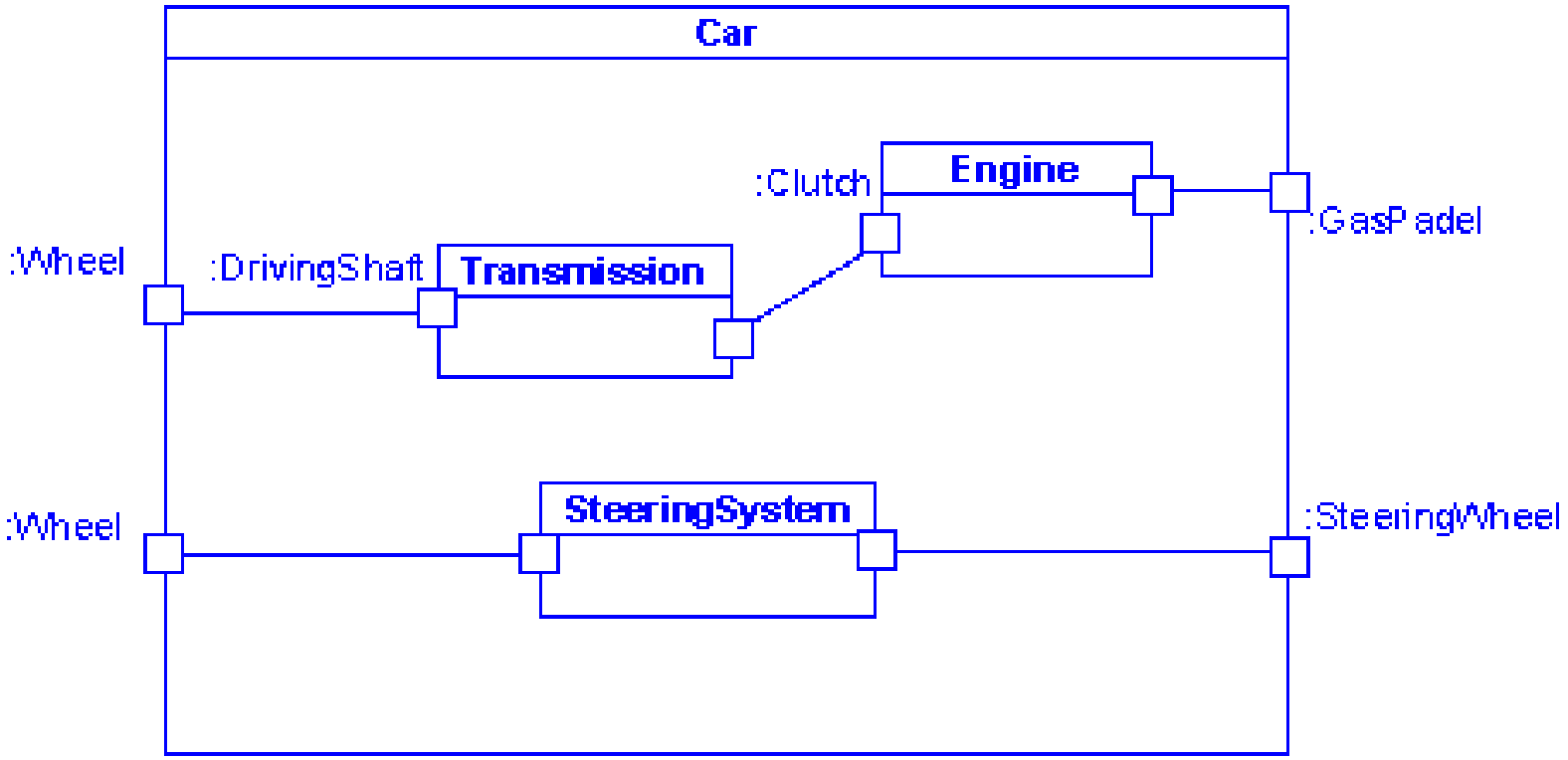
By default, object elements do not have compartments and their names are underlined and may show the name of the class from which the object is instantiated

Sometimes it is possible to represent an object's run time state, showing the set values of attributes in the particular instance



AOT LAB

Composite Structure Diagram

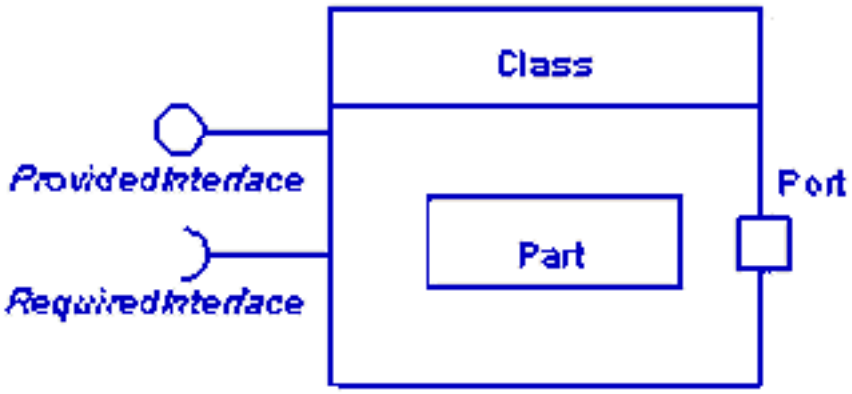


Composite Structure Diagram

- A composite structural diagrams shows the internal structure of a classifier, including its interaction points to other parts of the system
- A **classifier** is an UML element that is described by attributes and/or methods (i.e., a class, an interface or a component)
- A composite structure diagram is similar to a class diagram, but it depicts individual parts instead of whole classes

Structured Classifier

A **structured classifier** represents a class, often an abstract class, or a component whose behavior can be completely or partially described through interactions between parts

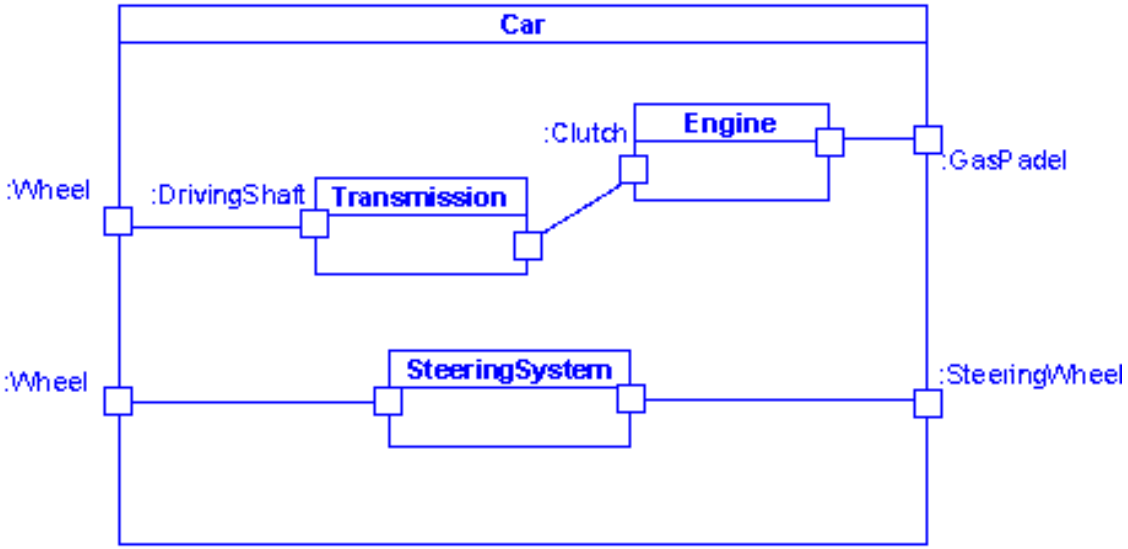


A **part** represents a role played at runtime by one instance of a class or by a collection of instances

An **encapsulated classifier** is a type of structured classifier that contains ports

A **port** is an interaction point that can be used to connect structured classifiers with their parts and with the environment

Port



Ports can optionally specify the services they provide and the services they require from other parts of the system

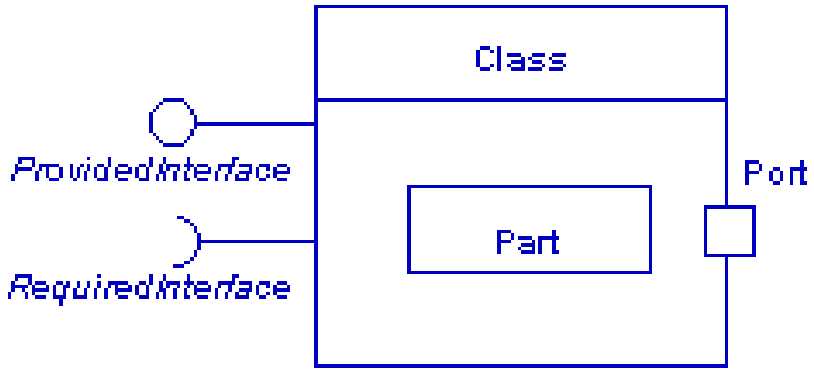
Public ports that are visible in the environment are shown straddling the boundary, while **protected ports** that are not visible in the environment are shown inside the boundary

Ports can either delegate received requests to internal parts, or they can deliver these directly to the behavior of the structured classifier that the port is contained within

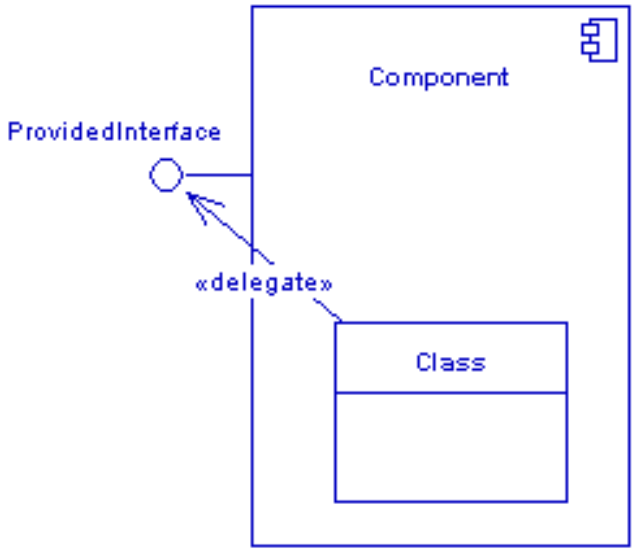
Exposed Interfaces

A **provided interface** is shown as a "ball on a stick" attached to the edge of a classifier element

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

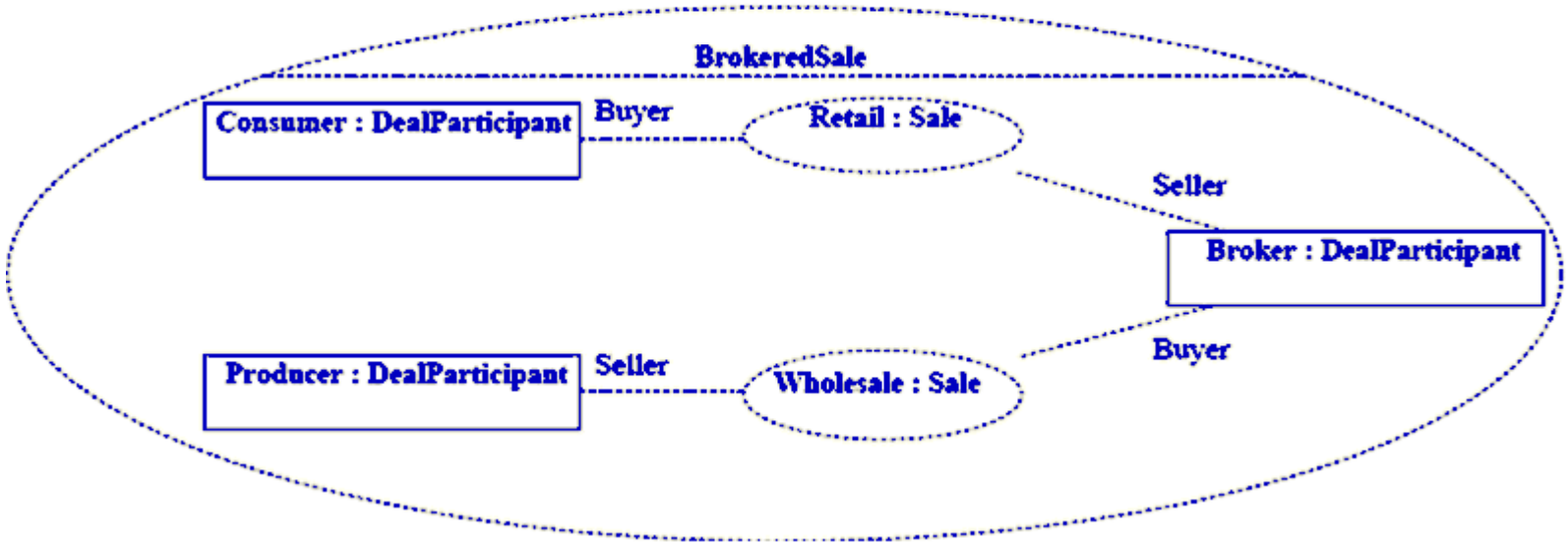


A delegate connector is used for defining the internal workings of a component's external ports and interfaces



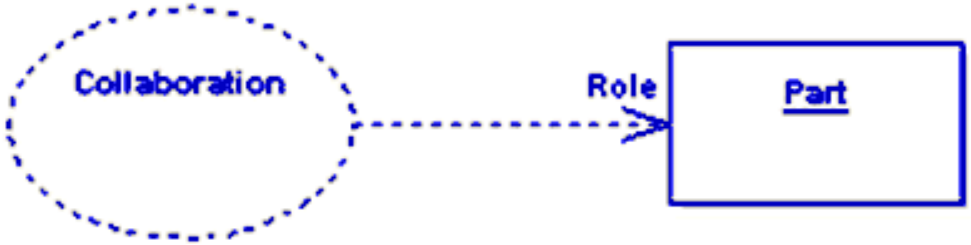
Collaboration

A **collaboration** defines a set of cooperating roles used collectively to illustrate a specific functionality

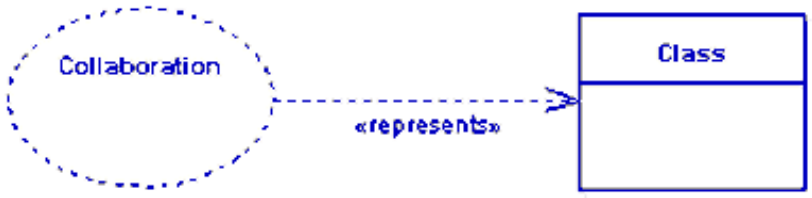


Collaboration

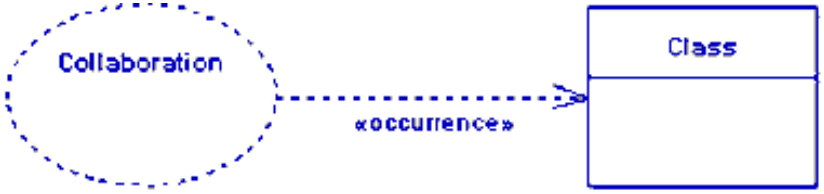
A **role binding** connector is drawn from a collaboration to the classifier that fulfils the role



A **represents connector** may be drawn from a collaboration to a classifier to show that a collaboration represents the classifier

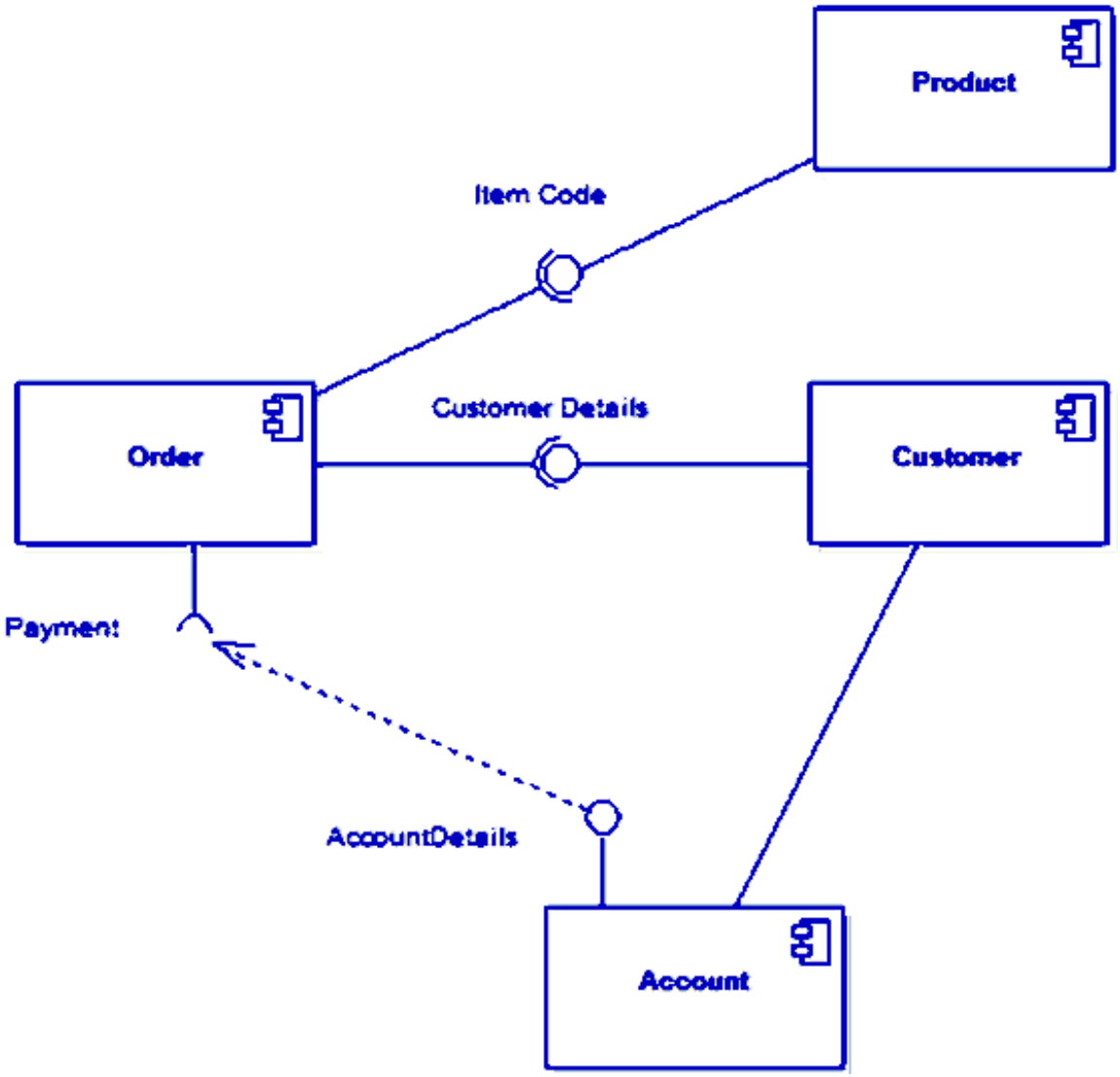


An **occurrence connector** may be drawn from a collaboration to a classifier to show that a collaboration is used in the classifier



AOT LAB

Component Diagram

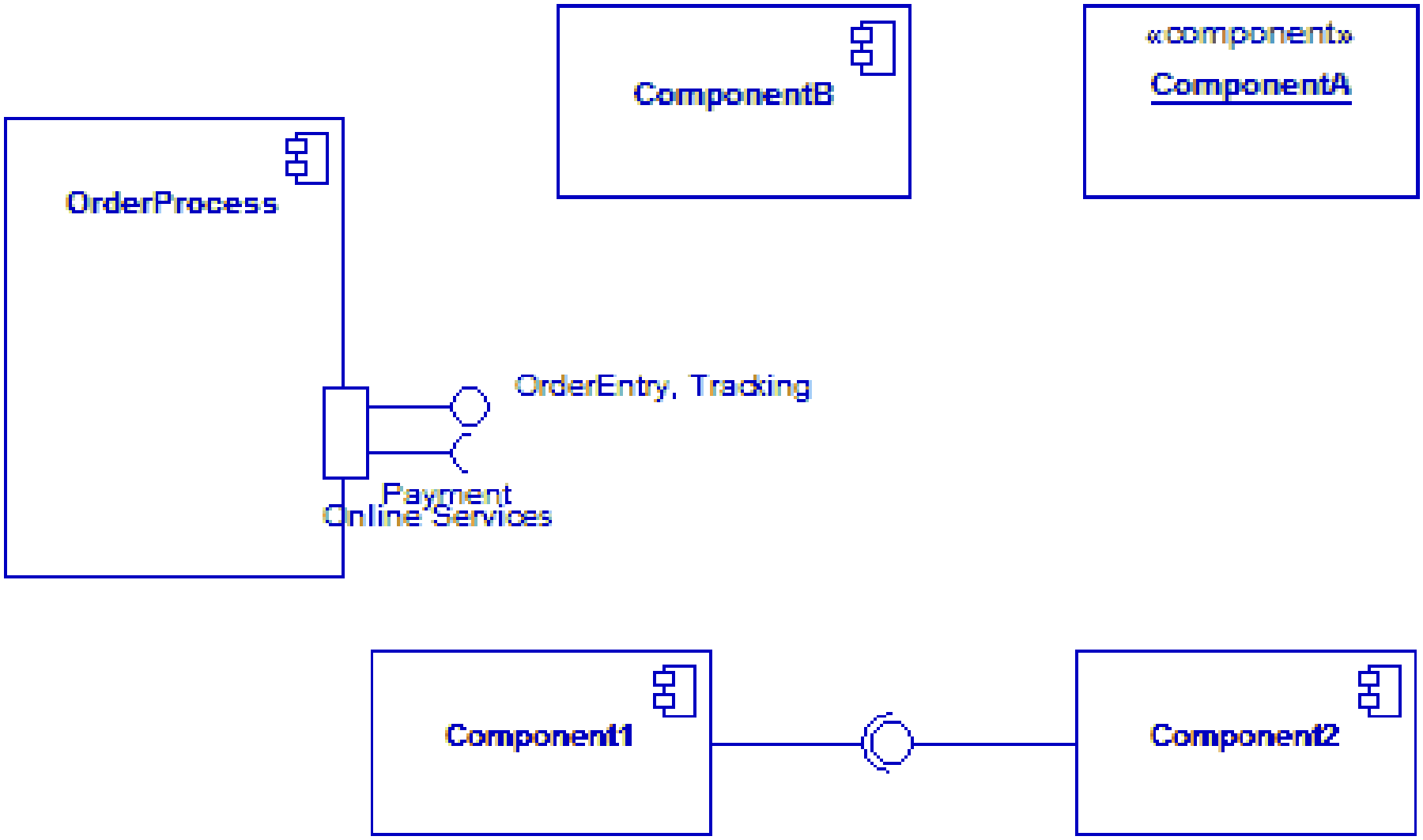


Component Diagram

- Component diagrams illustrate the pieces of software, embedded controllers, etc., that will make up a system
- A component diagram has a higher level of abstraction than a class diagram
  - Usually a component is implemented by one or more classes (or objects) at runtimes
- Components represent distributable physical units, including source code, object code, and executable code

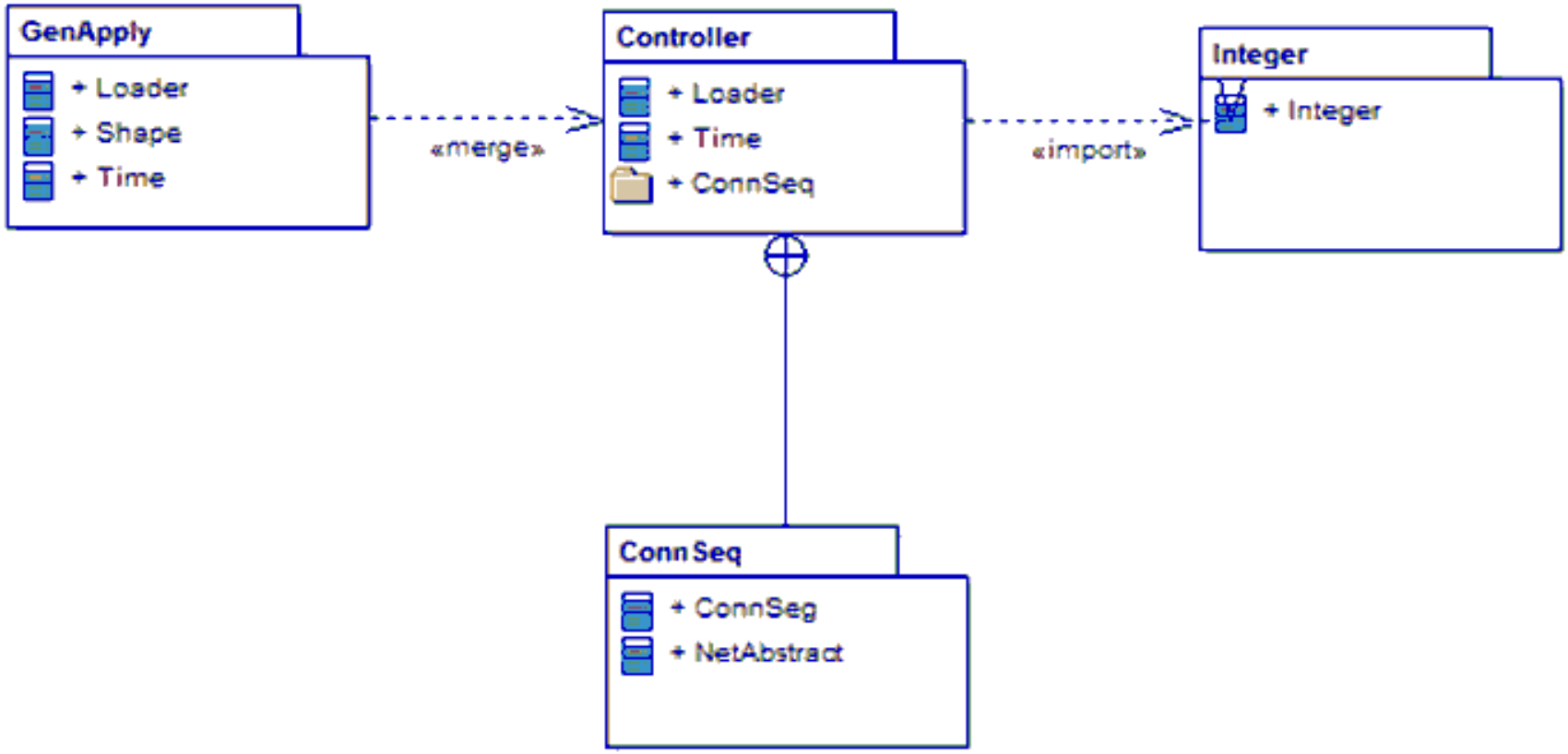
AOT LAB

Component Diagram



AOT LAB

Package Diagram



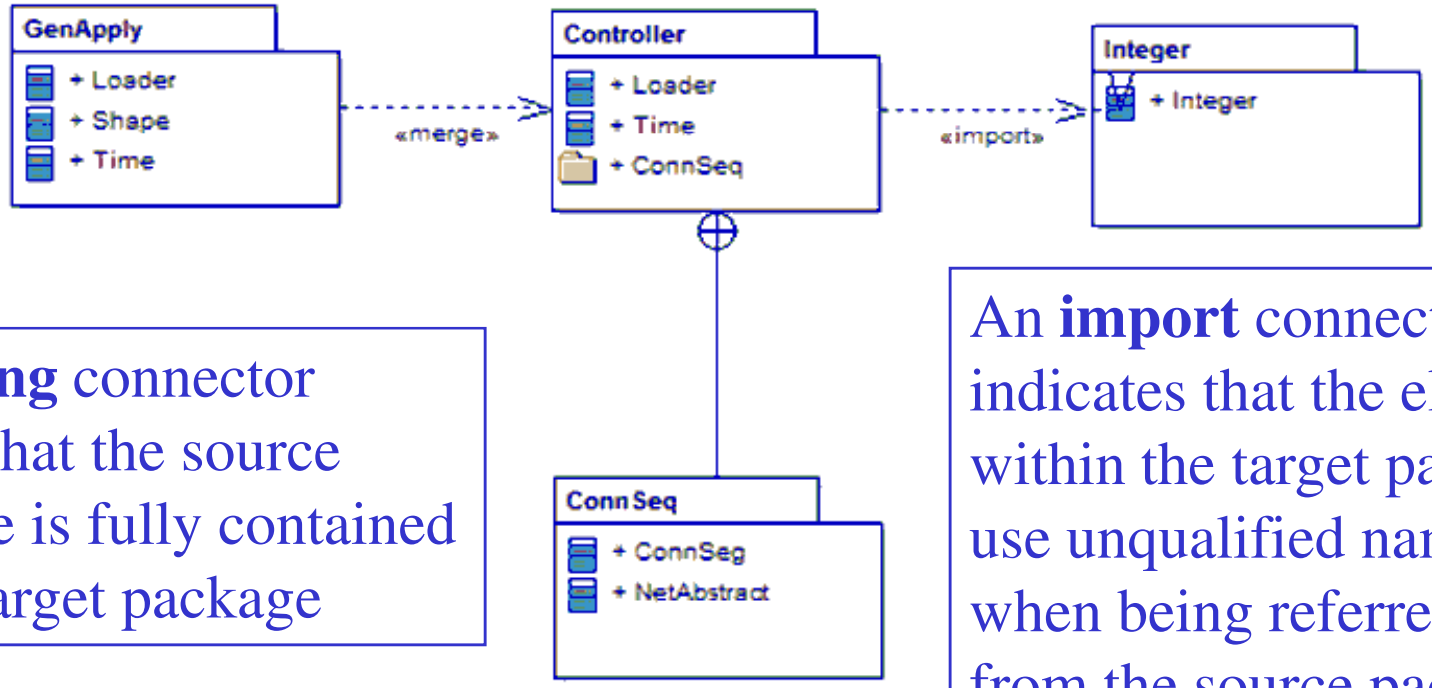
Package Diagram

- Package diagrams are used for
  - Decomposing a system into logical units of work describing the dependencies between them
  - Providing views of a system from multiple levels of abstraction
- The most common use for package diagrams is to organize use case diagrams and class diagrams, but may also be used for the other UML elements

Merge, Nesting and Import

A **merge** connector defines an implicit generalization between elements in the source package, and elements with the same name in the target package

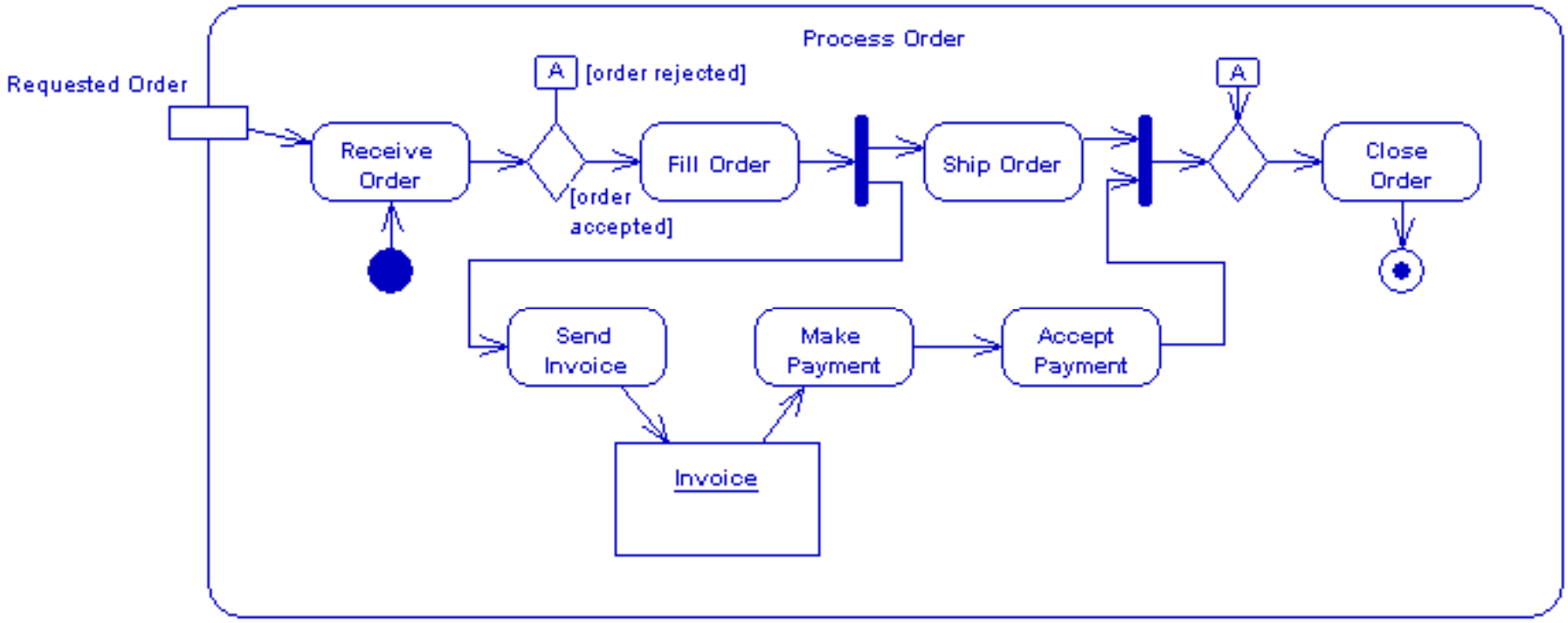
The source element definitions are expanded to include the element definitions contained in the target



A **nesting** connector shows that the source package is fully contained in the target package

An **import** connector indicates that the elements within the target package use unqualified names when being referred to from the source package

Activity Diagram



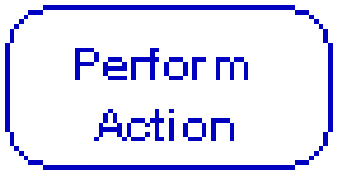
Activity Diagram

- An activity diagram is used to display the sequence of activities
- Activity diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity
- They may be used to detail situations where parallel processing may occur in the execution of some activities
- They are useful for business modeling where they are used for detailing the processes involved in business

AOT LAB

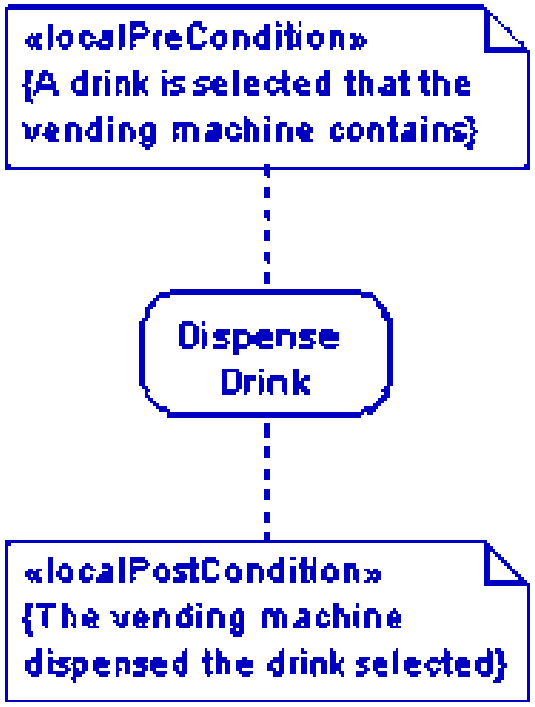
Activity and Action

An **activity** is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity



An **action** represents a single step within an activity

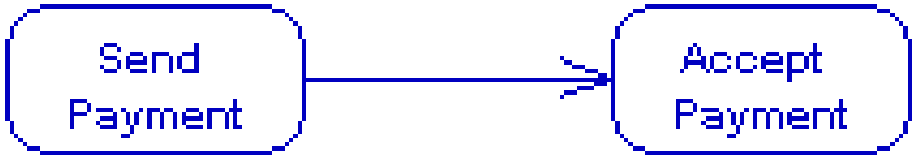
Some **constraints** can be attached to an action



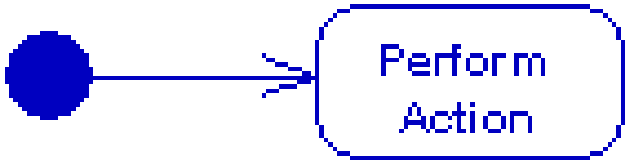
AOT LAB

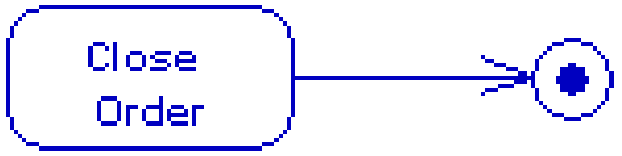
Control Flow and Endpoint Nodes

A **control flow** shows the flow of control from one action to the next

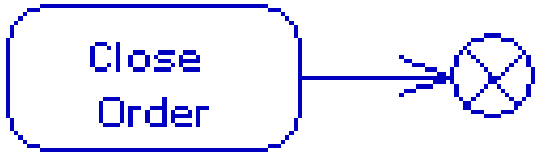


The **initial node** is depicted by a large black spot





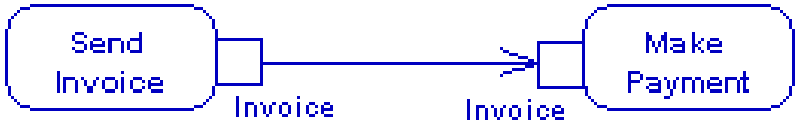
The **activity final node** denotes the end of all control flows within the activity



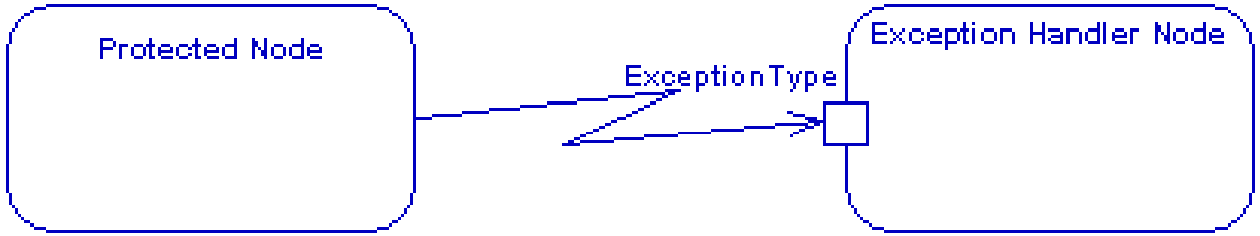
The **flow final node** denotes the end of a single control flow

AOT LAB

Object and Interrupt Flows

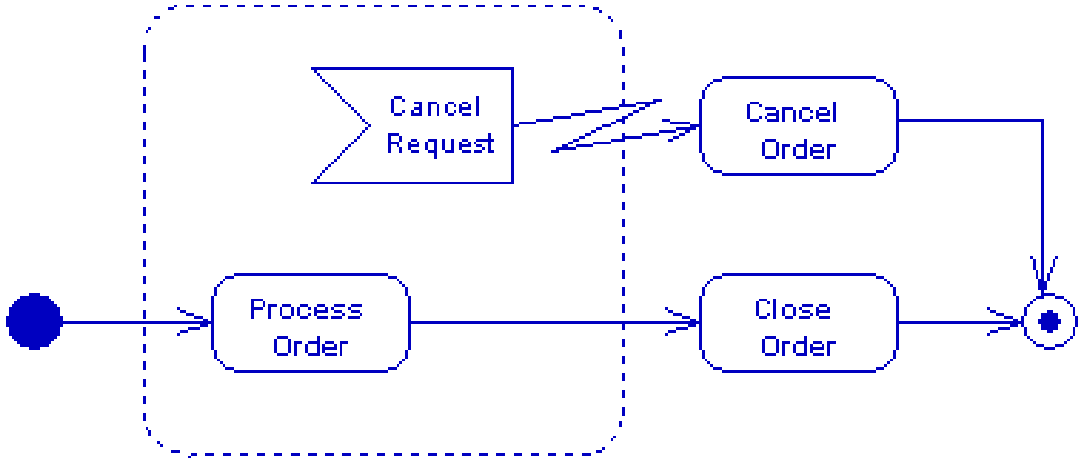


An **object flow** is a path along which objects or data can pass



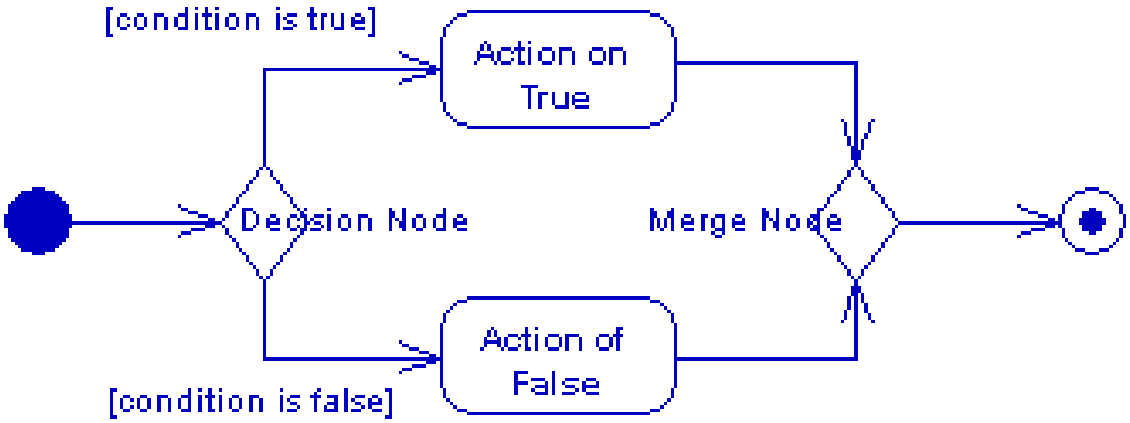
Exception handlers can be modeled on activity diagrams through the use of an **interrupt flow**

An **interruptible activity region** surrounds a group of actions that can be interrupted

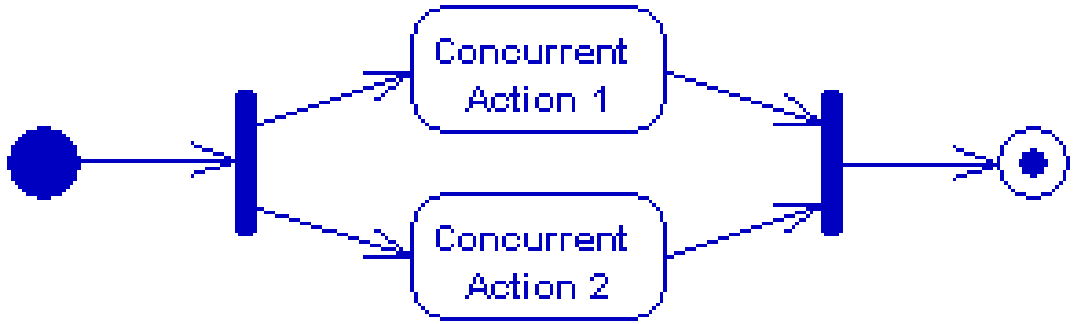


AOT LAB

Decision-Merge and Fork-Join

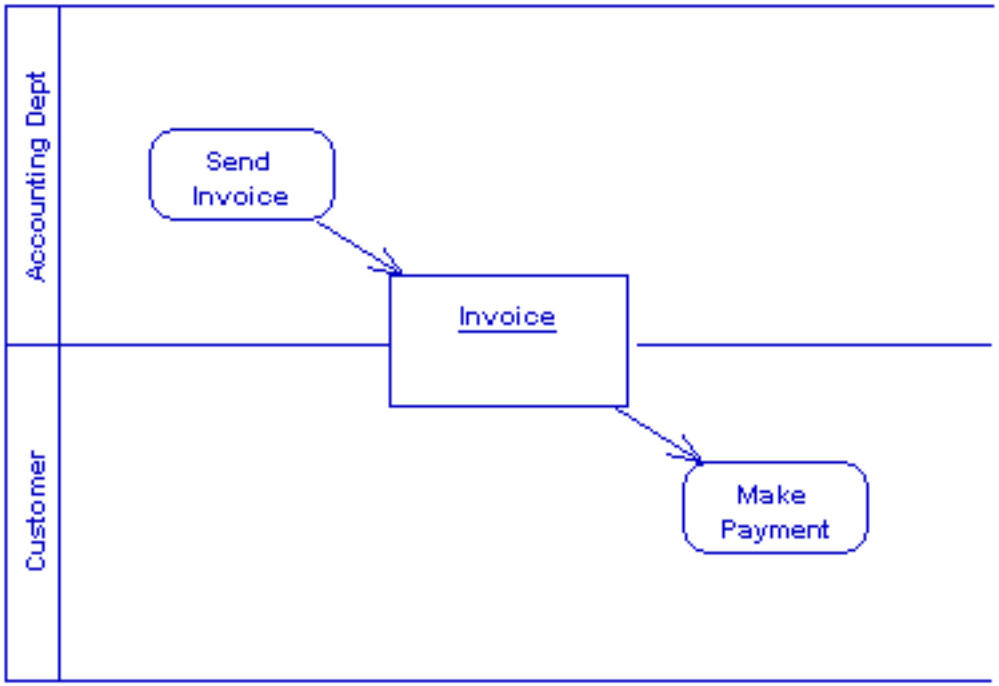


The control flows coming away from a decision node will have guard conditions which allow control to flow if the guard condition is met

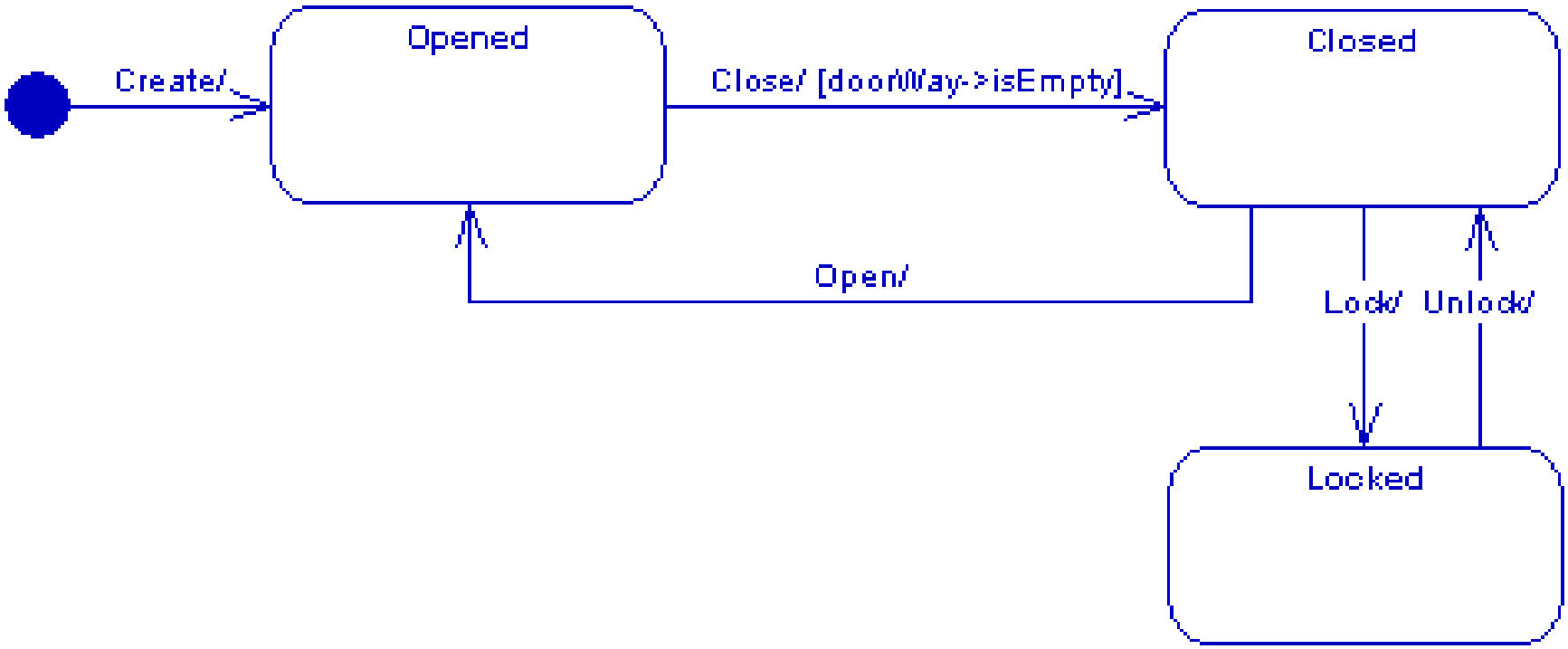


Activity Partition

An activity partition is used for logically separating the actions executed inside an activity



State Diagram



State Diagram

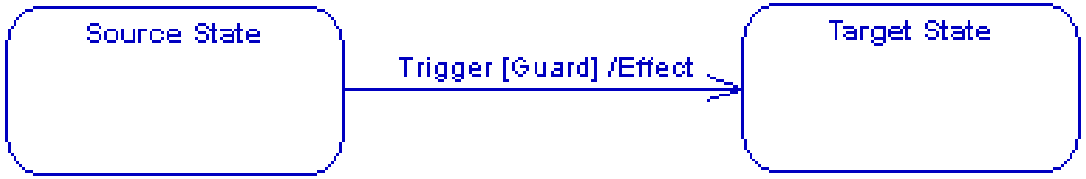
- A state machine diagram models the behavior of a single object

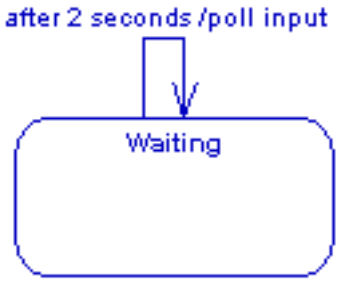
- It specifies the sequence of states that an object goes through during its lifetime in response to stimuli from the environment

State and Transition

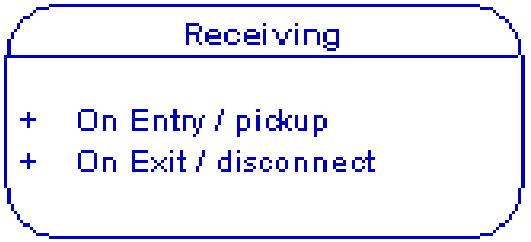
A **state** is denoted by a round-cornered rectangle with the name of the state written inside it

A **transition** from one state to the next is denoted by a line with arrowhead and may have a trigger, a guard and an effect





If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions

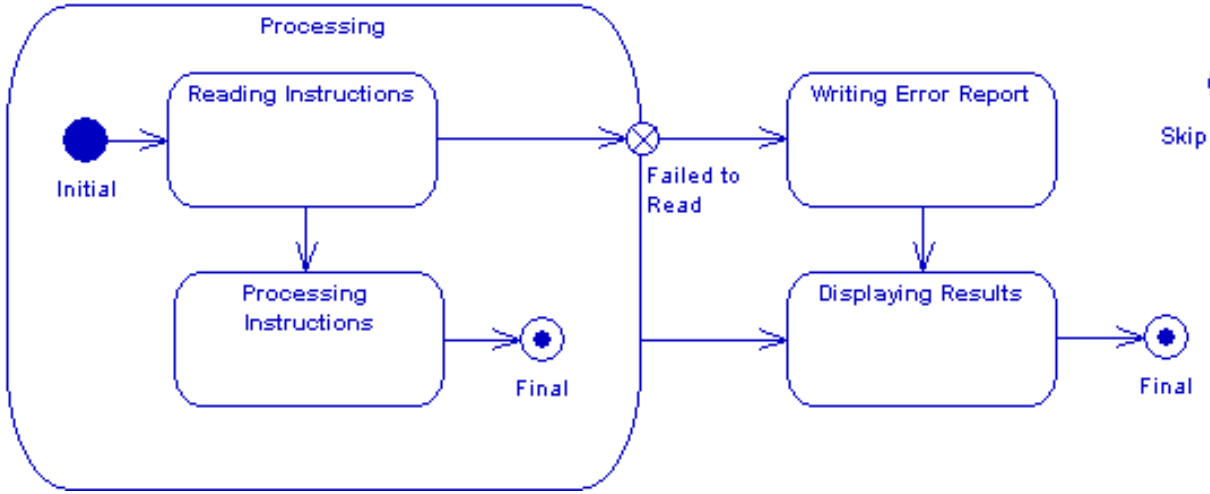


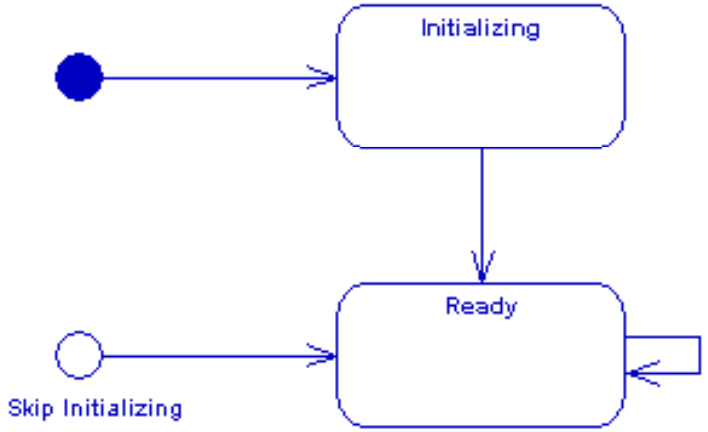
AOT LAB

Enter and Exit Points

The initial state and the final state are respectively denoted by a filled black circle and a circle with a dot inside and may also be labeled with a name

Sometimes may be possible to have a different exit point



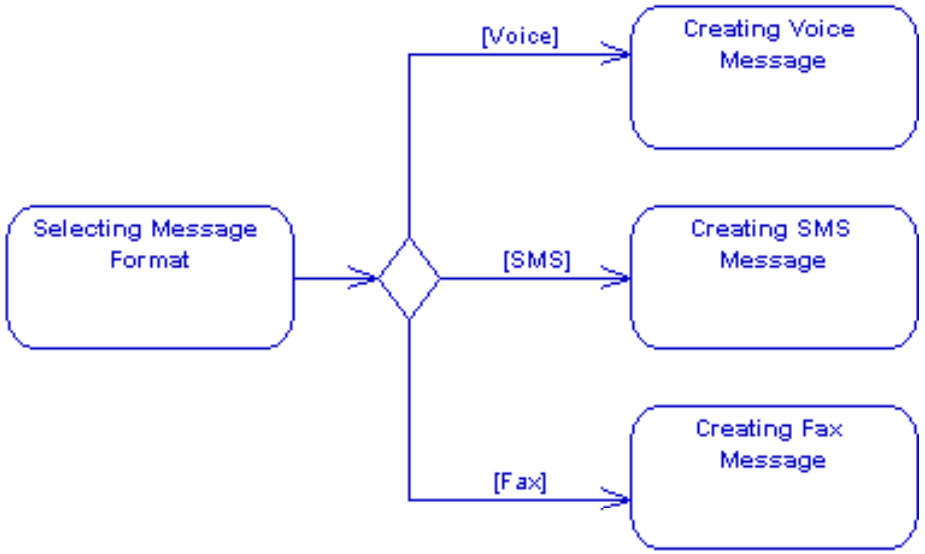


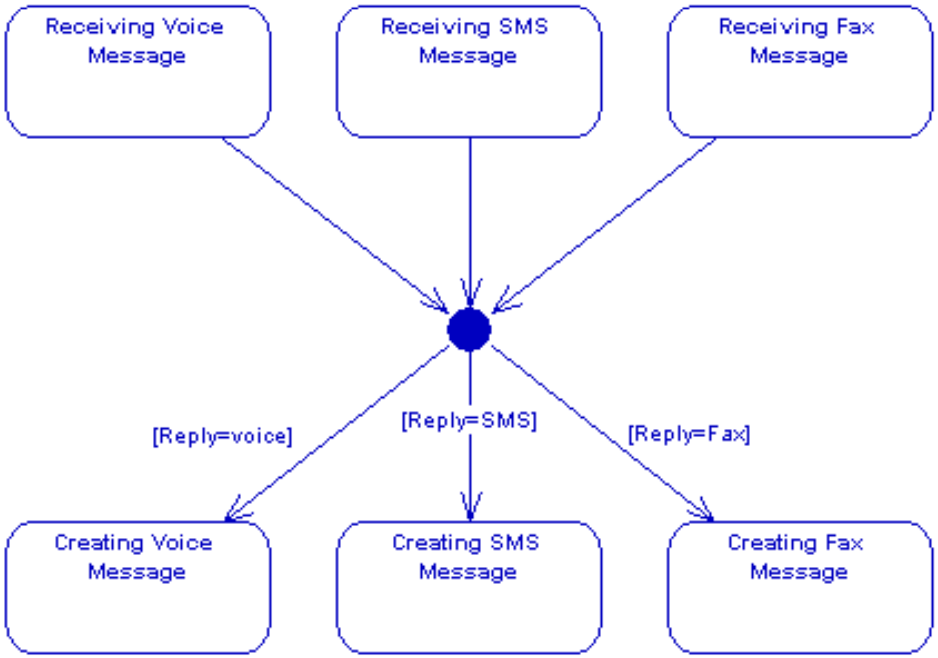
Sometimes may be possible to have an alternative start point

AOT LAB

Choice and Junction Pseudo-State

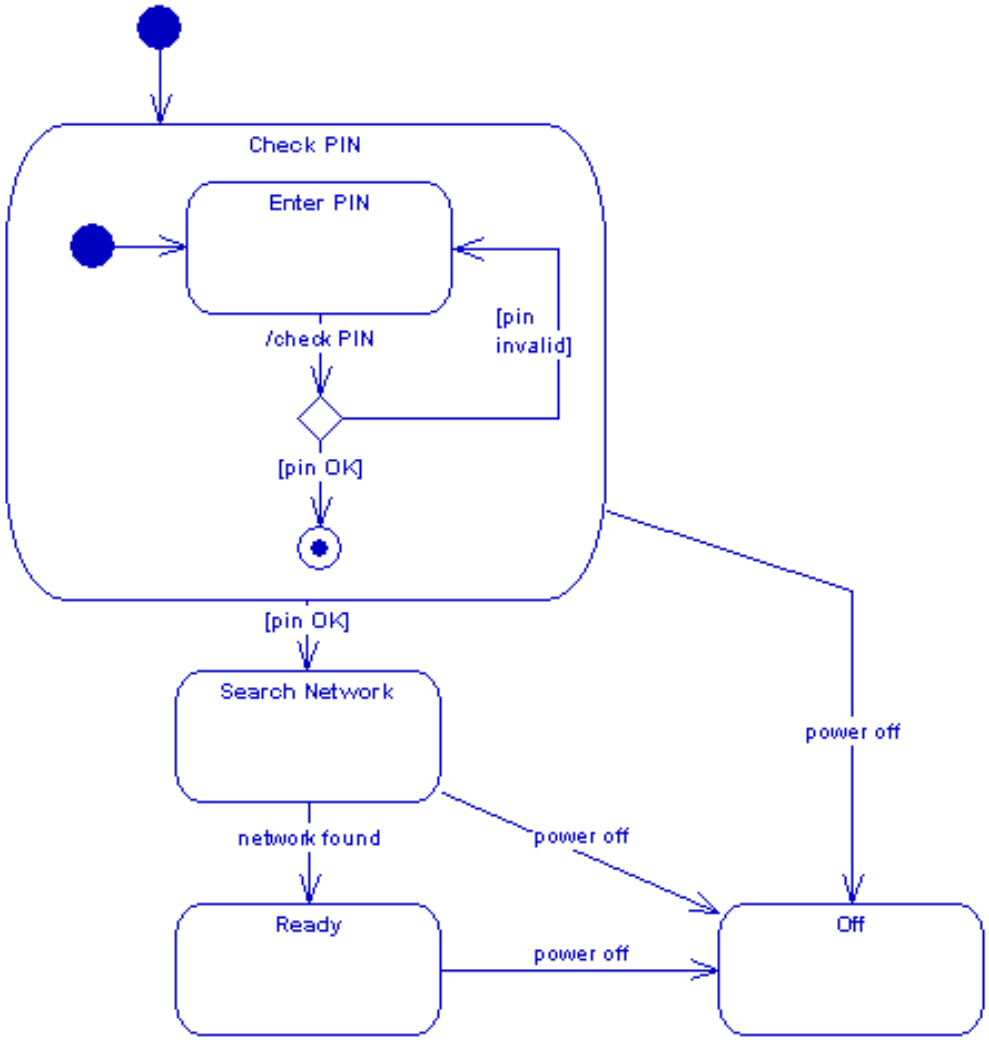
A **choice** pseudo-state is represented by a diamond with one transition arriving and two or more transitions leaving

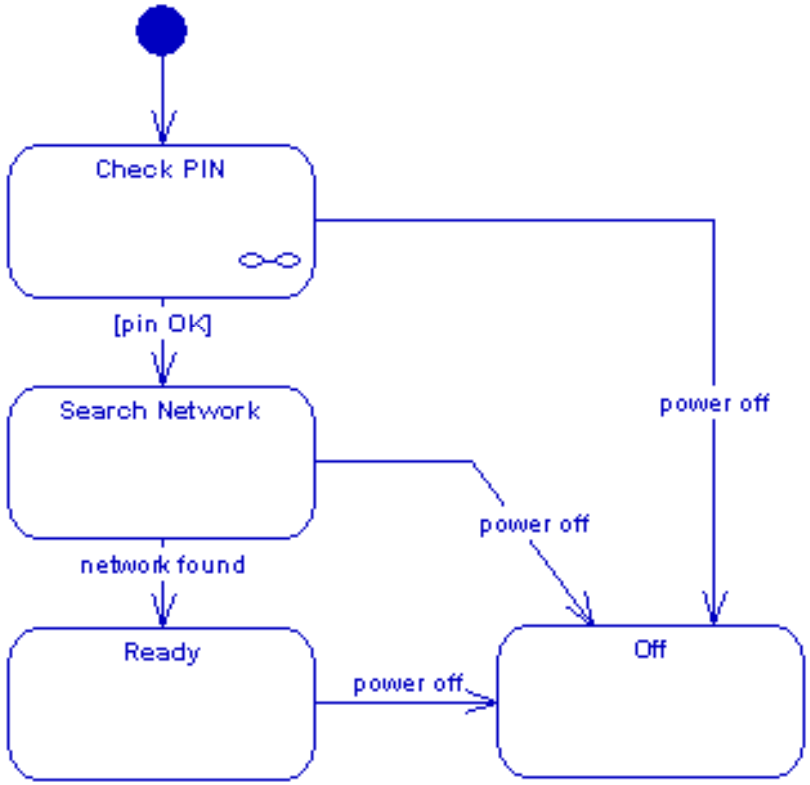




A **junction** pseudo-state can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition

Composition

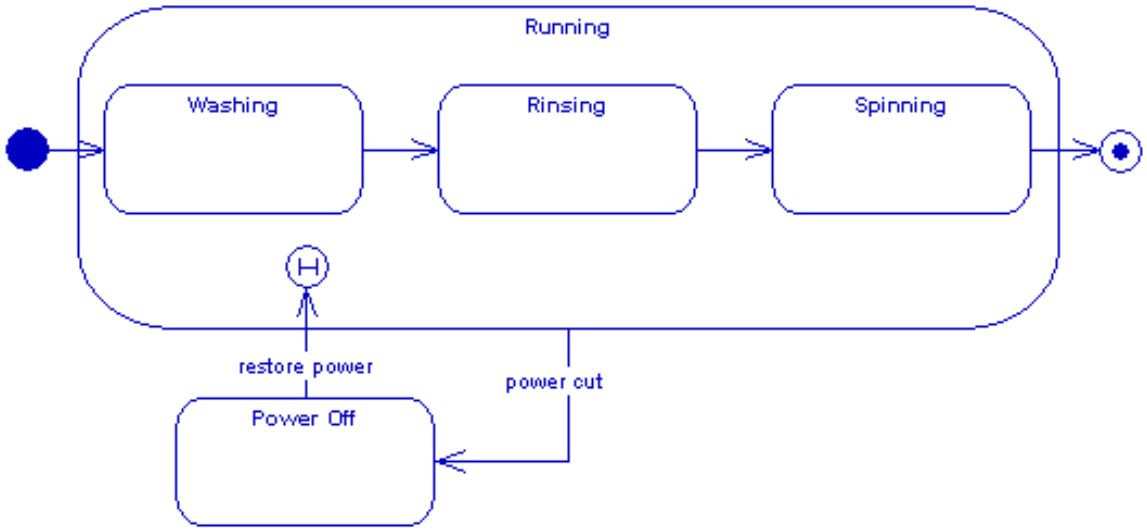


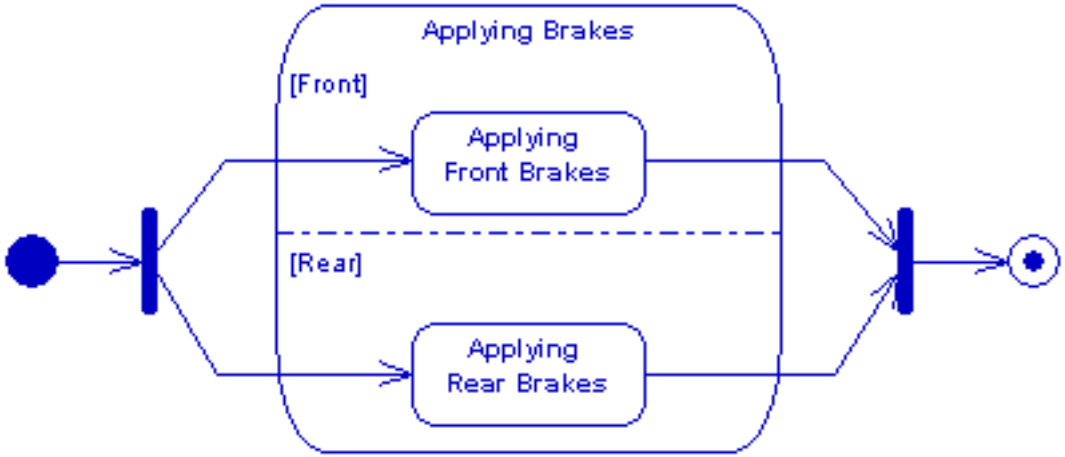


AOT LAB

History State and Concurrent Regions

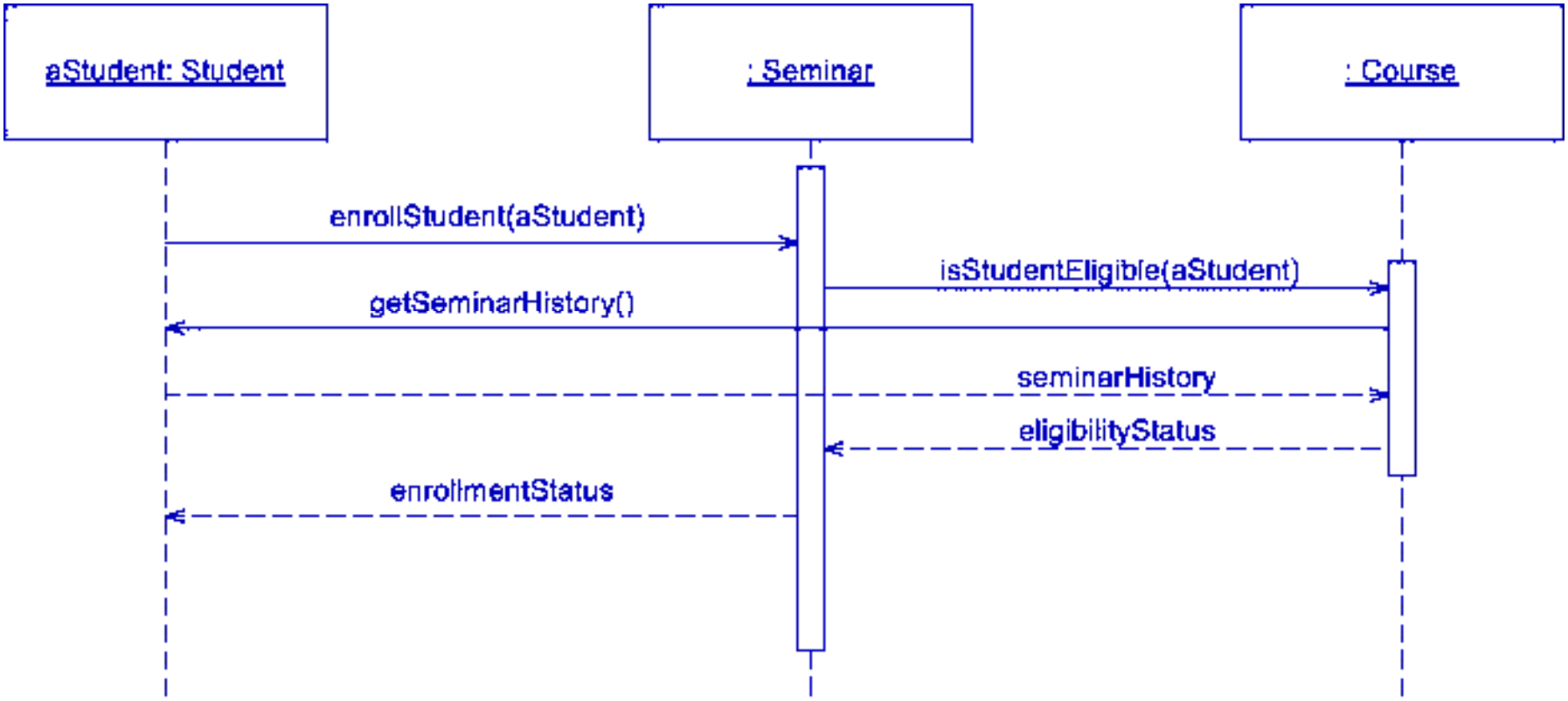
A history state is used to remember the last state of a state machine when it was interrupted





A state may be divided into regions containing sub-states that exist and execute concurrently

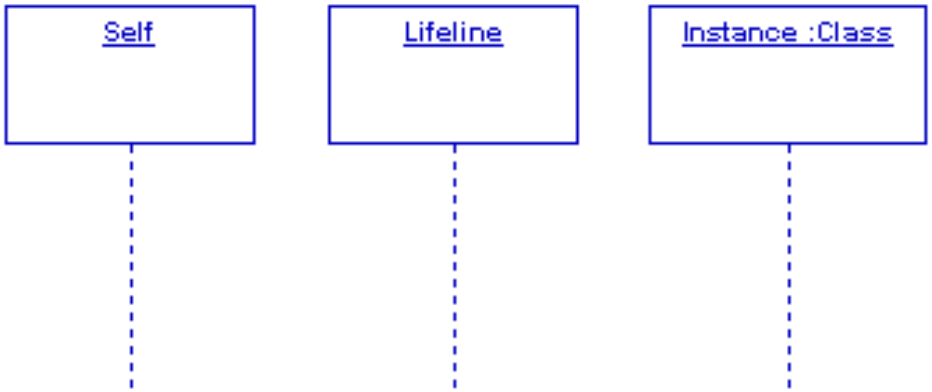
Sequence Diagram



Sequence Diagram

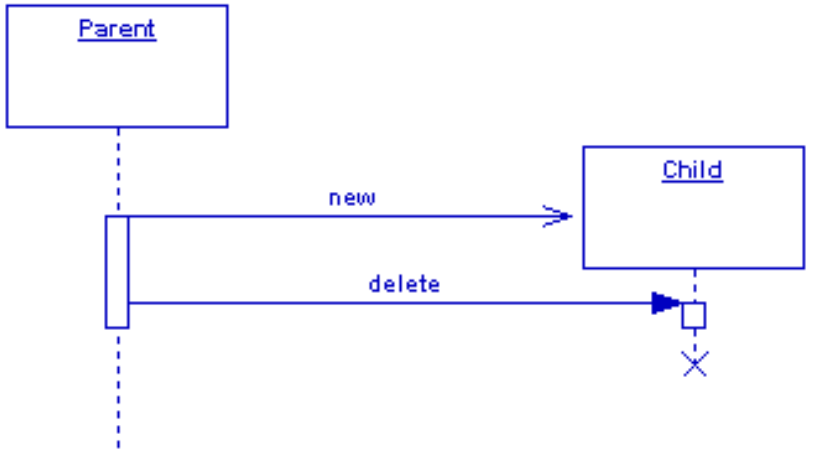
- Describes how a process is performed by a group of objects by a sequential set of interactions
- Facilitates assignment of responsibilities to classes and helps finding out new methods and new classes
- These diagrams contain the following elements:
  - Roles, which represent roles that objects may play within the interaction
  - Lifelines, which represent the existence of an object over a period of time
  - Activations, which represent the time during which an object is performing an operation
  - Messages, which represent communication between objects

Lifeline



A lifeline represents an individual participant in a sequence diagram

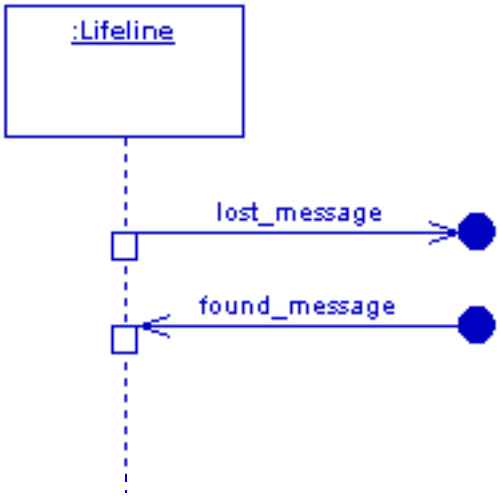
If lifeline name is "self", it indicates that the lifeline represents the classifier which owns the sequence diagram

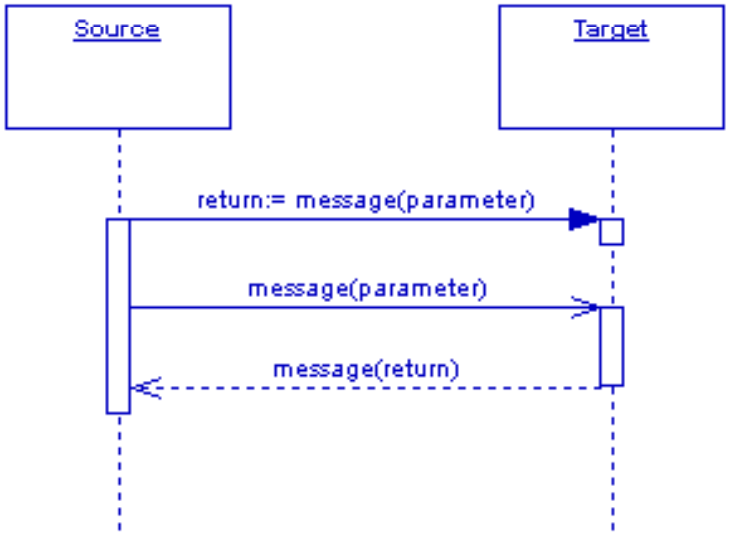


The lifeline can be started and ended

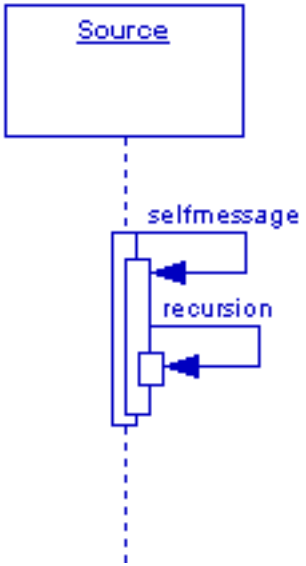
Message

A message is displayed as arrows and can be complete, lost or found, synchronous or asynchronous, call or signal



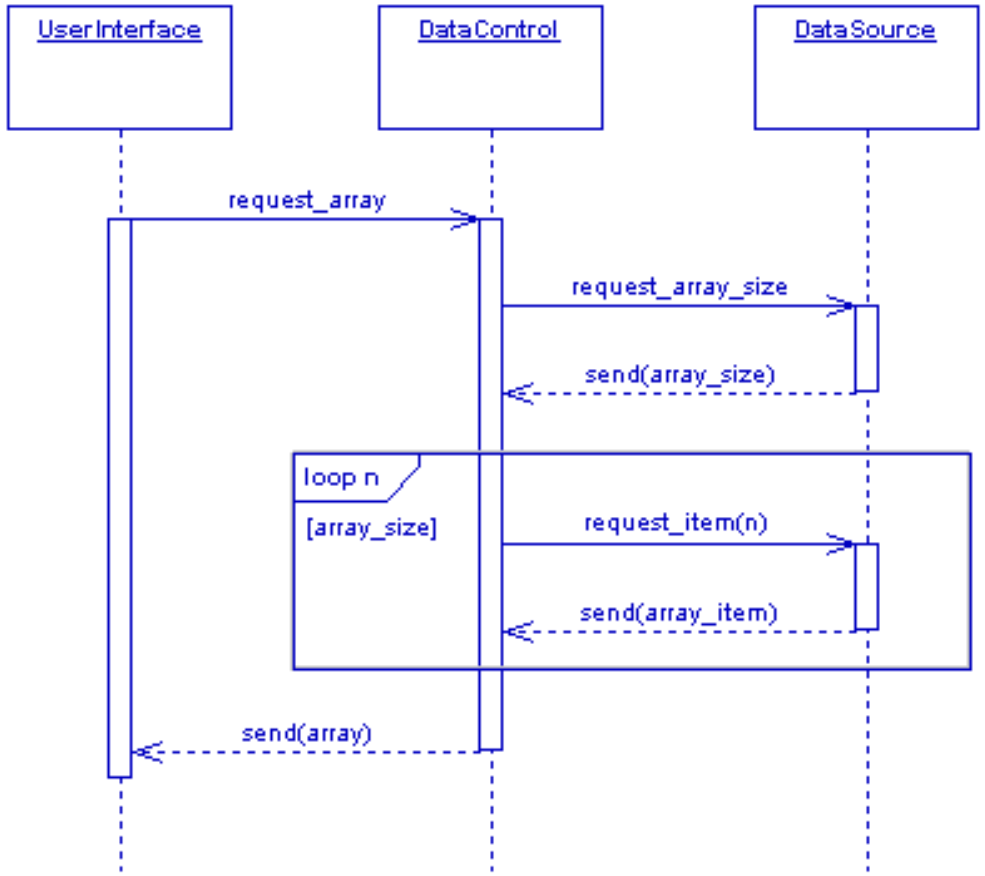


A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object



Fragment

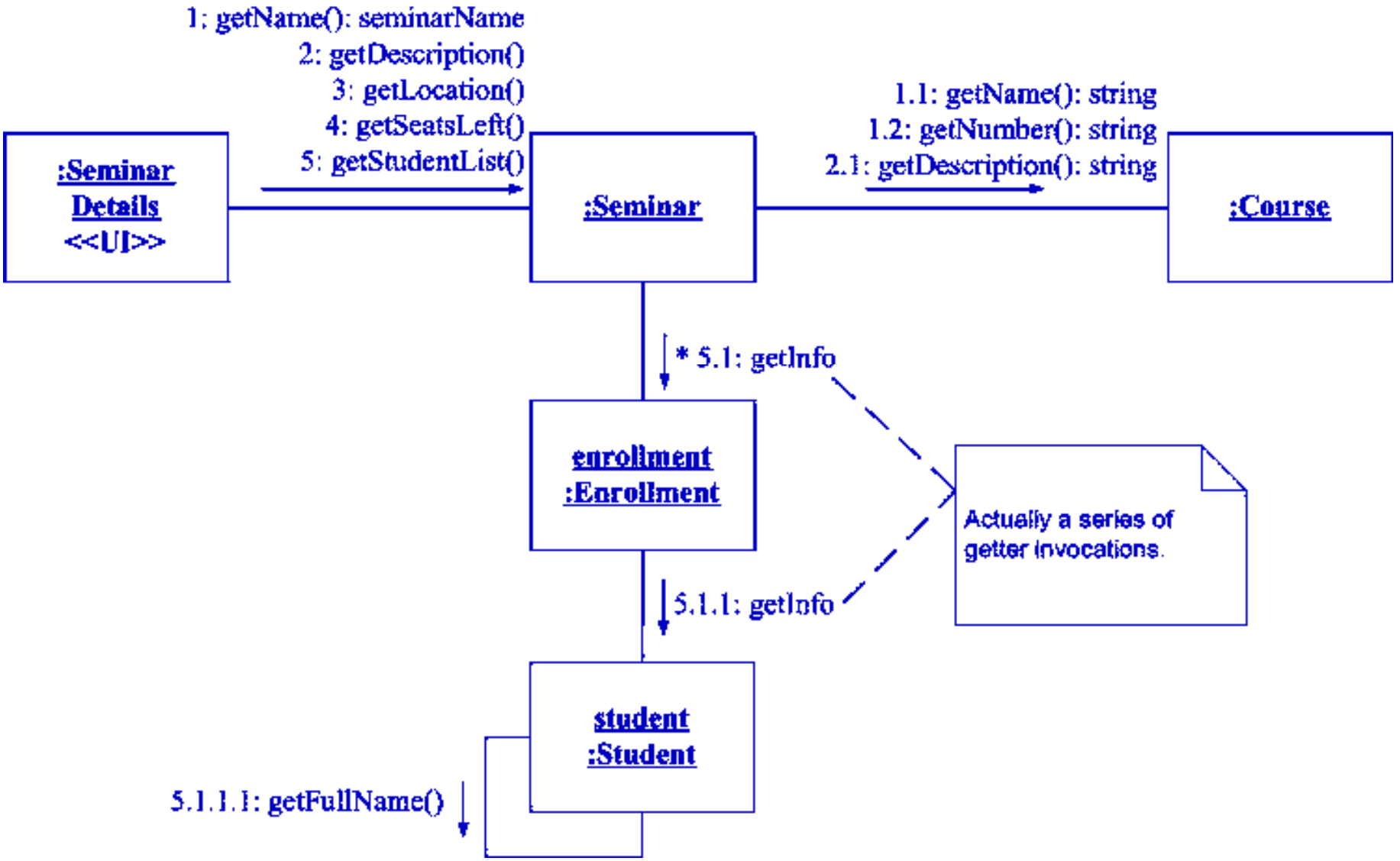
A fragments allows the representation of complex procedural logic inside a sequence diagram



- Alternative fragment models if…then…else constructs
- Option fragment models switch constructs
- Parallel fragment models concurrent processing
- Loop fragment encloses a series of messages which are repeated
- …

AOT LAB

Communication Diagram

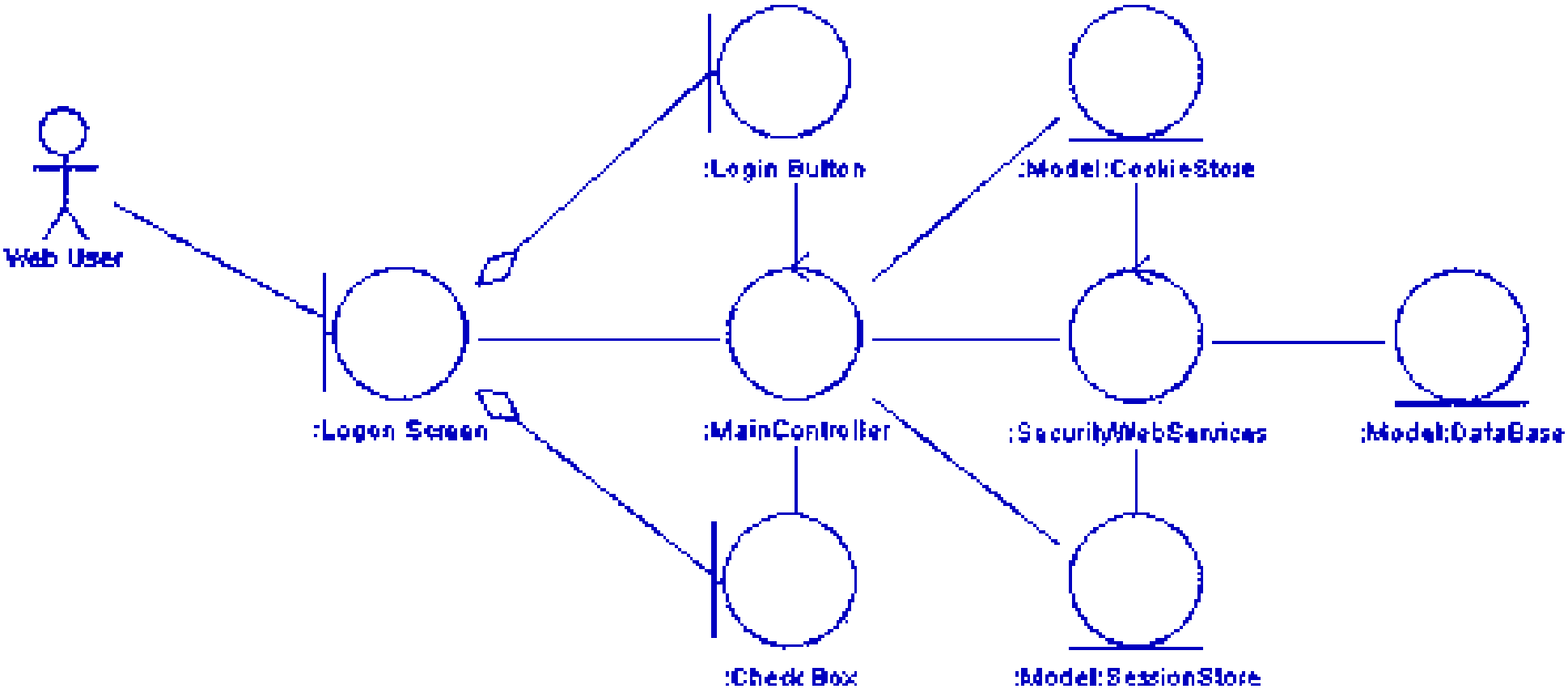


Communication Diagram

- A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus is on object relationships:
  - Objects are shown with association connectors between them
  - Messages are added to the associations and show as short arrows pointing in the direction of the message flow
  - The sequence of messages is shown through a numbering scheme
- They provides an alternative view to the sequence diagram in a format based on structure rather than time

AOT LAB

Robustness Diagram



Robustness Diagram

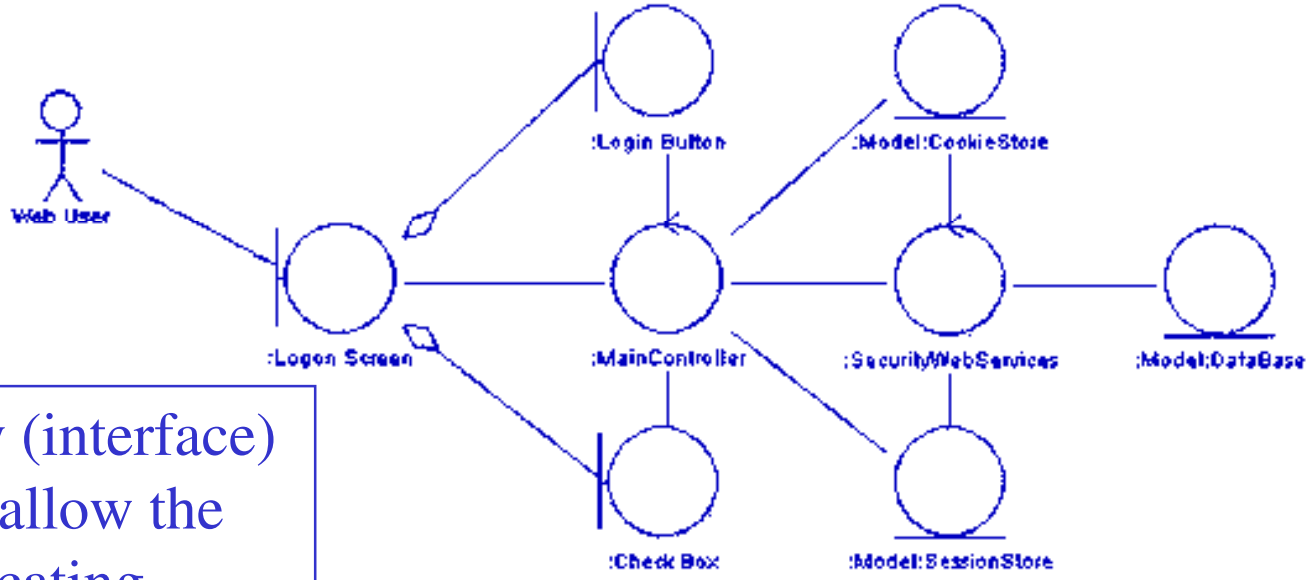
- A robustness diagram is basically a **simplified** UML **communication diagram**

- Their purpose is to provide a means of refining the use cases:
  - Checking their correctness
  - Determining if they address all necessary alternate courses of action
  - Discovering all the objects necessary to the design

AOT LAB

Actor, Boundary, Control and Entity

Control elements act as the glue between boundary and entity elements, implementing the logic required to manage the various elements and their interactions

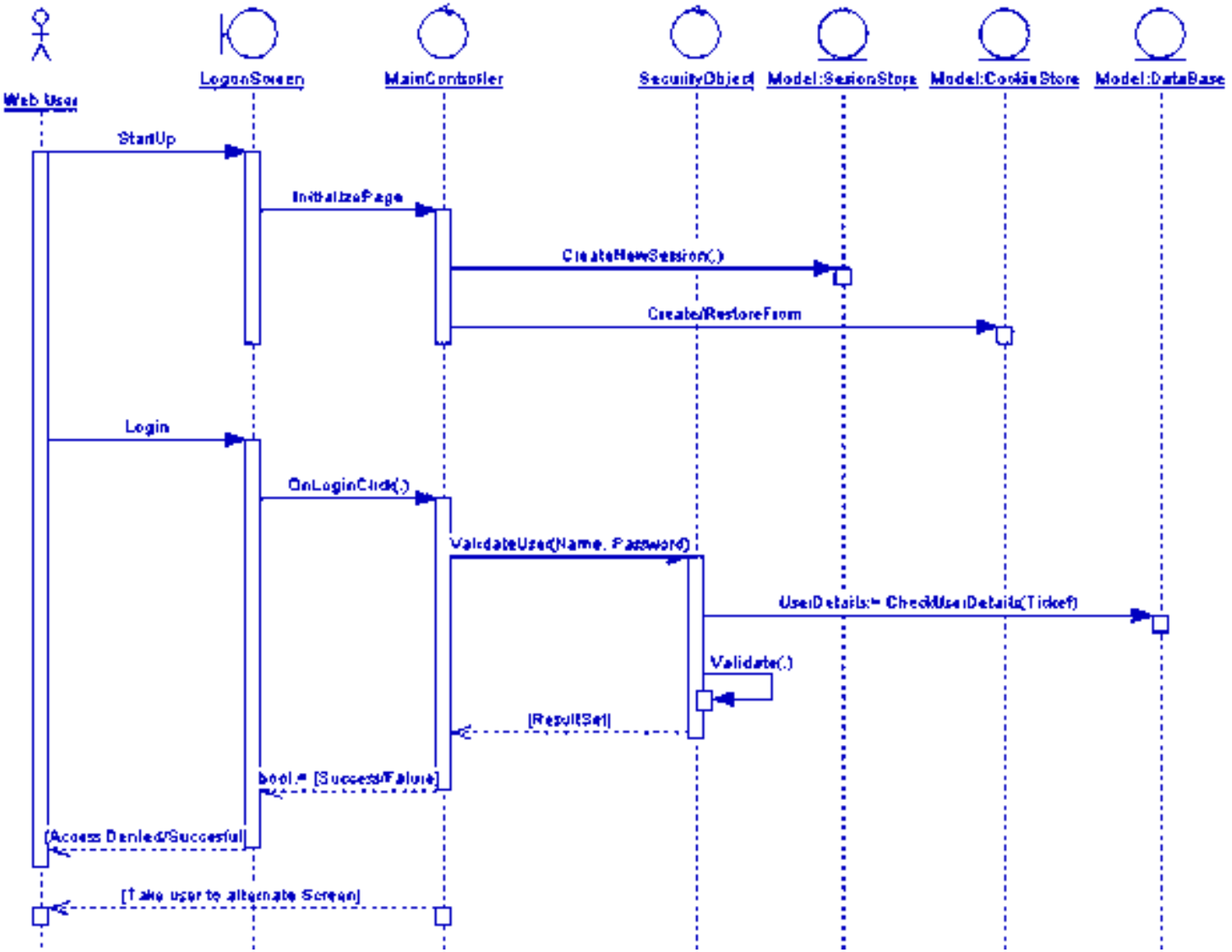


Boundary (interface) elements allow the communicating between actors and the internal parts of the system

Entity elements represent information unit of the system

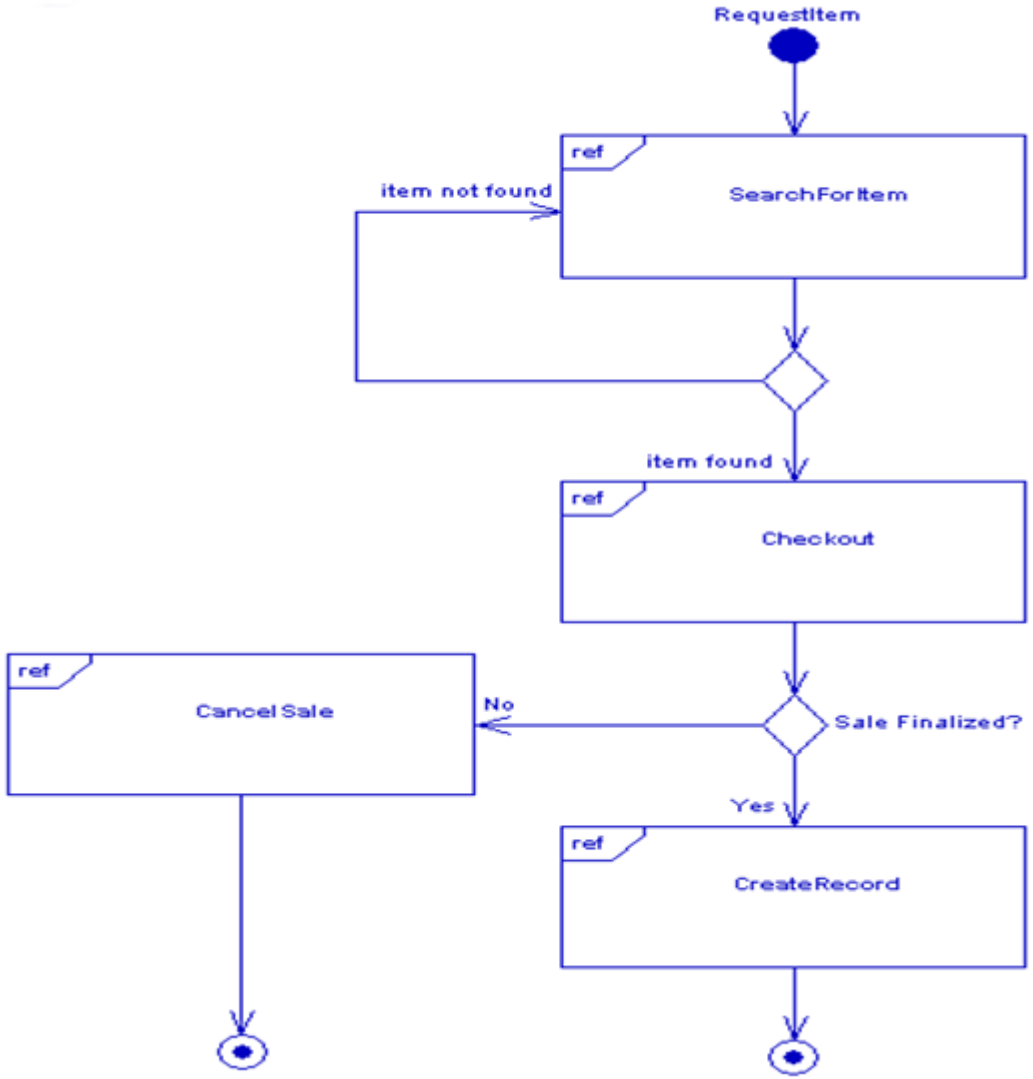
AOT LAB

Logon Sequence Diagram



AOT LAB

Interaction Overview Diagram



Interaction Overview Diagram

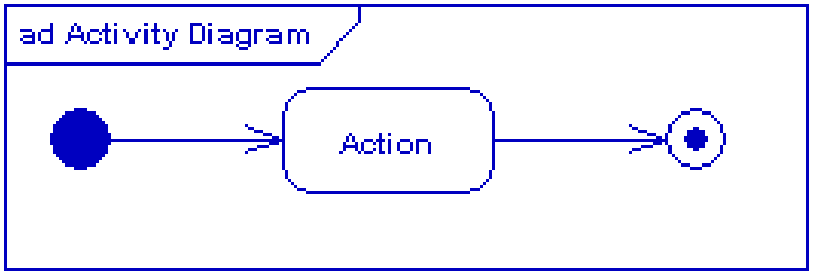
- An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams (sequence, communication, interaction overview and timing diagrams)
- Most of the notation for interaction overview diagrams is the same for activity diagrams
  - For example, initial, final, decision, merge, fork and join nodes are all the same
- However, interaction overview diagrams introduce two new elements: interaction occurrences and interaction elements

AOT LAB

Occurrence and Element

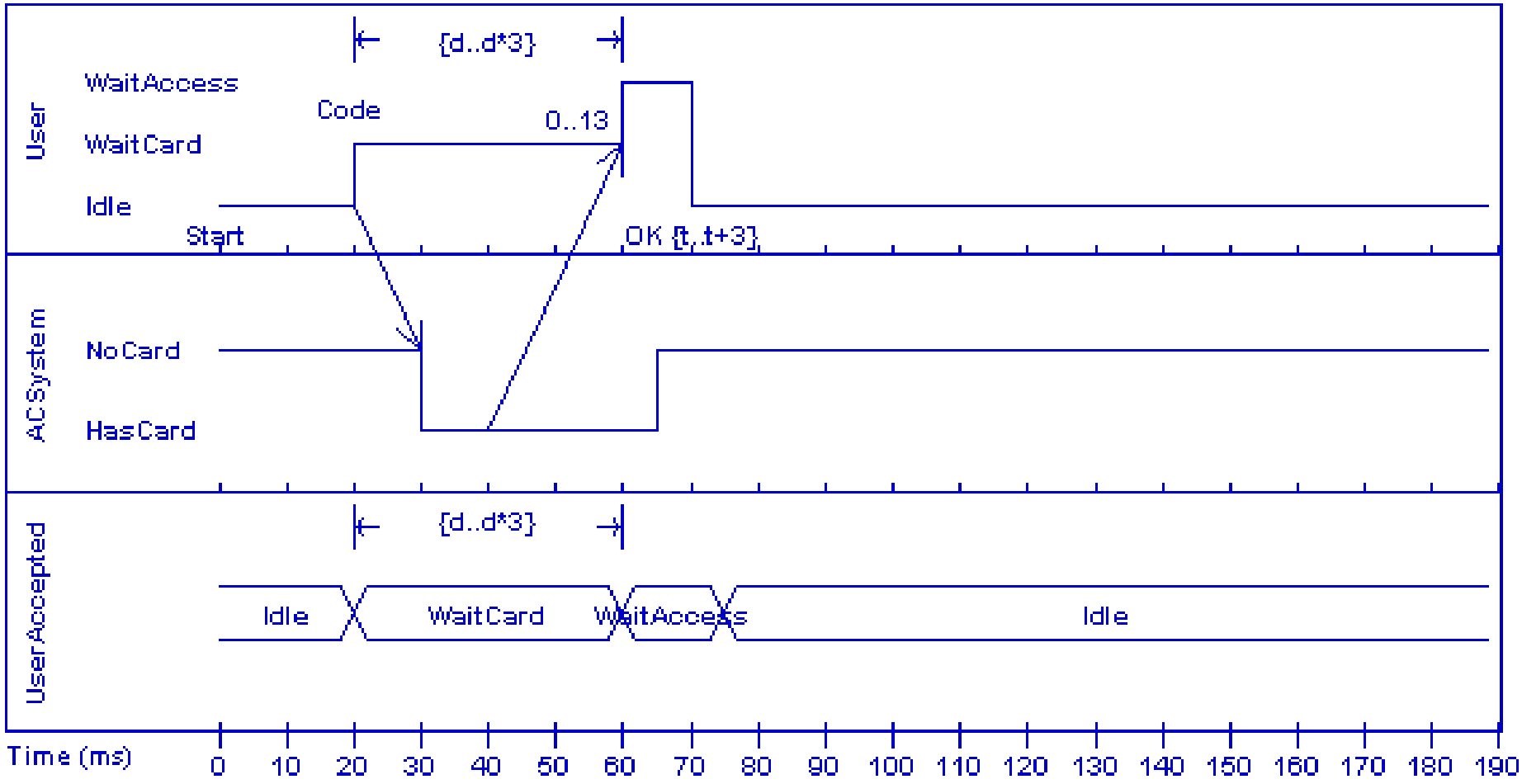
Occurrence

Element



AOT LAB

Timing Diagram



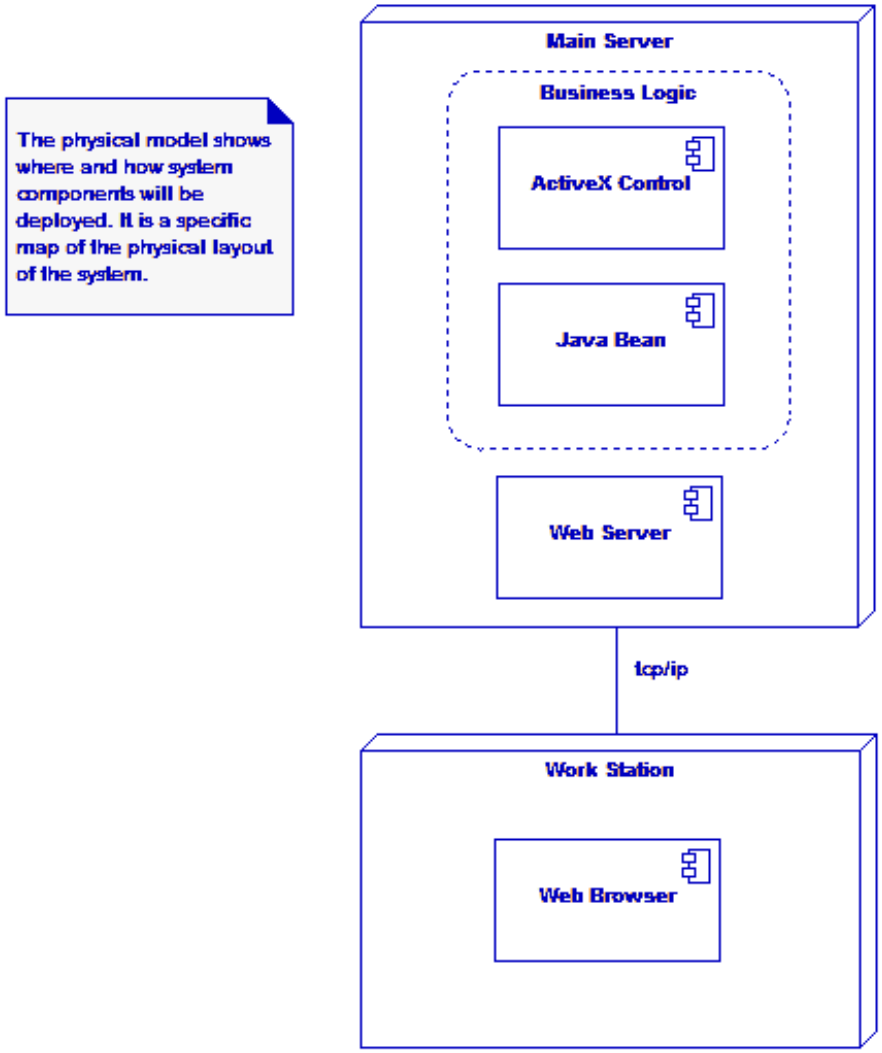
Timing Diagram

- A timing diagram is used to display the change in state or value of one or more elements over time

- It can also show the interaction between timed events and the time and duration constraints that govern them

AOT LAB

Deployment Diagram

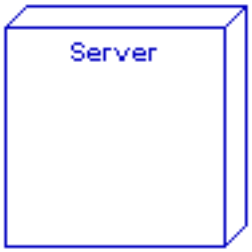


Deployment Diagram

- A deployment diagram models the run-time architecture of a system
  - Describes the configuration of hardware in a system in terms of nodes and connections
  - Describes the physical relationships between software and hardware
  - Displays how artifacts are installed and move around a distributed system

AOT LAB

Node and Artefact



A node is either a hardware or software element. A node instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon

An artifact is a product of the software development process.

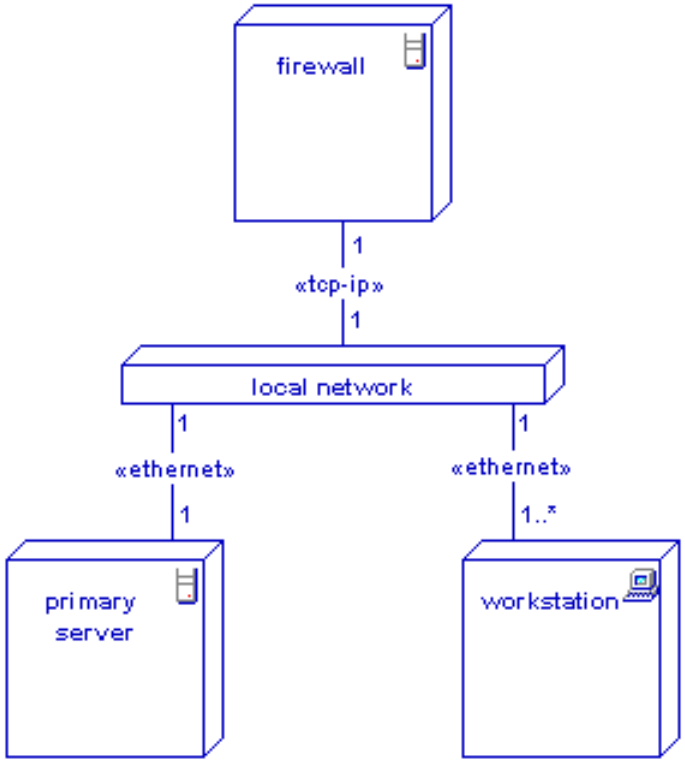
That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.

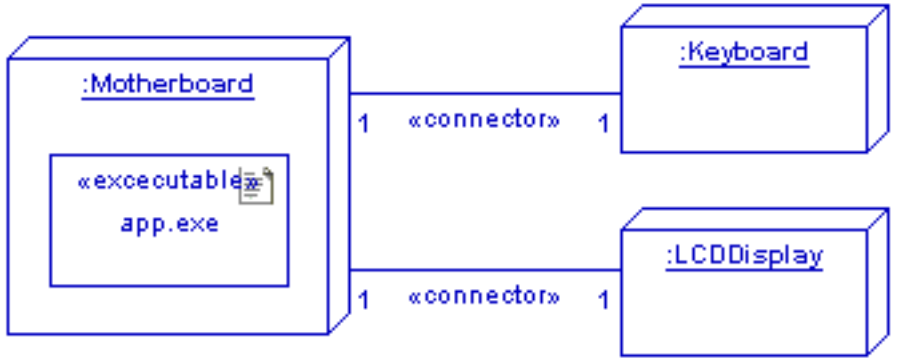
A number of standard stereotypes are provided for nodes, namely «cd rom», «computer», «pc», «pc client», «pc server», etc.

AOT LAB

Association and Composition

In the context of a deployment diagram, an association represents a communication path between nodes



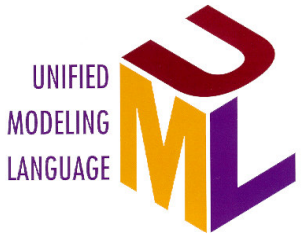


A node can contain other elements, such as components or artifacts

Use of UML Diagrams

Activity	High
Class	High
Communication	Low
Component	Medium
Composite Structural	Low
Deployment	Medium
Interaction Overview	Low
Object	Low
Package	Low
Sequence	High
State	Medium
Timing	Low
Use Case	Medium

AOT LAB



Unified Modeling Language

Object Constraint Language

What is OCL?

- The Object Constraint Language (OCL) is a language that enables the description of expressions and constraints on object-oriented
- OCL is a typed formal language with a precise syntax and semantics
- OCL was developed at IBM by Jos Warmer as a language for business modeling within IBM
- OCL is not a programming language
  - It is not possible to write program logic or flow control in OCL

Why Use OCL in UML Models?

- UML diagrams are typically not refined enough to provide all the relevant aspects of a specification
- For instance, there may be the need to describe additional constraints on the relationships between model entities and that can be described through:
  - Natural language expressions, that always result in ambiguities
  - Formal language expressions, they are usable to persons with a strong mathematical background
- OCL has been developed to fill this gap:
  - It is a formal language, but remains easy to read and write for all the business or system modelers

OCL Constraints

- OCL is based on constraints
  - Constraints are restrictions on one or more values of an object-oriented model or system
- OCL constraints are declarative
  - They specify what must be true not what must be done
- OCL constraints have no side effects
  - Evaluating an OCL expression does not change the state of the system
- OCL constraints have formal syntax and semantics
  - Their interpretation is unambiguous

Advantages of Constraints

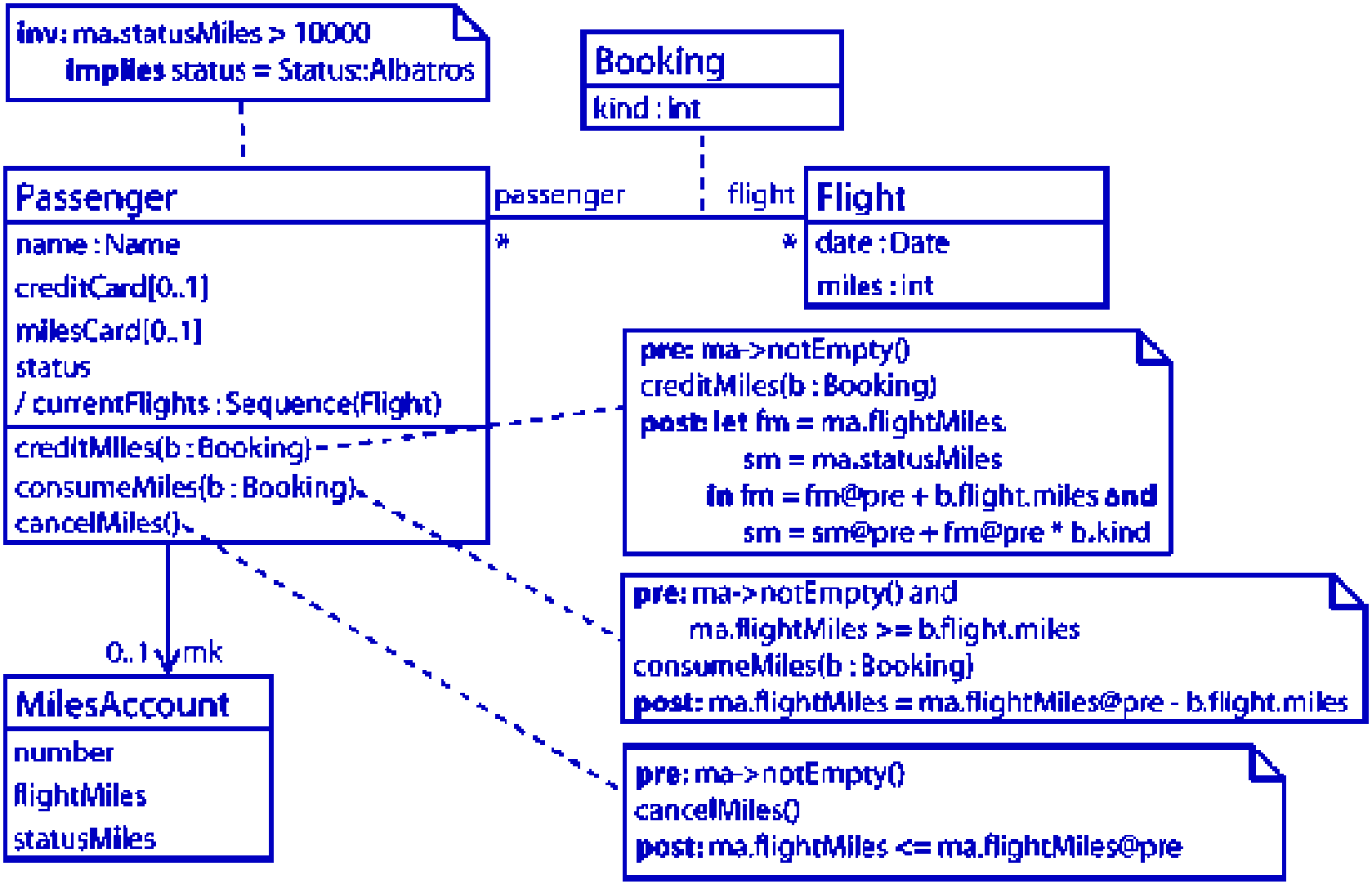
- Better documentation
  - Constraints add information about the model elements and their relationships to the visual models used in UML
  - It is way of documenting UML models
- More precision
  - OCL constraints have formal semantics, hence, can be used to reduce the ambiguity in UML models
- Communication without misunderstanding
  - UML models are used to communicate between developers
  - Using OCL constraints modelers can communicate unambiguously

Where Use OCL?

- As a query language
- To specify invariants on classes and types in the class model
- To specify type invariant for stereotypes
- To describe pre / post conditions on operations
- To describe guards
- To specify target (sets) for messages and actions
- To specify constraints on operations
- To specify derivation rules for attributes for any expression over a UML model

AOT LAB

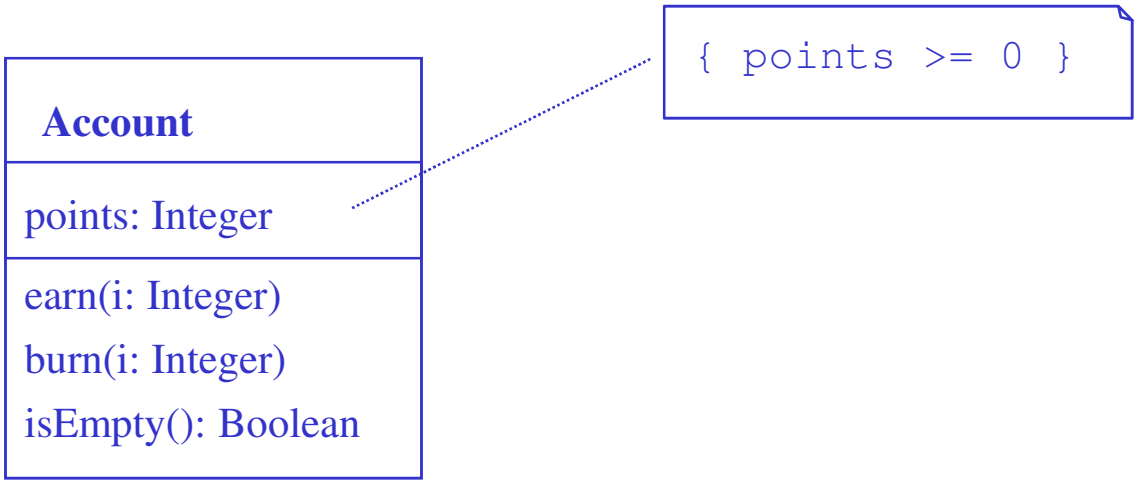
Where Use OCL?



Invariant

- An invariant is a constraint that is connected to a modeling element: class, and interface or type and has to hold for all their instances
- An invariant must be true at all times when the instance is at rest
- An instance is not at rest when an operation is under execution

Invariant

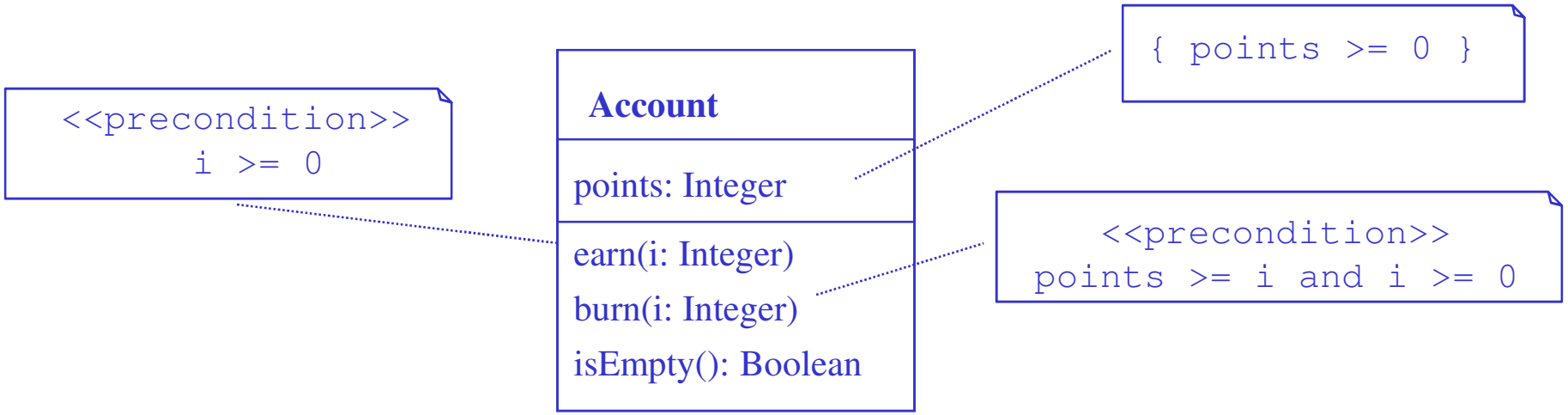

Account
points: Integer
earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean
{ points >= 0 }

Precondition

- A precondition is a constraint that must be true when an operation is invoked
- It is the responsibility of the caller to satisfy the condition
- This condition is supposed to be true, and anything else is a programming error
  - If the condition is not satisfied, no statement can be made about the integrity of the operation or the system
  - In practice, explicitly checking preconditions by the receiver may detect many errors

AOT LAB

Precondition

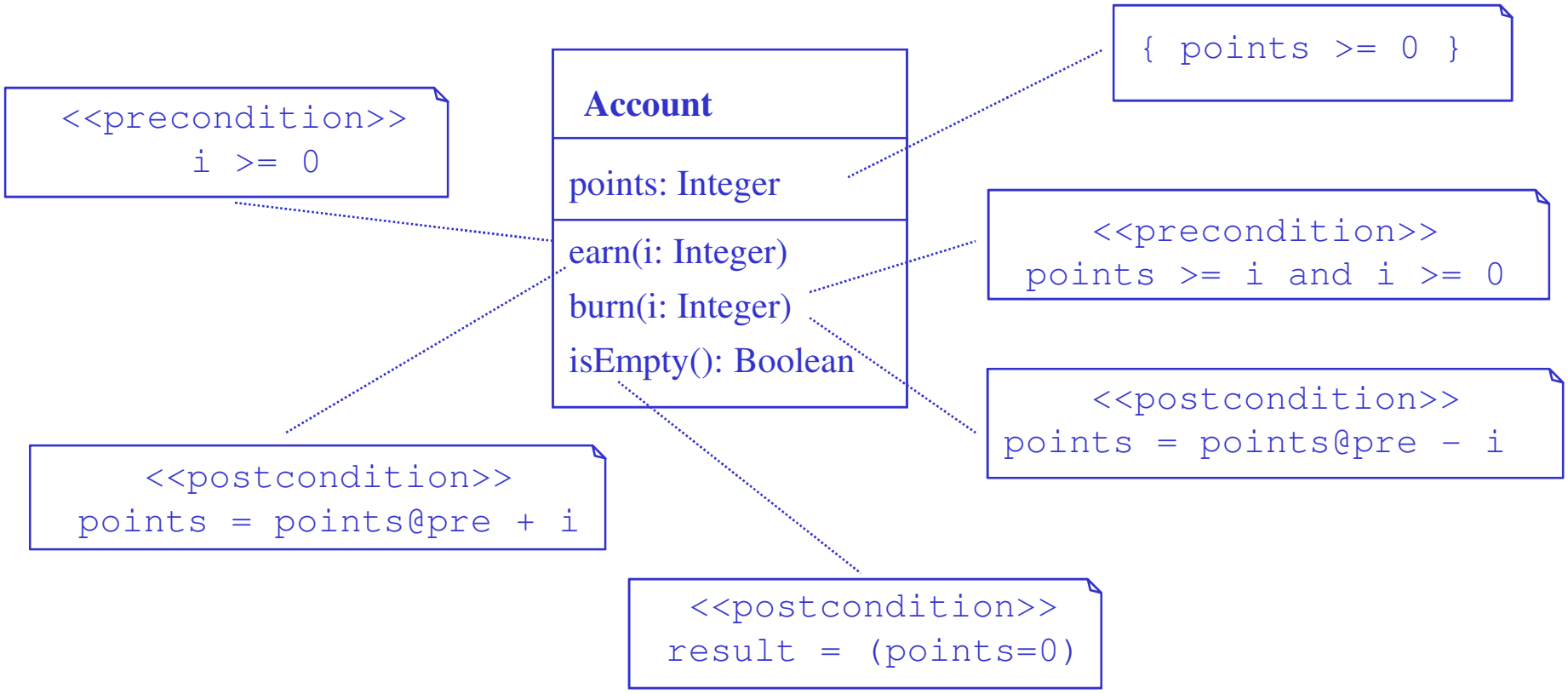


Postcondition

- A postcondition is a constraint that must be true after the completion of an operation
- This condition is supposed to be true, and anything else is a programming error
- It can be useful to test the postcondition after the operation, but this is in the nature of debugging a program

AOT LAB

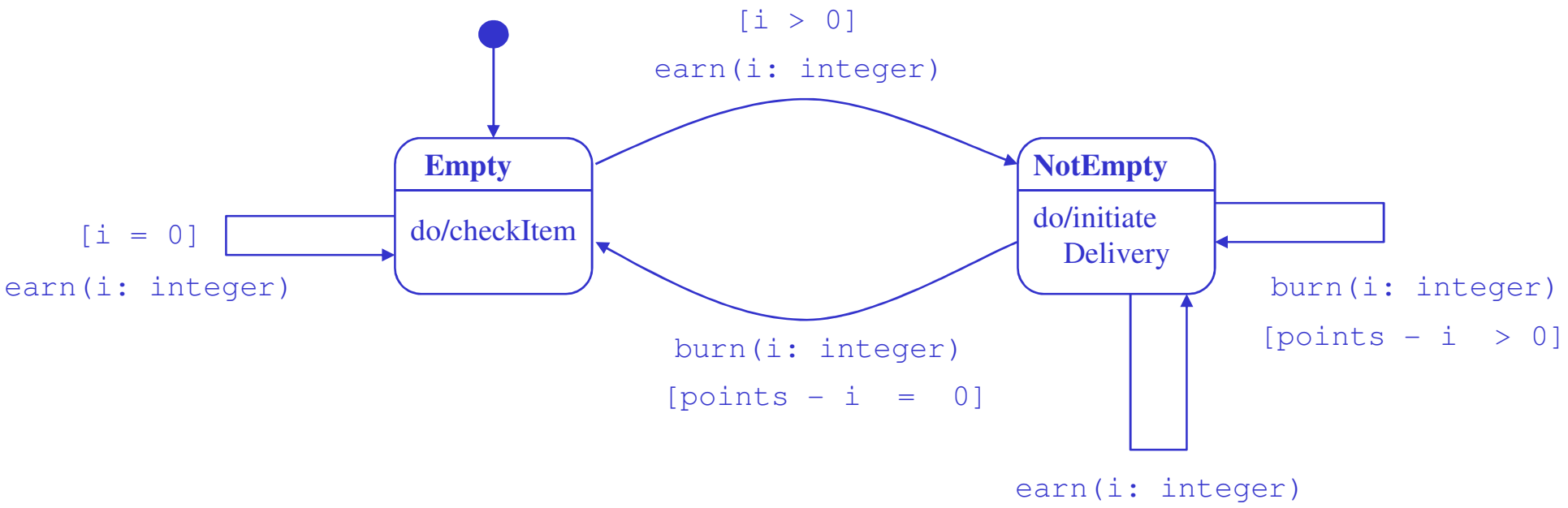
Postcondition



Guard

- A guard is a constraint that must be true before a transition can occur
- A guard is evaluated before the transition so can be thought of as a pre-condition
- A guard is usually used in activity and state diagrams

Guard


[i > 0]
earn(i: integer)
Empty
do/checkItem
NotEmpty
do/initiate
Delivery
[i = 0]
earn(i: integer)
burn(i: integer)
[points - i = 0]
burn(i: integer)
[points - i > 0]
earn(i: integer)

OCL Expressions and Constraints

- Each OCL expression has a result
  - The value that results by evaluating the expression
- Each OCL expressions can contain only query operations
  - Query operations return a value, but do not change anything
  - Is not possible the activation of processes or non-query operations within OCL
- The type of an OCL expression is the type of the result value
- An OCL constraint is a Boolean OCL expression

OCL Types

- Basic types
  - Real
  - Integer
  - String
  - Boolean
- Collection types
  - They are the result of navigation through associations in an UML model
- User-defined model types
  - All classes, types and interfaces in an UML model

Operations Defined for Every OCL Type

- Two model objects can be compared
  - o1 = o2, o1 <> o2
- The type of an object can be checked
  - oclIsTypeOf(type)
    - Returns true only for the instances of type
  - oclIsKindOf(type)
    - Returns true for the instances of type and of its subtypes
- The type of an object can be retrieved
  - oclType

Basic Types

- Real
  - r1 + r2, r1 – r2, r1 * r2, r1 / r2, r.abs, r.floor, r.round, r1.max(r2), r1.min(r2)
  - r1 = r2, r1 <> r2 , r1 < r2, r1 > r2, r1 <= r2, r1 >= r2
- Integer
  - i1 + i2, i1 – i2, i1 * i2, i1.div(i2), i1.mod(i2), i1 / i2, i.abs, i1.max(i2), i1.min(i2)
  - i1 = i2, i1 <> i2 , i1 < i2, i1 > i2, i1 <= i2, i1 >= i2
- Note that Integer is a subclass of Real
  - For each parameter of type Real, an Integer can be used as the actual parameter

Basic Types

- String
  - s.size, s.substring(2, 3), s1.concat(s2), s.toInteger, s.toReal
  - s1 = s2, s1 <> s2
  - Note that character positions run from **1** to **s.size**
- Boolean
  - b1 = b2, b1 <> b2, b1 or b2, b1 xor b2, b1 and b2, not b, b1 implies b2

Collections

- Collection is an **abstract** predefined OCL type
- Real collections are defined through its subtypes:
  - **Set**: is the mathematical set, that is, it does not contain duplicate elements
  - **OrderedSet**: is a Set where the elements are ordered
  - **Bag**: is like a set, but may contain duplicates
  - **Sequence**: is like a Bag, but the elements are ordered

Operation on Collections

- select(b), reject(b)
  - This results in a collection that contains all the elements from collection for which the boolean expression, b, is true / false
  - self.employee->select(age > 50)
  - self.employee->reject(age > 50)
- collect(e)
  - This results in a collection that contains the results of all the evaluations of the expression, e
  - self.employee->collect(person.birthDate)

Operation on Collections

- forAll(b)
  - This results in a Boolean that is true if the Boolean expression, b, is true for all elements of the collection
  - self.employee->forAll(age <= 65)
- exists(b)
  - This results in a Boolean that is true if the Boolean expression, b, is true for at least one element of the collection
  - self.employee->exists(age <= 65)

Operation on Collections

- Set, Bag, OrderedSet and Sequence
  - size(), count(o), sum()
  - c1 = c2, includes(o), excludes(o), includesAll(c), excludesAll(c), isEmpty(), notEmpty()
- Set & Bag and OrderedSet & Sequence
  - union(c), intersection(c), c1 – c2
  - including(o), excluding(o)
- OrderedSet and Sequence
  - append(o), prepend(o), insertAt(i, o)
  - at(i), indexOf(o), first(), last()
  - subOrderedSet(i1, i2), subSequence(i1, i2)

Model Types and Properties

- Model types are classes, interfaces and types used / defined in an UML model
- Properties of a model type are:
  - Attributes
  - Operations and methods
  - Navigations that are derived from the associations
  - Enumerations defined as attribute types
- Properties of a model type can be referenced in OCL expressions

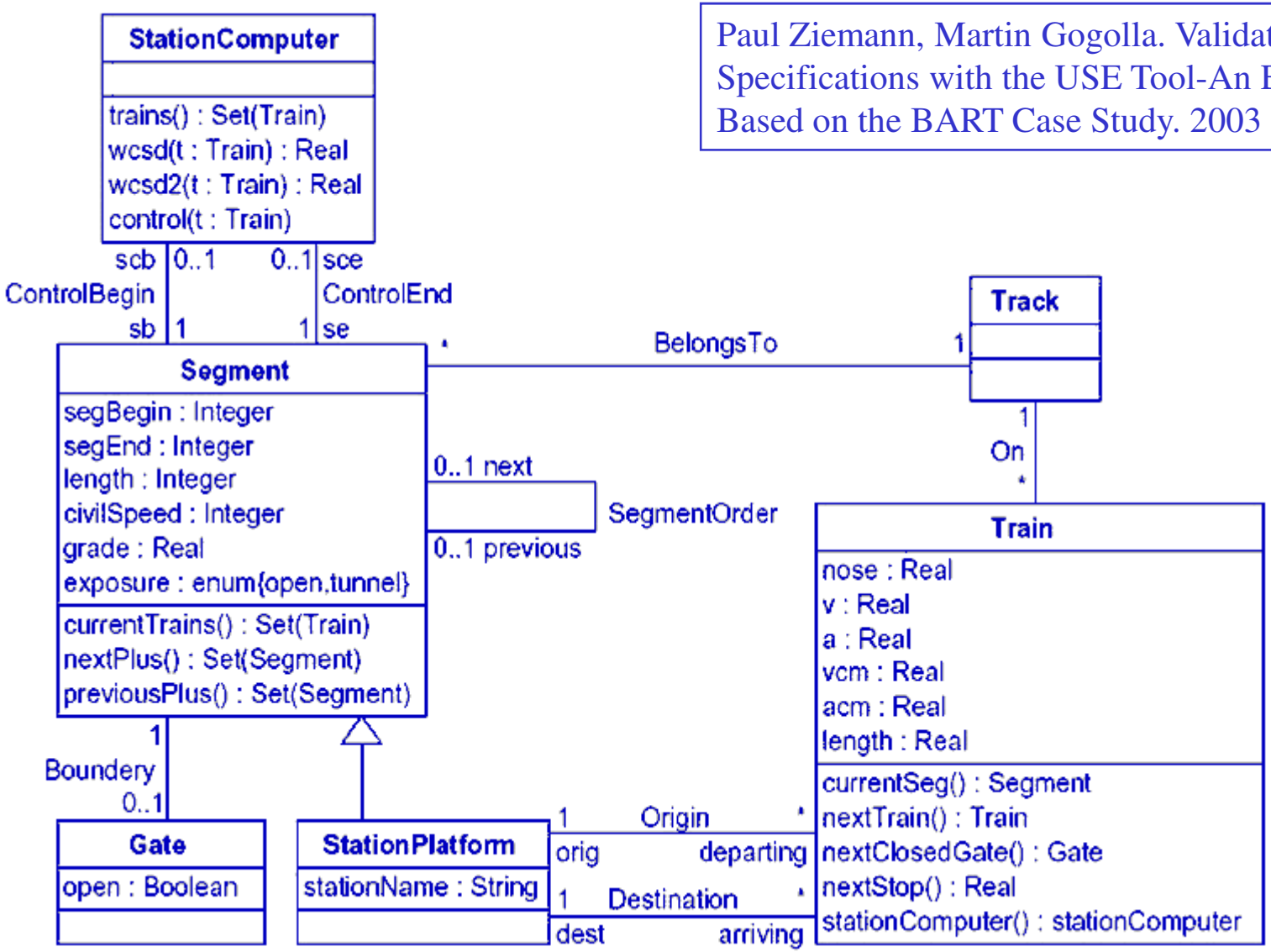
Enumerations

- An enumeration defines a number of enumeration literals, that are the possible values of the enumeration
  - Enumerations are types in UML and have a name
  - Within OCL one can refer to the value of an enumeration
- If in the UML model there is an enumeration named Gender with values 'female' or 'male‘, in OCL they can be referred as follows:
  - Gender::male
  - Gender::female

AOT LAB

Automatic Train Control System

Paul Ziemann, Martin Gogolla. Validating OCL Specifications with the USE Tool-An Example Based on the BART Case Study. 2003



Segment Constraints

- If for a segment there is a next segment, the begin of the next is equal to the end of the former

  self.next.isDefined **implies** self.next.segBegin = self.segEnd

- The length of a segment has to be the difference of segment end and begin

  self.segEnd - self.segBegin = self.length

- Connected segments belong to the same track

  self.next.isDefined implies self.track = self.next.track

Train Constraints

- The origin and the destination of a train have to be connected by a sequence of segments

```
self.orig.nextPlus()->includes(self.dest)
```

- A train should stay below the maximum speed that segment of track can handle

```
self.trains()->forAll(t | t.v <=t.currentSeg().civilSpeed)
```

Train Constraints

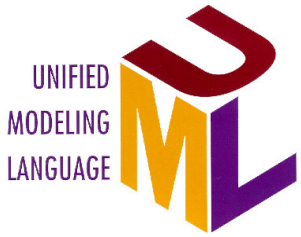
- A train should not enter a closed gate

```
self.trains()->forAll(t | t.nextClosedGate().isDefined
  implies
t.nose+self.wcsd(t) < t.nextClosedGate().segment.segEnd)
```

- A train should never get so close to a train in front that if the train in front stopped suddenly (e.g., derailed) the (following) train would hit it

```
self.trains()->forAll(t | t.nextTrain().isDefined
  implies
t.nose+self.wcsd(t) < t.nextTrain().nose - t.nextTrain().length)
```

AOT LAB



Unified Modeling Language

Extending UML

Why Extending UML?

- UML does not contain domain specific concepts, but contains general purpose concepts for system modeling
- However, UML provides mechanisms to extend itself
  - These mechanisms allow to widen the UML meta model in order to embed concepts of a specific domain
  - Building UML models for that domain will be much easier within the extended UML meta model

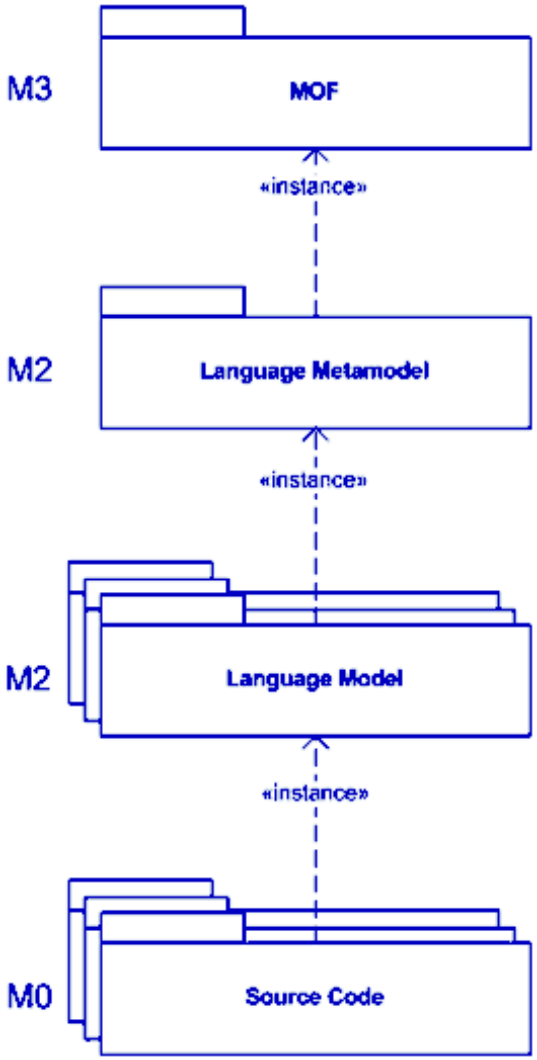
What is a Meta Model?

- Meta modeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for the modeling in a predefined class of problems
- A meta model maintains the precise definition of the constructs and rules needed for creating a set of related models
- A meta model is a model that explains a set of related models

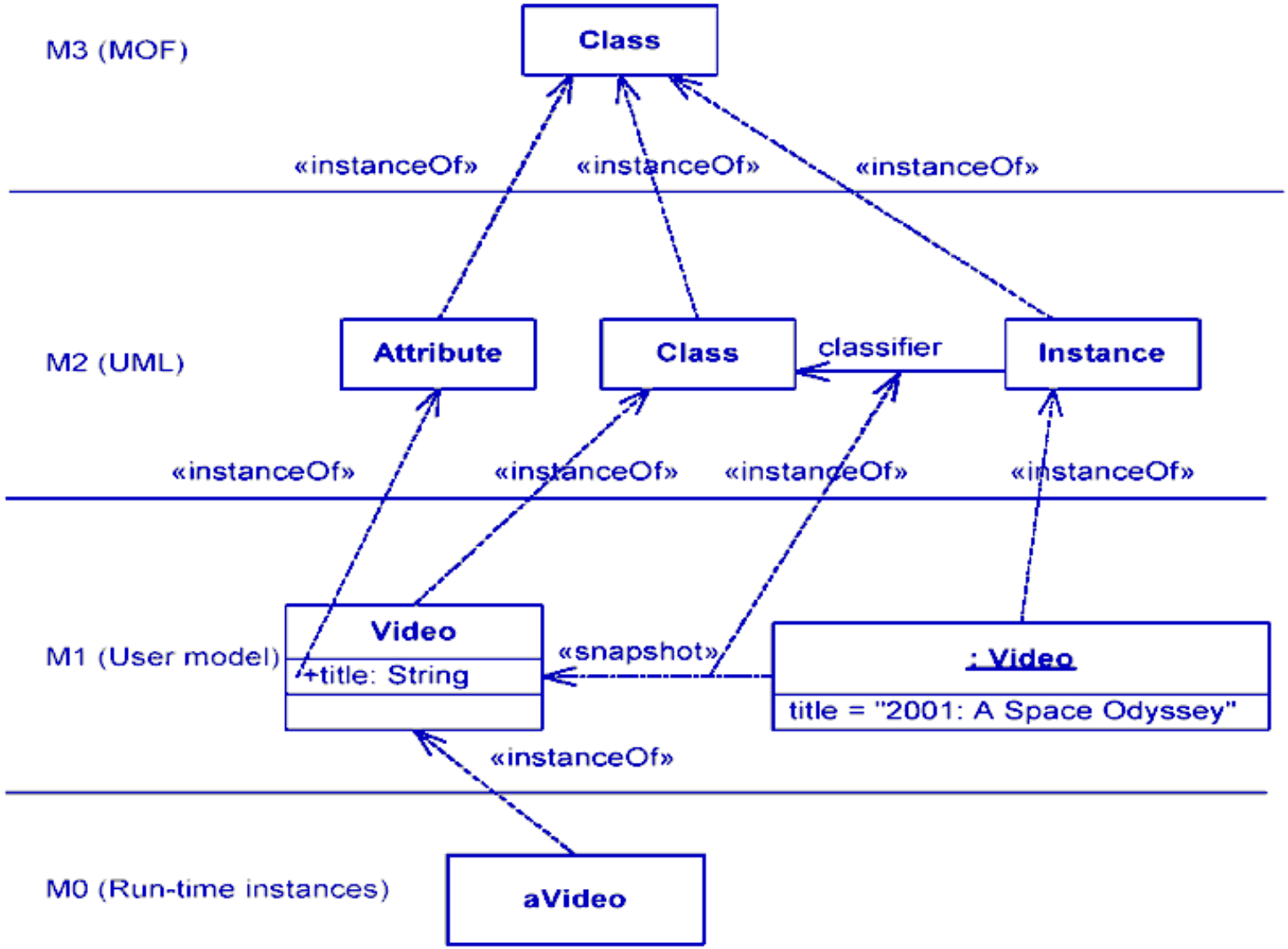
What is a Meta Data?

- The term meta data is used to refer data whose purpose is to describe other data
- A meta data is the information about a particular data set which may describe, for example, how, when, and by whom it was received, created, accessed, and/or modified and how it is formatted
  - Some meta data, such as file dates and sizes, can easily be seen by users
  - Other meta data can be hidden or embedded and unavailable to computer users who are not technically adept
  - A meta data is generally not reproduced in full form when a document is printed

UML Meta-Model Hierarchy



UML Meta-Model Hierarchy

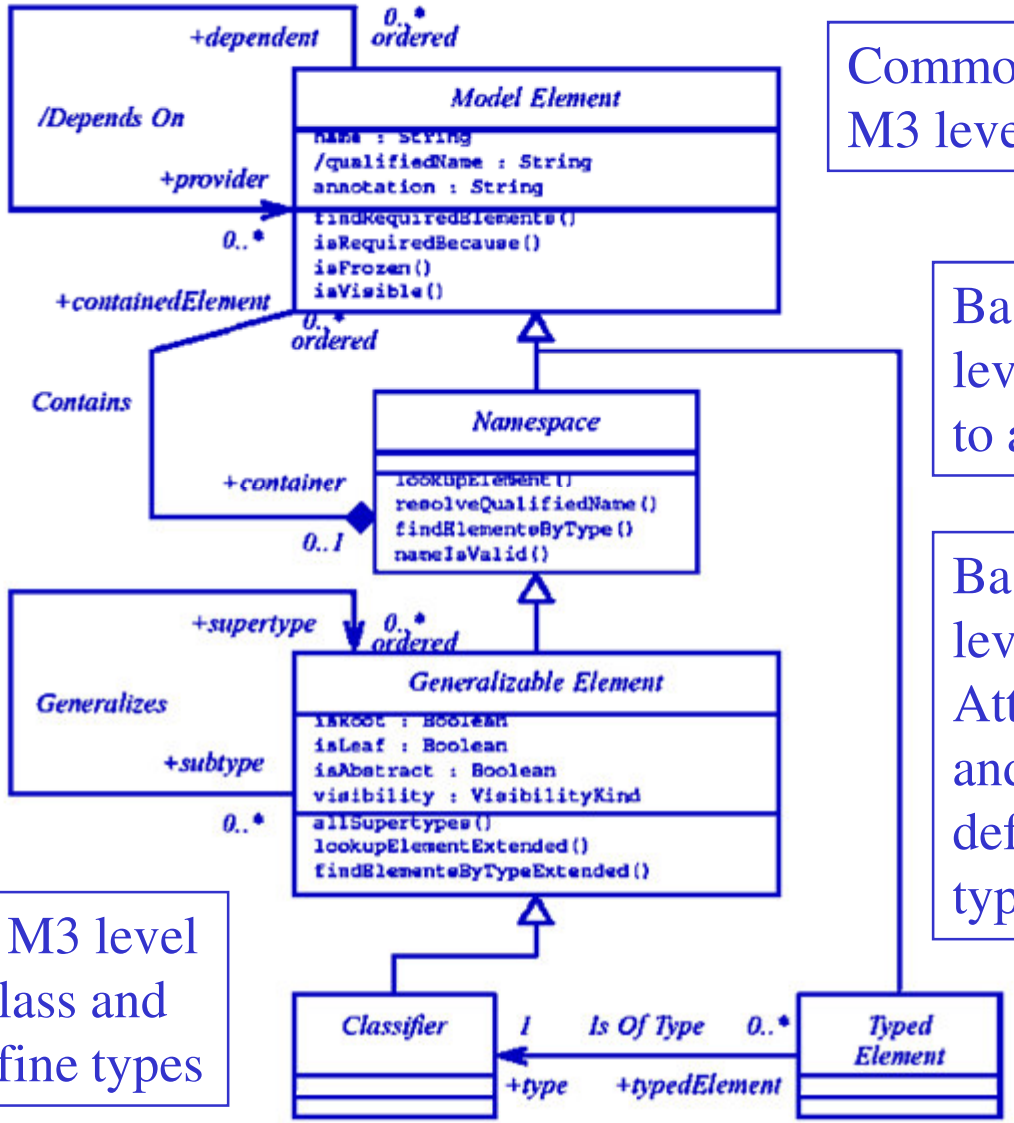


What is MOF?

- The Meta Object Facility (MOF) provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems
- MOF provides very simple rules for modeling metadata
  - Meta models are defined on the basis of on simple class modeling concepts and using UML class modeling notation
- MOF decouples the modeling concepts from the desirable meta data services
  - Meta data interchange, Reflection, Federation, Life Cycle, Versioning, Identity, Queries, …

AOT LAB

Key Abstract Classes



Common base class of all M3 level classes

Base class for all M3 level classes that need to act as containers

Base class for all M3 level classes such as Attribute, Parameter, and Constant whose definition requires a type specification

Base class for all M3 level classes that support generalization

Base class for all M3 level classes such as Class and DataType that define types

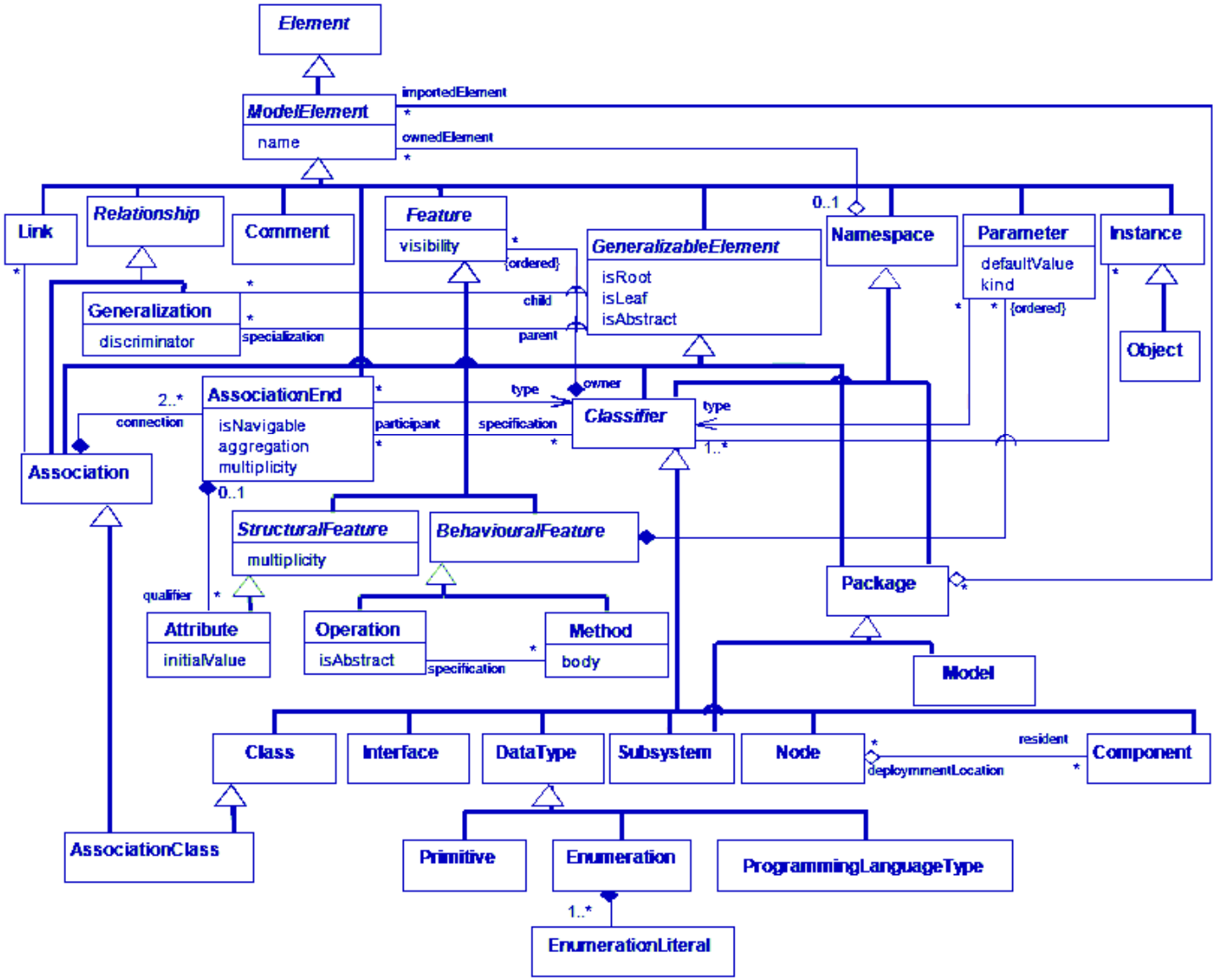
AOT LAB

Concrete Classes and Associations

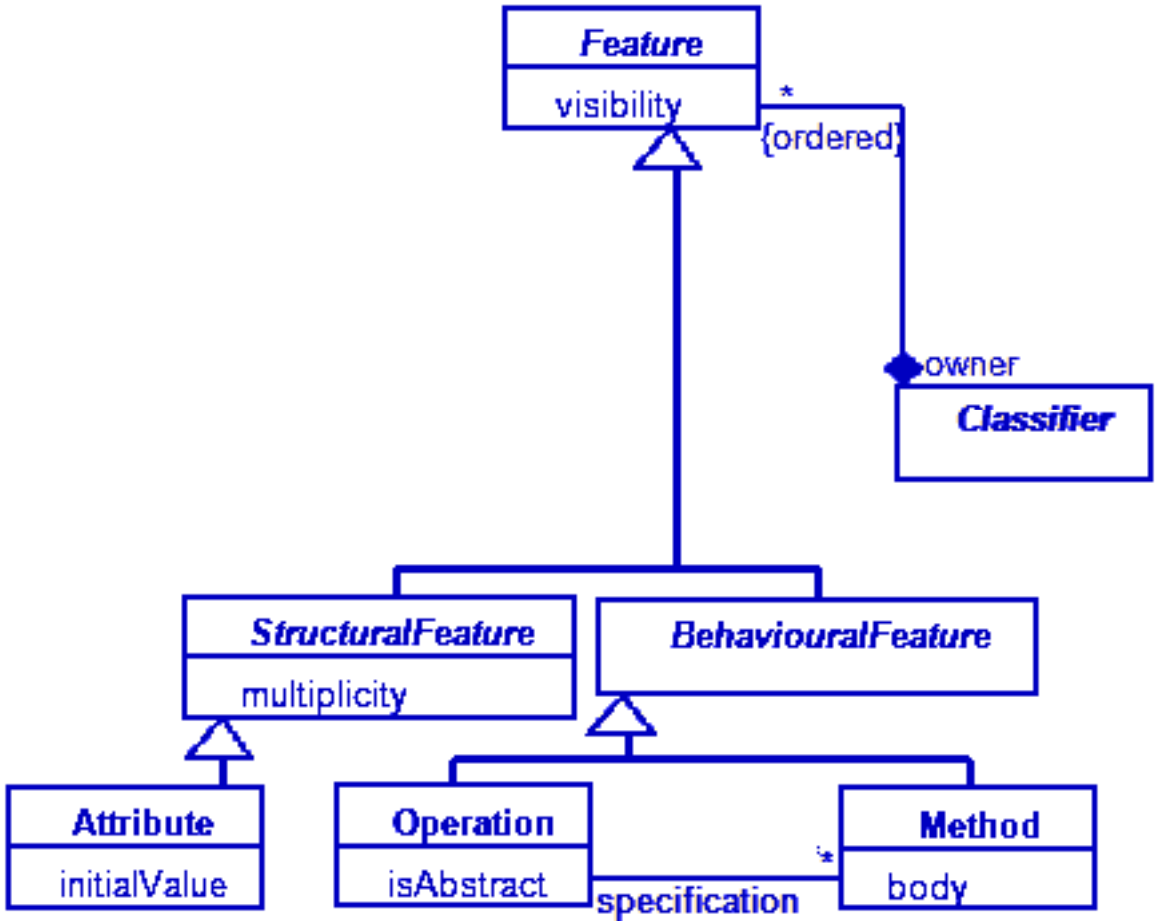
- Class
- Association
- Exception
- Attribute
- Constant
- Constraint
- Contains
- Generalizes
- IsOfType
- DependsOn

AOT LAB

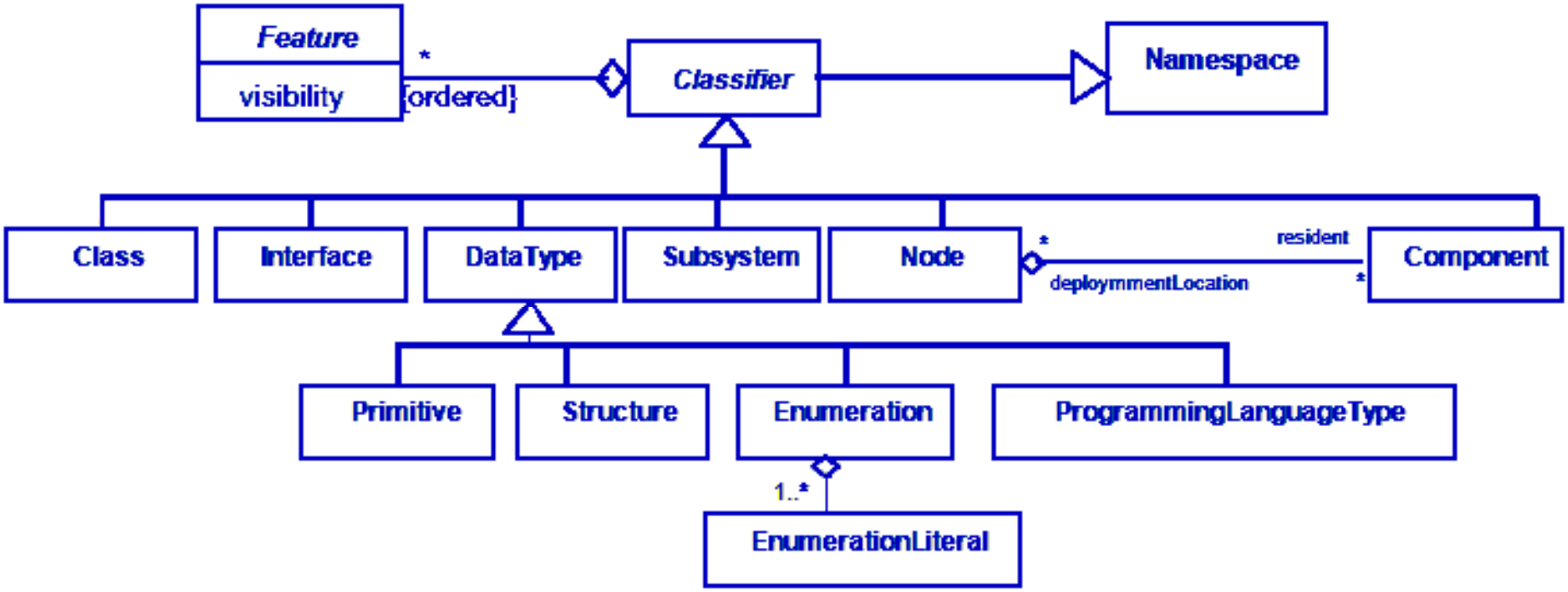
UML Meta Model



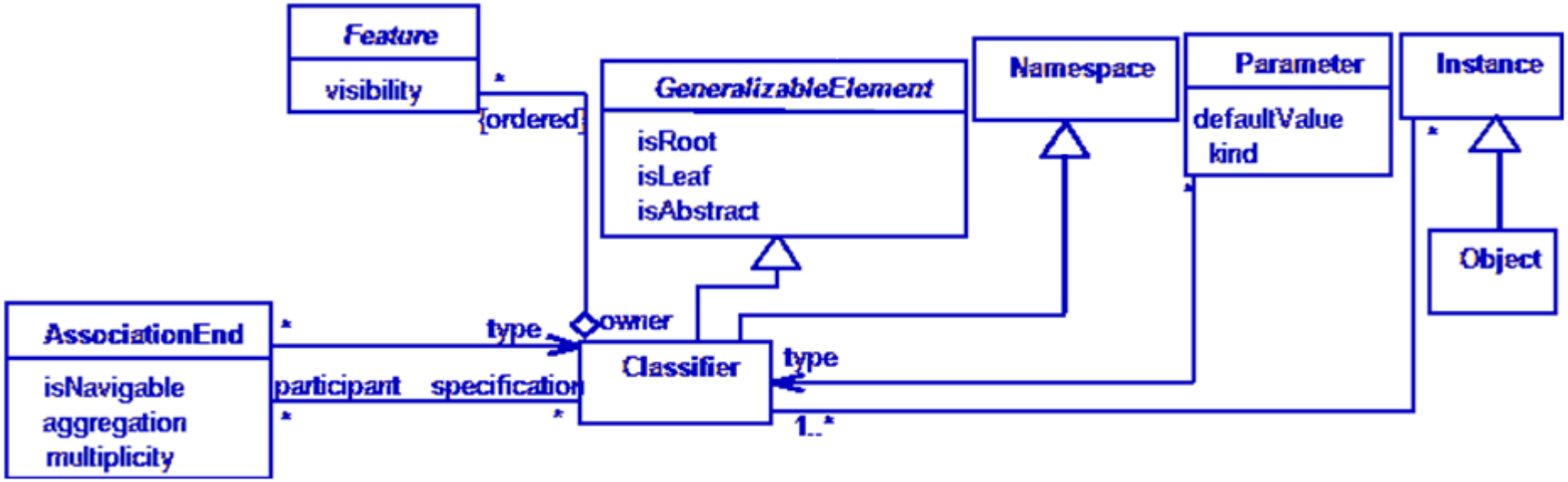
Feature


Feature
visibility
*
{ordered}
owner
Classifier
StructuralFeature
multiplicity
BehaviouralFeature
Attribute
initialValue
Operation
isAbstract
specification
*
Method
body

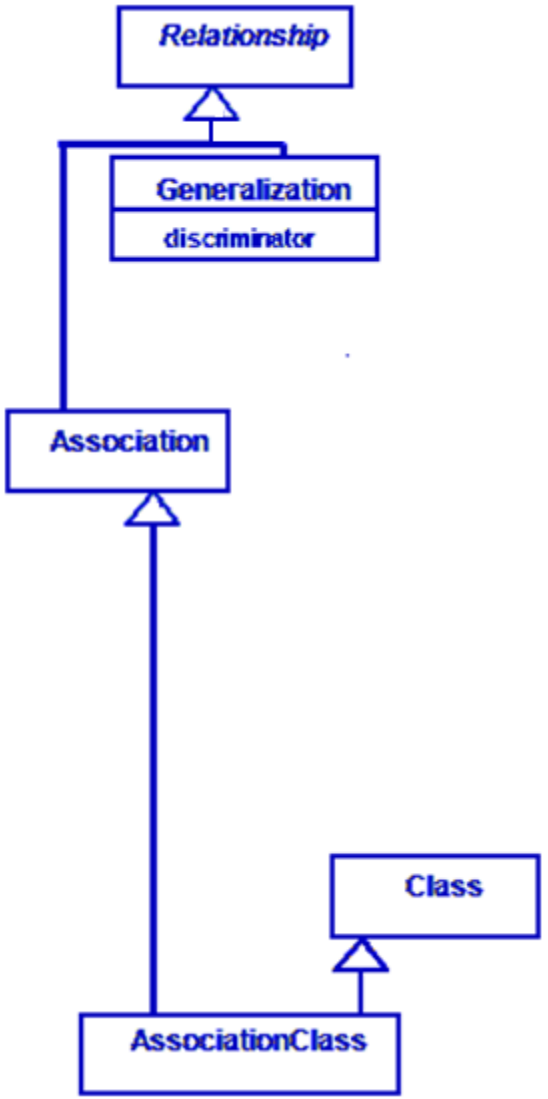
Classifier


Feature
visibility
*
{ordered}
Classifier
Namespace
Class
Interface
DataType
Subsystem
Node
*
resident
deploymmentLocation
*
Component
Primitive
Structure
Enumeration
ProgrammingLanguageType
1..*
EnumerationLiteral

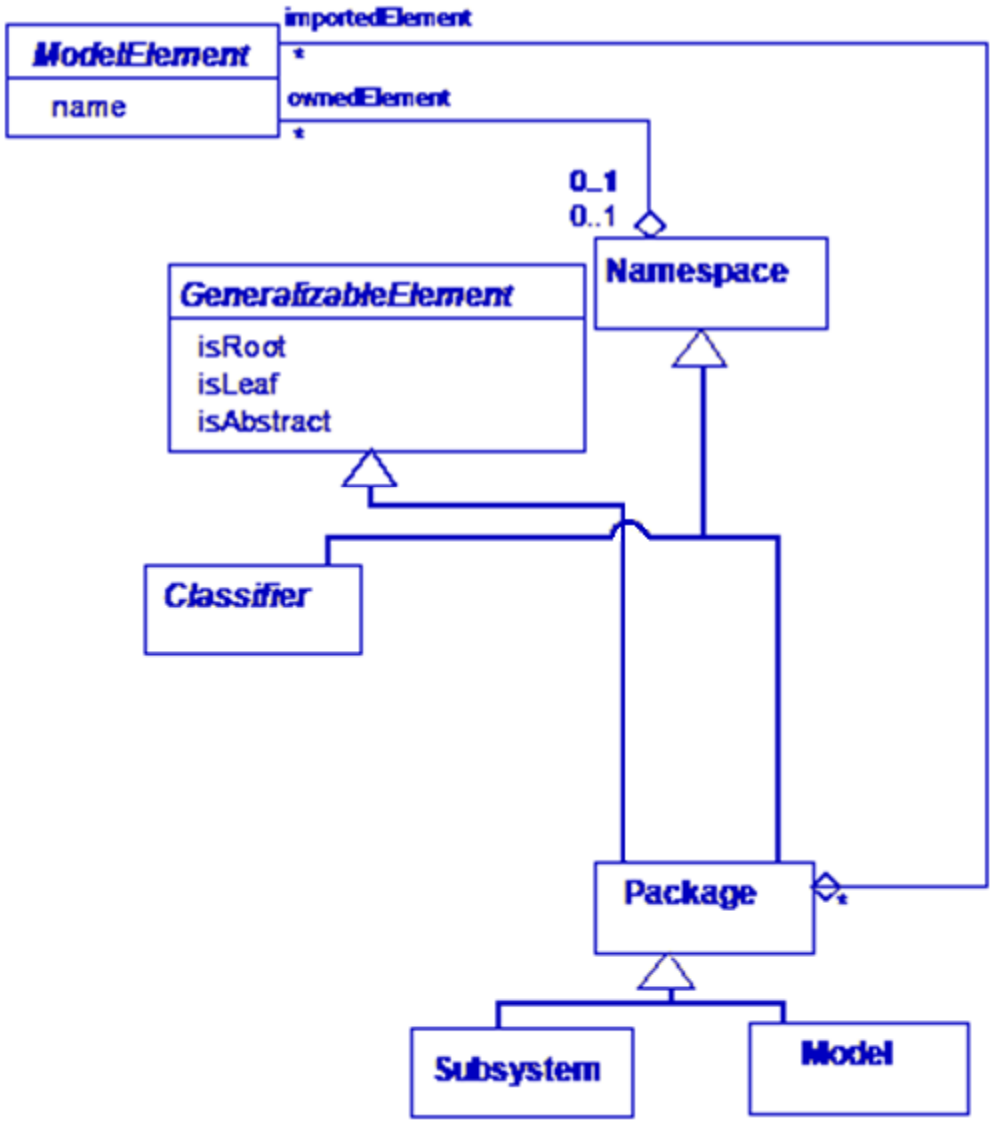
Classifier


Feature
visibility
{ordered}
GeneralizableElement
isRoot
isLeaf
isAbstract
Namespace
Parameter
defaultValue
kind
Instance
Object
AssociationEnd
isNavigable
aggregation
multiplicity
participant
specification
type
owner
Classifier
type
1..*

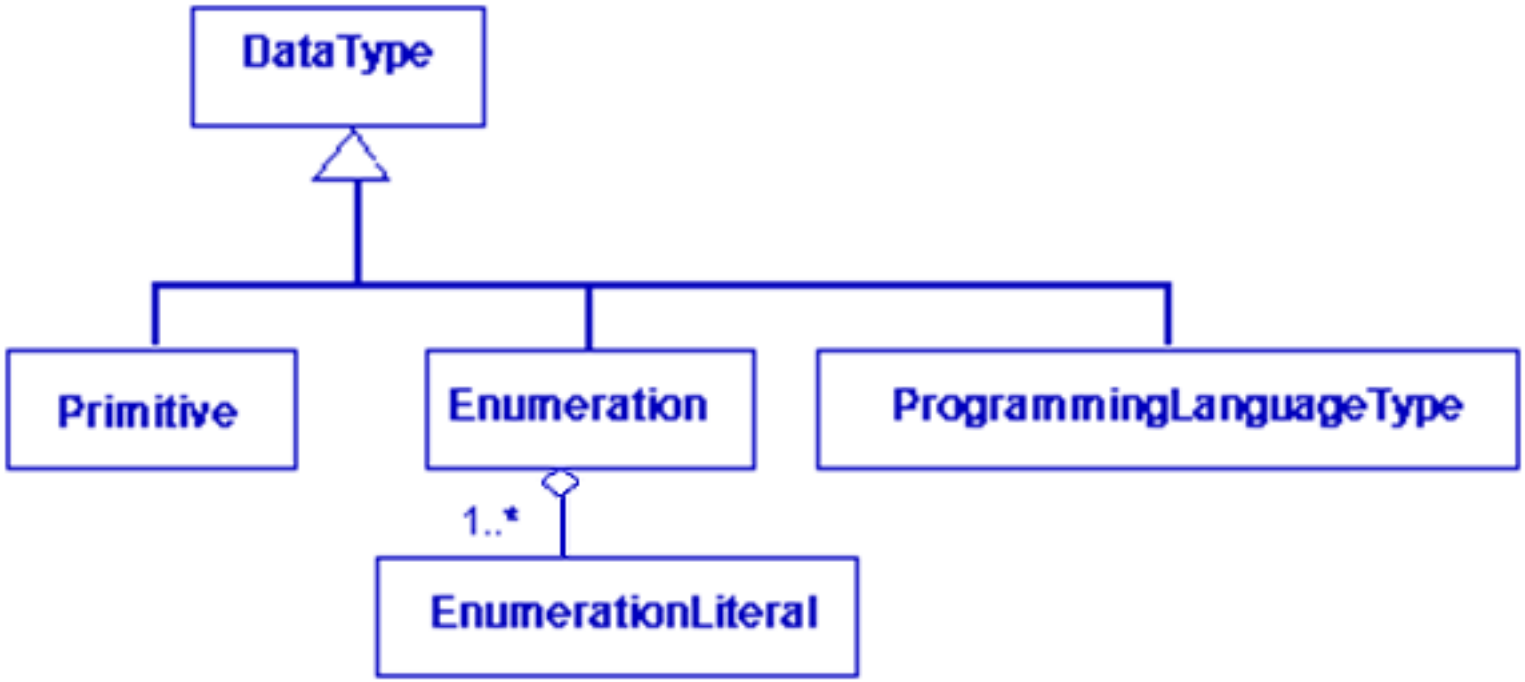
Relationships



Namespace

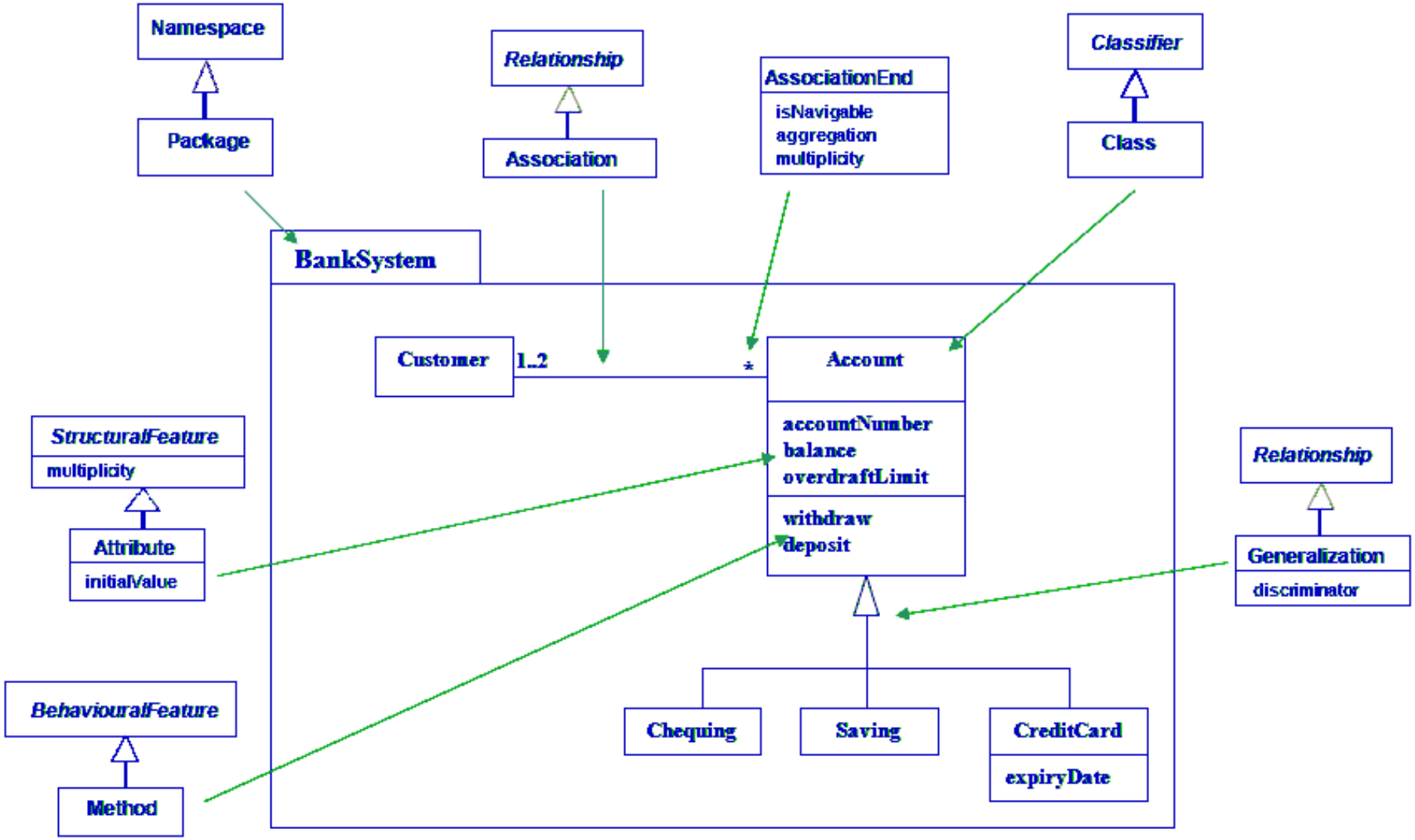

importedElement
ModelElement
name
*
ownedElement
*
0..1
0..1
Namespace
GeneralizableElement
isRoot
isLeaf
isAbstract
Classifier
Package
*
Subsystem
Model

Data Types


DataType
Primitive
Enumeration
ProgrammingLanguageType
1..*
EnumerationLiteral

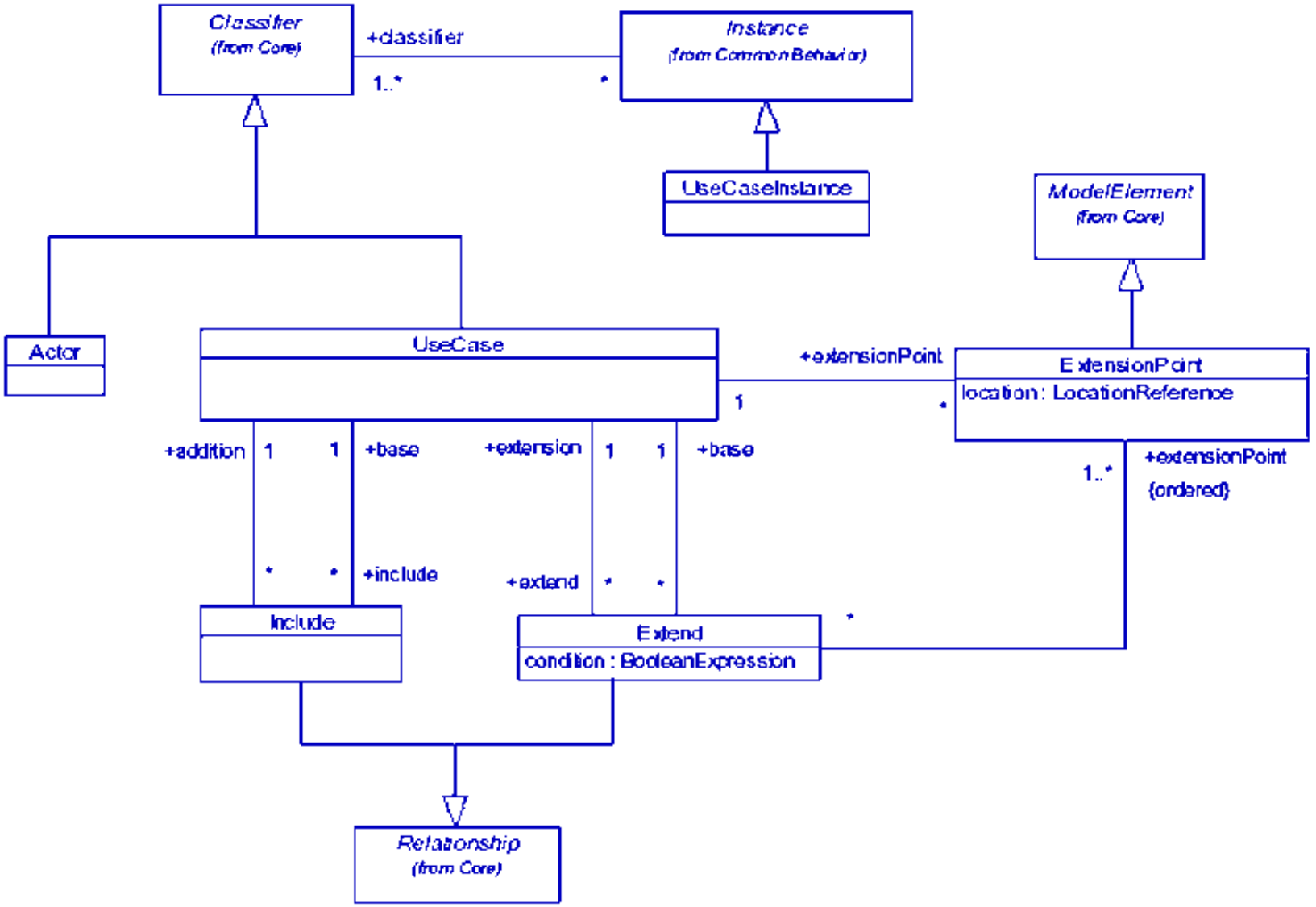
AOT LAB

Class Meta Model – Model Mapping



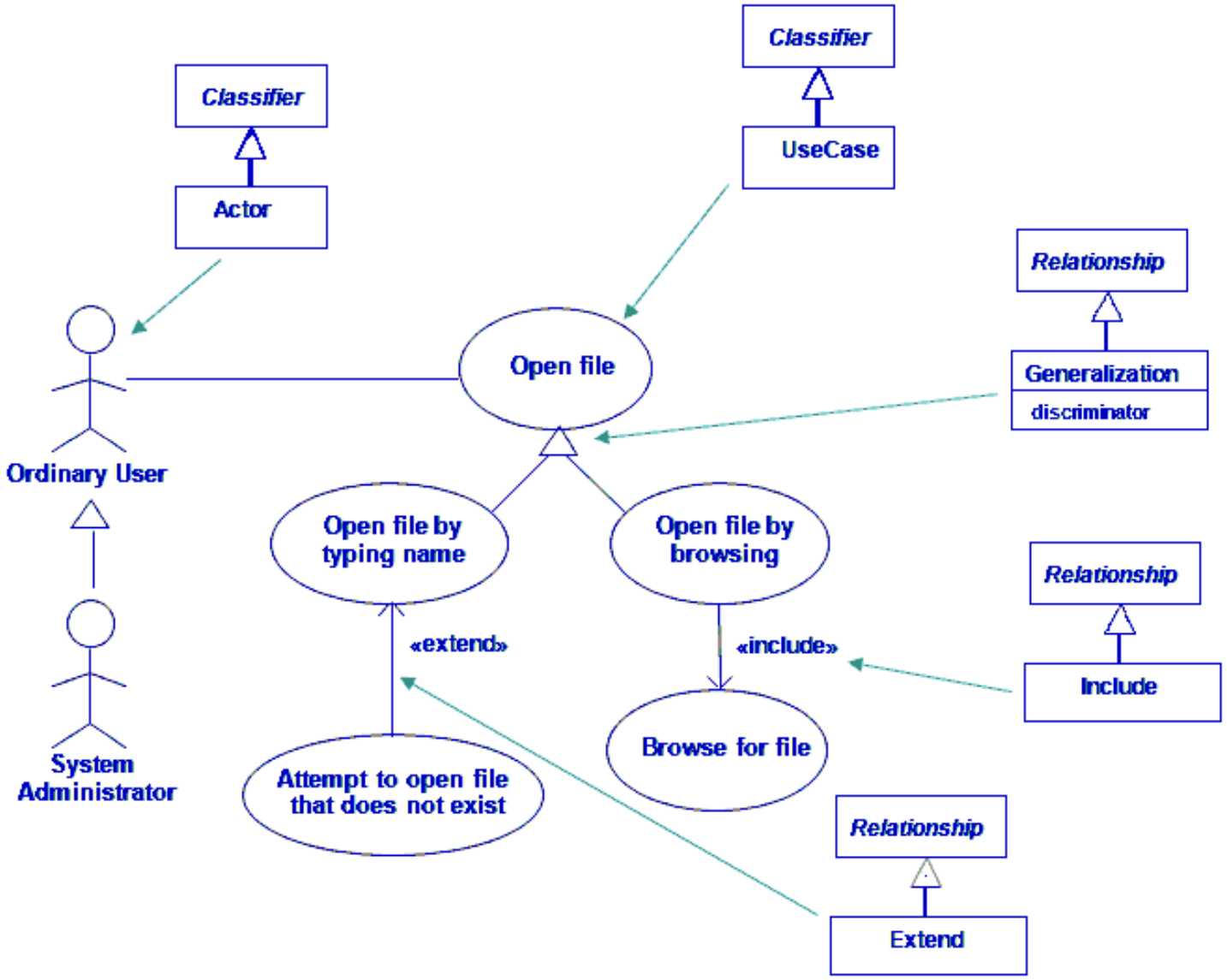
AOT LAB

Use Case Meta Model



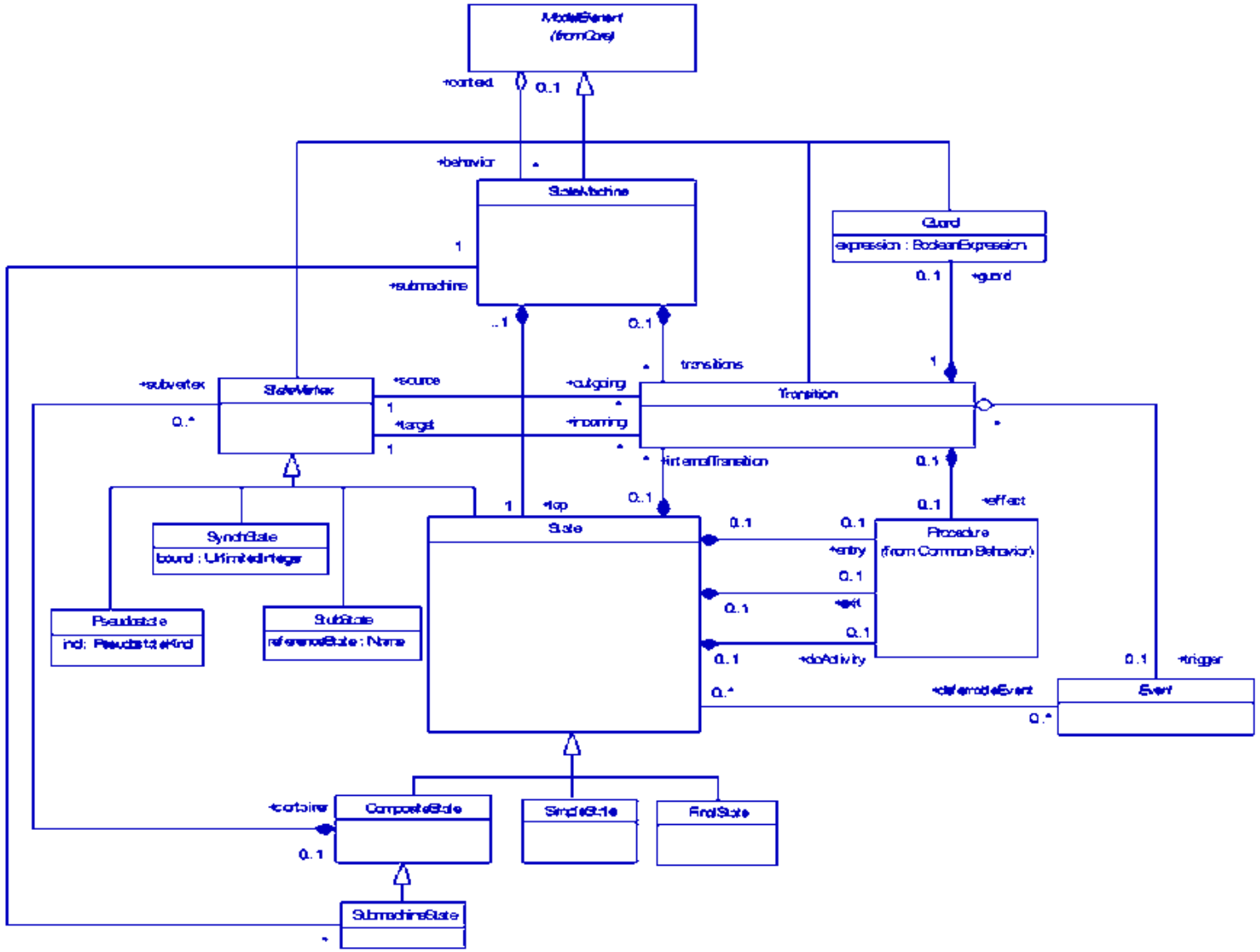
AOT LAB

Use Case Meta Model – Model Mapping

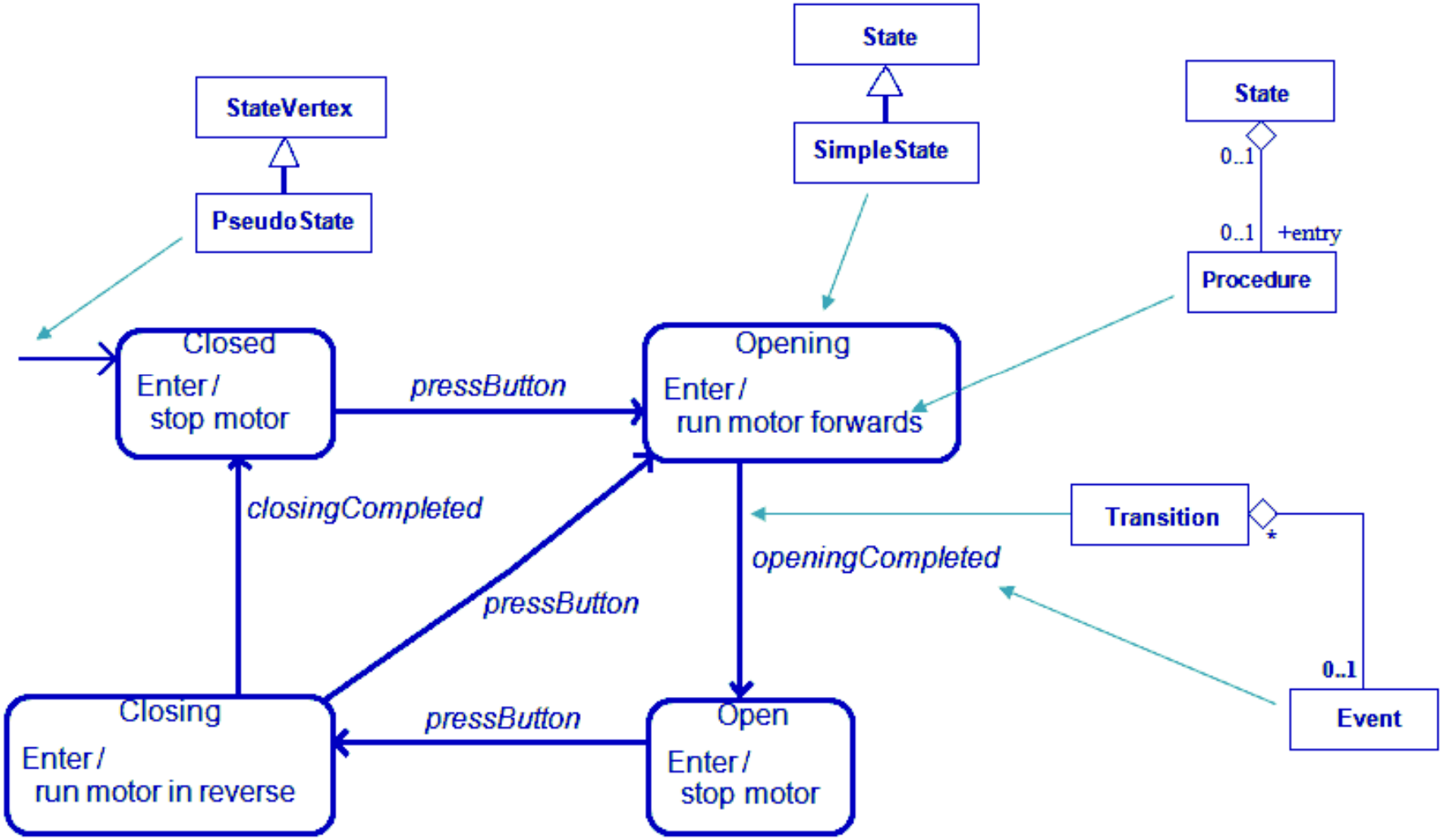


AOT LAB

State Meta Model



State Meta Model – Model Mapping

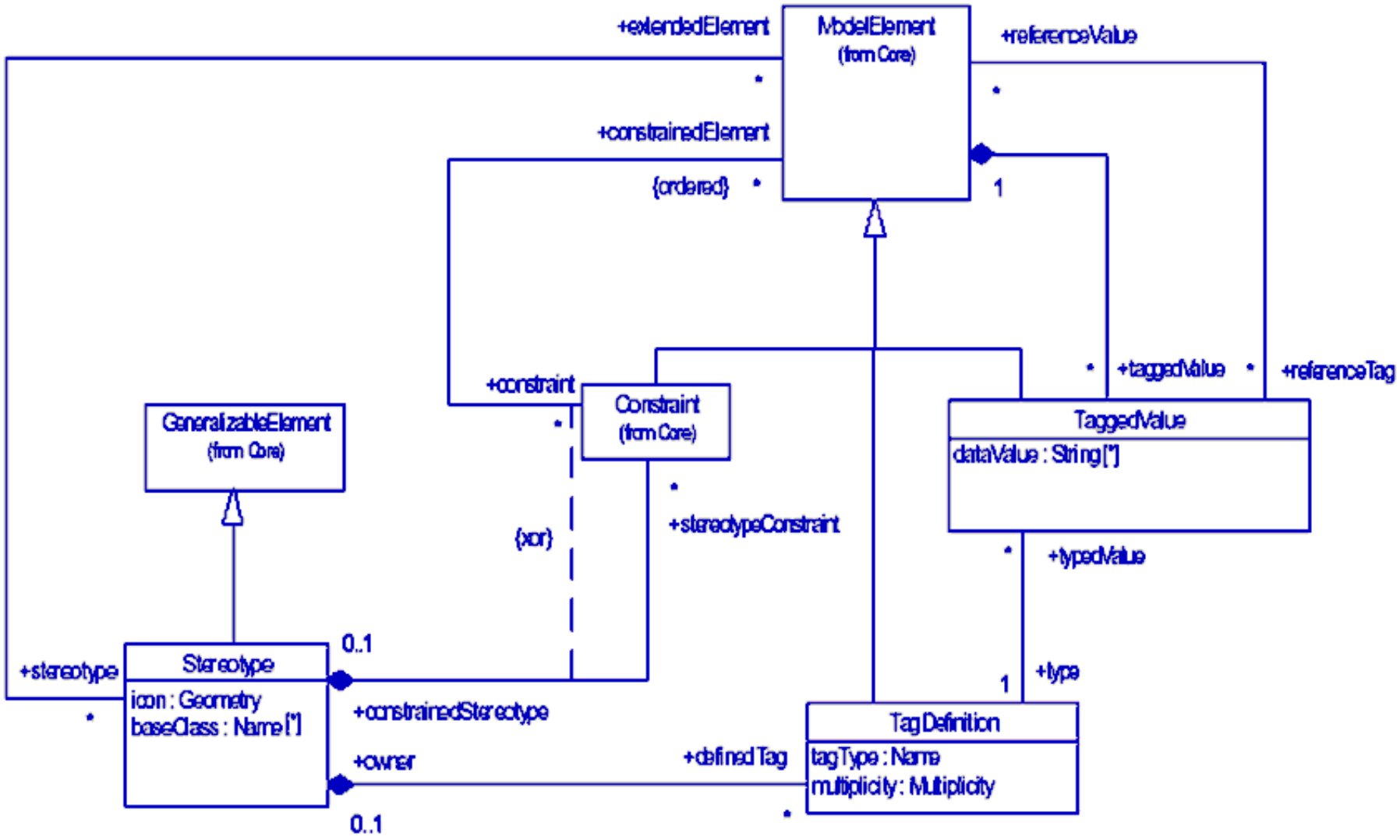


UML Extensibility

- A new dialect of UML can be defined by using **profiles** to customize the language for particular platforms and domains (lightweight extension)
  - Compliant with UML standard
- A new language can be specified at the MOF level by adding new meta classes and meta relationships (heavyweight extension)
  - Not compliant with UML standard

AOT LAB

UML Extension Mechanisms



Stereotype

- A stereotype denotes a variation on an existing UML modeling element with the same structure, but with a difference in meaning
  - A stereotypes extends the UML vocabulary
- A stereotype can be used to further define additional constraints and tag definitions of a model element in a UML diagram
- A stereotype can vary the semantics of an existing element (it is not a new model element per se)

Stereotype

- Augment UML classification mechanism based on built-in UML meta model class hierarchy
- Adds "virtual" UML meta classes with new:
  - Semantics
  - Meta-attributes
  - Property lists
  - Constraints
  - Graphical representation

Tagged Value

- A tagged value allow the association of extra information with a modeling element
- A tagged value is a property used for specifying a keyword-value pair of a model element where the keyword is an attributes
- A tagged value is not a class attribute, but can considered a metadata, since its value applies to the element itself and not to its instances
- One of the most common uses of a tagged value is to specify properties that are relevant to code generation or configuration management

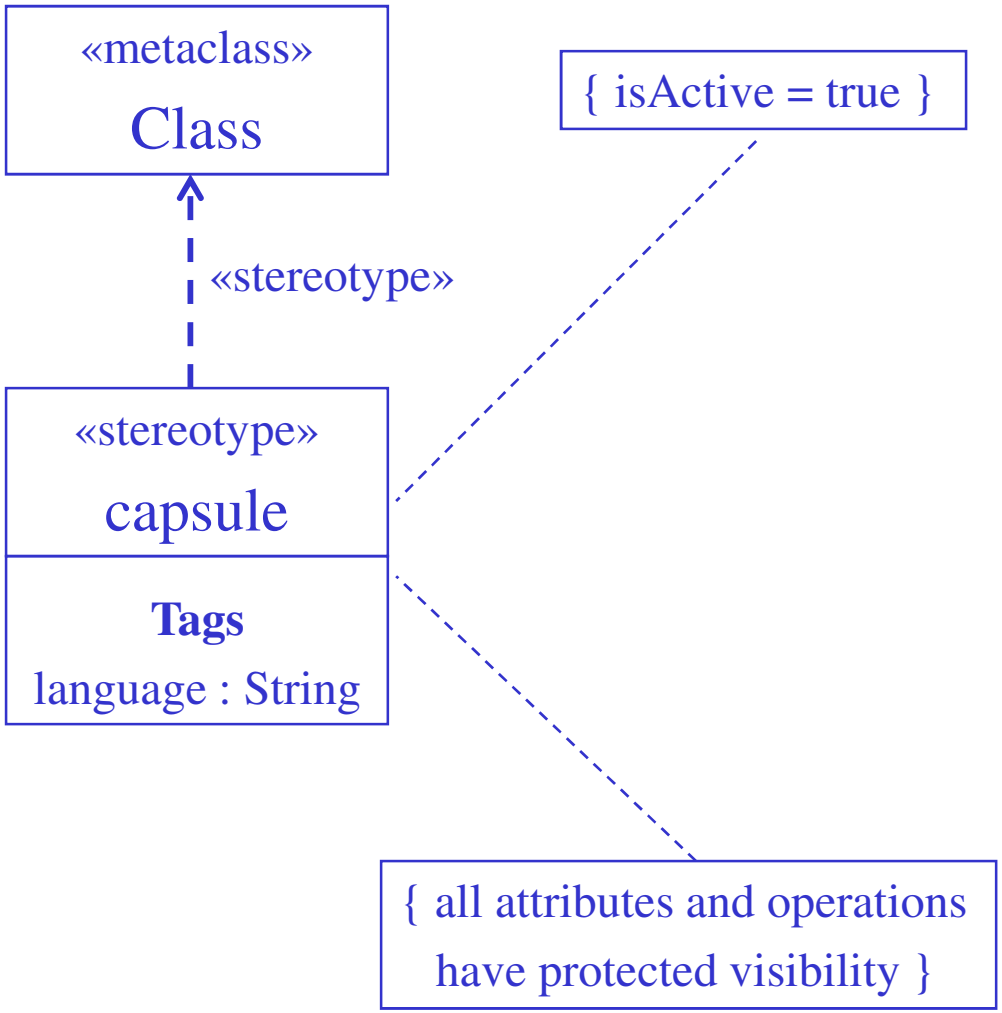
Constraint

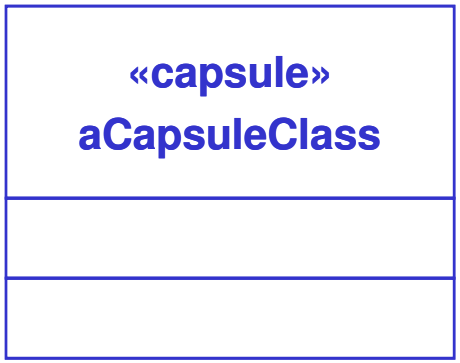
- A constraint allows the specification of the semantics and/or of the condition that must be held true at all times for the elements of a model
- A constraint allows the extension of the semantics of a UML building block by adding new rules, or modifying existing ones
- A constraint can be expressed by an OCL invariant, precondition or postcondition

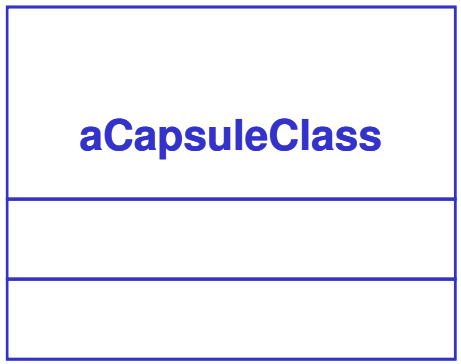
Capsule

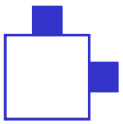
- A capsule is a special type of concurrent object used in modeling certain real-time systems
- By definition, all classes of this type:
  - Are active (concurrent)
  - Have only features (attributes and operations) with protected visibility
  - Have a special “language” characteristic used for code generation purposes

Stereotype


«metaclass»
Class
«stereotype»
«stereotype»
capsule
Tags
language : String
{ isActive = true }
{ all attributes and operations
have protected visibility }


«capsule»
aCapsuleClass


aCapsuleClass

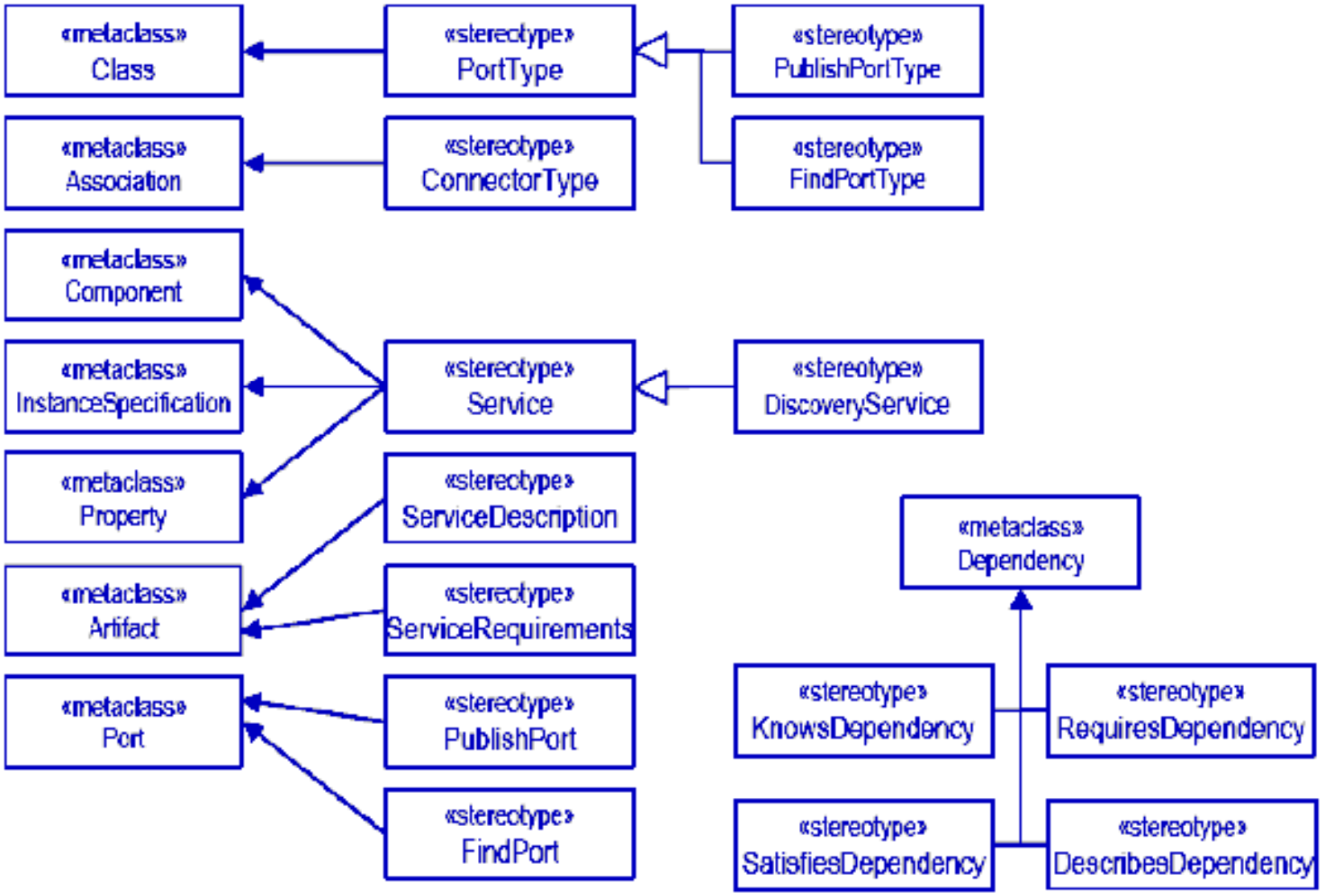


Profile

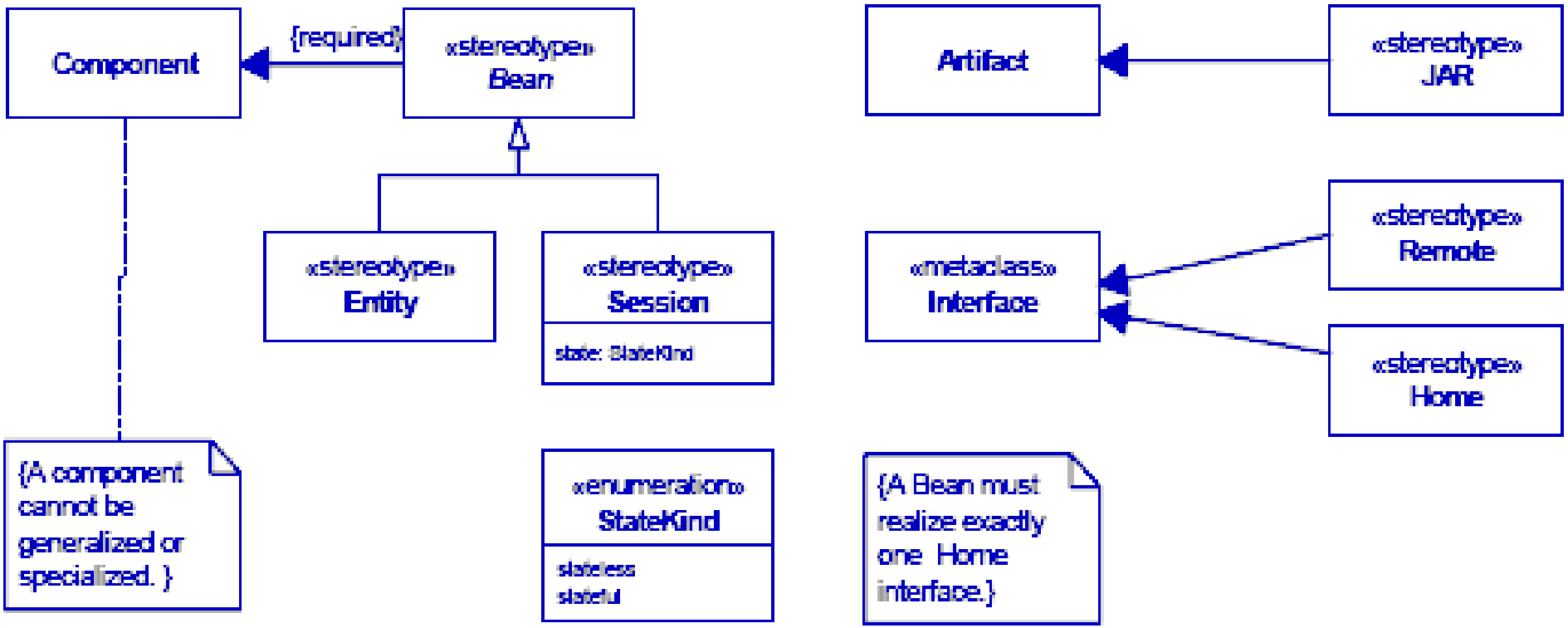
- A profile groups a set of extension mechanisms for building UML models in a particular domain

- A profile consists of a package that contains a collection of stereotypes, tag definitions and constraints that work together to define new semantics for a model

SOA Profile


«metaclass»
Class
«stereotype»
PortType
«stereotype»
PublishPortType
«metaclass»
Association
«stereotype»
ConnectorType
«stereotype»
FindPortType
«metaclass»
Component
«metaclass»
InstanceSpecification
«stereotype»
Service
«stereotype»
DiscoveryService
«metaclass»
Property
«stereotype»
ServiceDescription
«metaclass»
Dependency
«metaclass»
Artifact
«stereotype»
ServiceRequirements
«stereotype»
KnowsDependency
«stereotype»
RequiresDependency
«metaclass»
Port
«stereotype»
PublishPort
«stereotype»
SatisfiesDependency
«stereotype»
DescribesDependency
«stereotype»
FindPort

EJB Profile


Component
{required}
«stereotype»
Bean
«stereotype»
Entity
«stereotype»
Session
state: StateKind
{A component cannot be generalized or specialized. }
«enumeration»
StateKind
stateless
stateful
Artifact
«stereotype»
JAR
«metaclass»
Interface
«stereotype»
Remote
«stereotype»
Home
{A Bean must realize exactly one Home interface.}