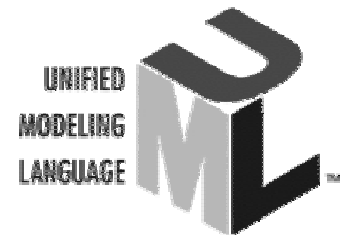


Object Modeling with OMG UML Tutorial Series

# Introduction to UML: Structural and Use Case Modeling

Cris Kobryn

Co-Chair UML Revision Task Force



© 1999-2001 OMG and Contributors: Crossmeta, EDS, IBM, Enea Data, Hewlett-Packard, IntelliCorp, Kabira Technologies, Klasse Objecten, Rational Software, Telelogic, Unisys



# Overview

---

- Tutorial series
- Quick tour
- Structural modeling
- Use case modeling



# Tutorial Series

---

- Lecture 1: Introduction to UML: Structural and Use Case Modeling
- Lecture 2: Behavioral Modeling with UML
- Lecture 3: Advanced Modeling with UML

[Note: This version of the tutorial series is based on *OMG UML Specification* v. 1.4, UML Revision Task Force recommended final draft, OMG doc# ad/01-02-13.]



# Tutorial Goals

---

## ■ What you will learn:

- what the UML is and what is it not
- UML's basic constructs, rules and diagram techniques
- how the UML can model large, complex systems
- how the UML can specify systems in an implementation-independent manner
- how UML, XMI and MOF can facilitate metadata integration

## ■ What you will not learn:

- Object Modeling 101
- object methods or processes
- Metamodeling 101



# Quick Tour

---

- Why do we model?
- What is the UML?
- Foundation elements
- Unifying concepts
- Language architecture
- Relation to other OMG technologies



# Why do we model?

---

- Provide structure for problem solving
- Experiment to explore multiple solutions
- Furnish abstractions to manage complexity
- Reduce time-to-market for business problem solutions
- Decrease development costs
- Manage the risk of mistakes

# The Challenge



Tijuana “shantytown”:  
<http://www.macalester.edu/~jschatz/residential.html>

# The Vision



Fallingwater:  
<http://www.adelaide.net.au/~jpolias/FLW/Images/FallingWater.jpeg>



# Why do we model graphically?

---

- Graphics reveal data.
  - Edward Tufte  
*The Visual Display of Quantitative Information, 1983*
- 1 bitmap = 1 megaword.
  - Anonymous visual modeler



# Quick Tour

---

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documentingthe artifacts of software systems
- Added to the list of OMG adopted technologies in November 1997 as UML 1.1
- Most recent minor revision is UML 1.3, adopted in November 1999
- Next minor revision will be UML 1.4, planned to be adopted in Q2 2001
- Next major revision will be UML 2.0, planned to be completed in 2002

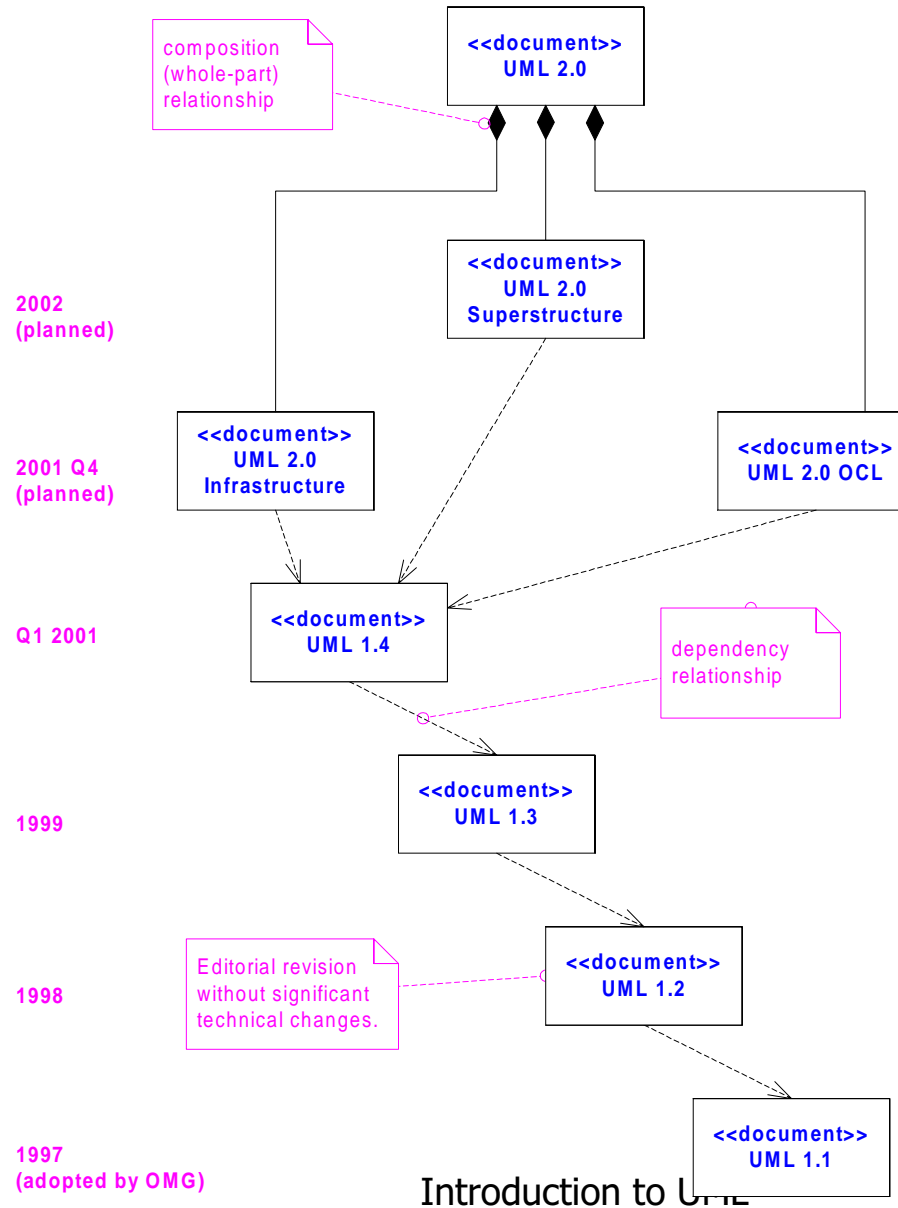


# UML Goals

---

- Define an easy-to-learn but semantically rich visual modeling language
- Unify the Booch, OMT, and Objectory modeling languages
- Include ideas from other modeling languages
- Incorporate industry best practices
- Address contemporary software development issues
  - scale, distribution, concurrency, executability, etc.
- Provide flexibility for applying different processes
- Enable model interchange and define repository interfaces

# OMG UML Evolution



From [Kobryn 01a].

Introduction to UML



# OMG UML Contributors

---

Aonix	Microsoft
Colorado State University	ObjecTime
Computer Associates	Oracle
Concept Five	Ptech
Data Access	OAQ Technology Solutions
EDS	Rational Software
Enea Data	Reich
Hewlett-Packard	SAP
IBM	Softeam
I-Logix	Sterling Software
InLine Software	Sun
Intellicorp	Taskon
Kabira Technologies	Telelogic
Klasse Objecten	Unisys
Lockheed Martin	...



# OMG UML 1.4 Specification

---

- UML Summary
- UML Semantics
- UML Notation Guide
- UML Example Profiles
  - Software Development Processes
  - Business Modeling
- Model Interchange
  - Model Interchange Using XMI
  - Model Interchange Using CORBA IDL
- Object Constraint Language



# Tutorial Focus: the Language

---

- language = syntax + semantics
  - syntax = rules by which language elements (e.g., words) are assembled into expressions (e.g., phrases, clauses)
  - semantics = rules by which syntactic expressions are assigned meanings
- *UML Notation Guide* – defines UML's graphic syntax
- *UML Semantics* – defines UML's semantics



# Foundation Concepts

---

- Building blocks
- Well-formedness rules

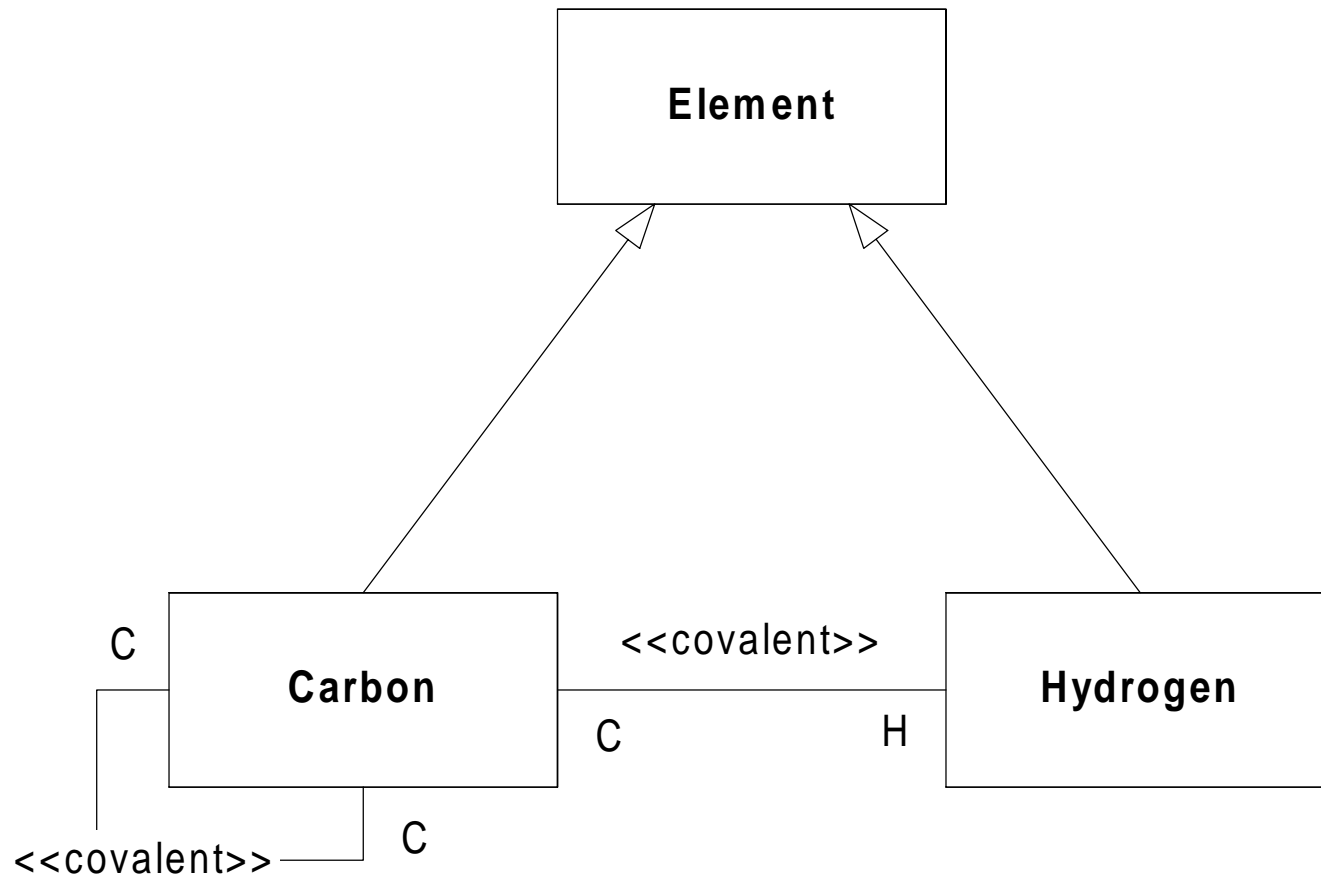


# Building Blocks

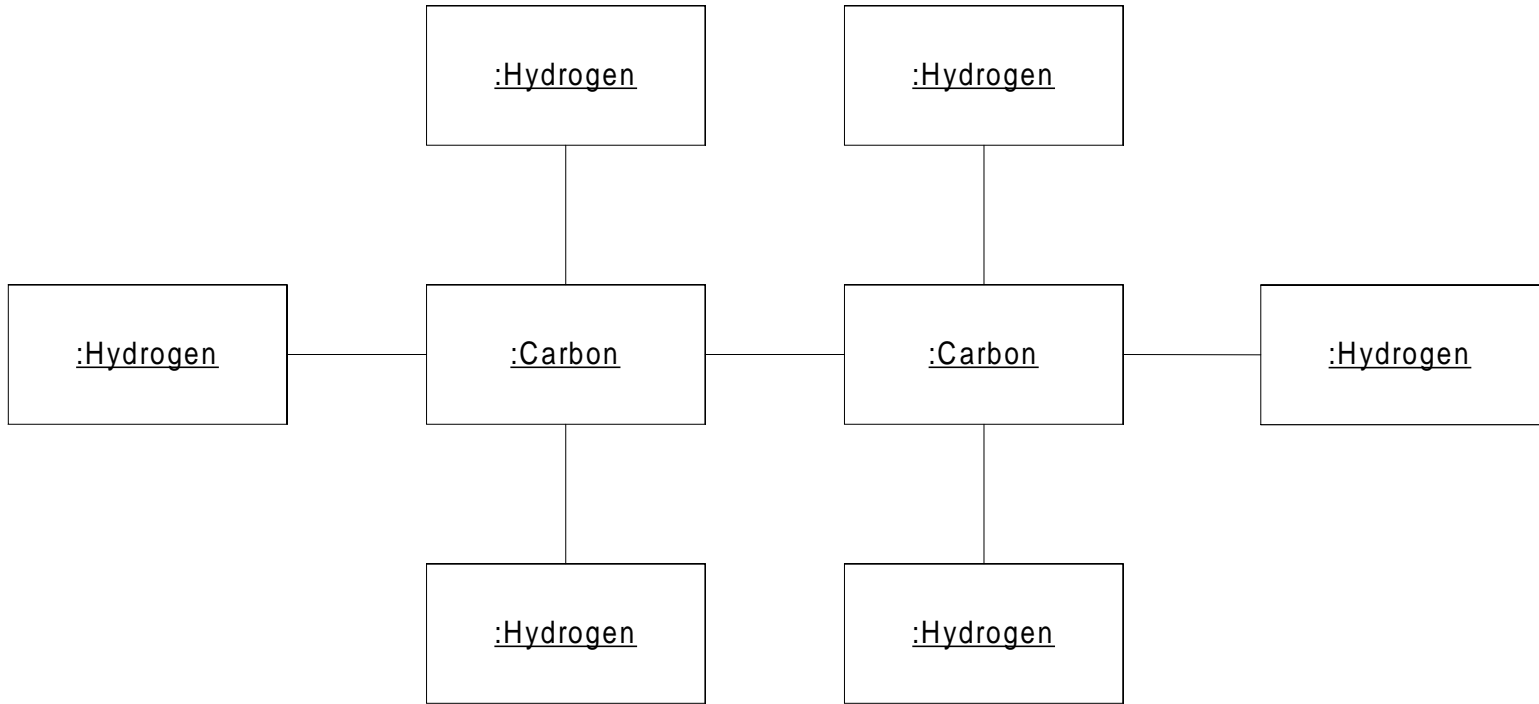
---

- The basic building blocks of UML are:
  - model elements (classes, interfaces, components, use cases, etc.)
  - relationships (associations, generalization, dependencies, etc.)
  - diagrams (class diagrams, use case diagrams, interaction diagrams, etc.)
- Simple building blocks are used to create large, complex structures
  - cf. elements, bonds and molecules in chemistry
  - cf. components, connectors and circuit boards in hardware

# Diagram: Classifier View



# Diagram: Instance View





# Well-Formedness Rules

---

- Well-formed: indicates that a model or model fragment adheres to all semantic and syntactic rules that apply to it.
- UML specifies rules for:
  - naming
  - scoping
  - visibility
  - integrity
  - execution (limited)
- However, during iterative, incremental development it is expected that models will be incomplete and inconsistent.



# Well-Formedness Rules (cont'd)

---

- Example of semantic rule: Class [1]

- **English:** If a Class is concrete, all the Operations of the Class should have a realizing Method in the full descriptor.
- **OCL:** `not self.isAbstract implies self.allOperations->forAll (op | self.allMethods->exists (m | m.specification->includes(op)))`



# Well-Formedness Rules (cont'd)

---

- Example of syntactic rules: Class
  - **Basic Notation:** A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines.
  - **Presentation Option:** Either or both of the attribute and operation compartments may be suppressed.
- Example of syntactic guideline: Class
  - **Style Guideline:** Begin class names with an uppercase letter.



# Unifying Concepts

---

- classifier-instance dichotomy
  - e.g., an object is an instance of a class OR  
a class is the classifier of an object
- specification-realization dichotomy
  - e.g., an interface is a specification of a class OR  
a class is a realization of an interface
- analysis-time vs. design-time vs. run-time
  - modeling phases (“process creep”)
  - usage guidelines suggested, not enforced

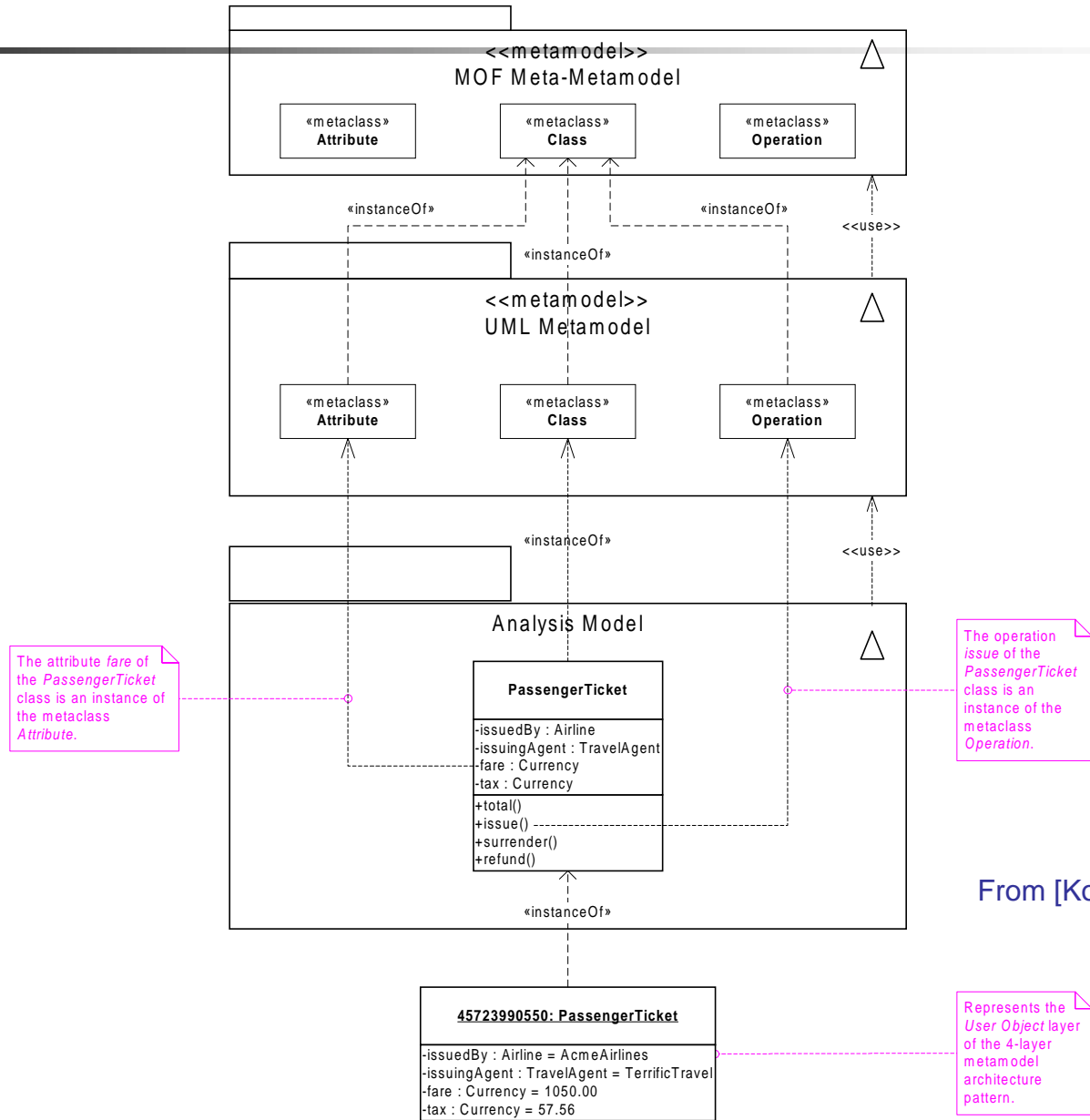


# Language Architecture

---

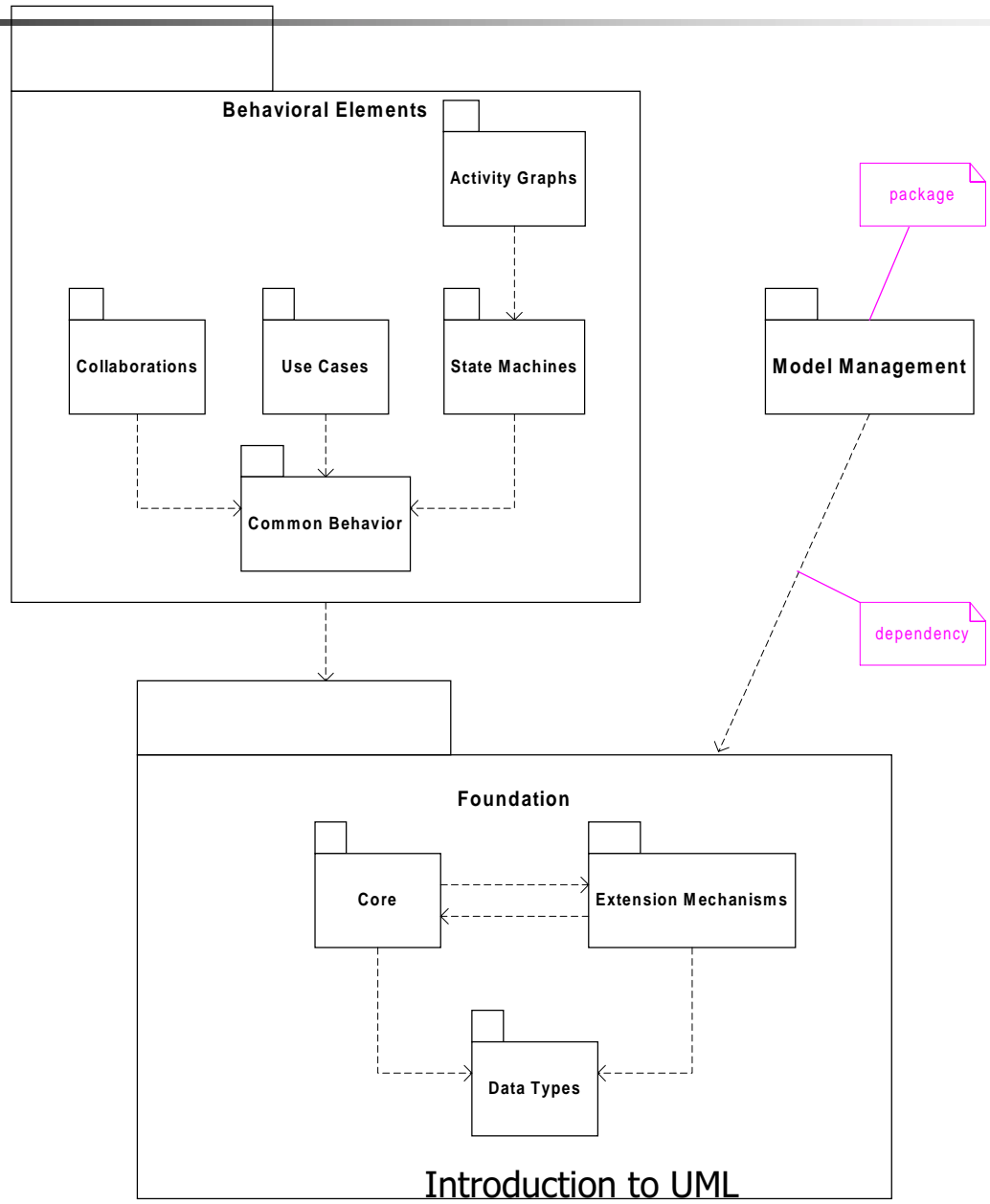
- Metamodel architecture
- Package structure

# Metamodel Architecture



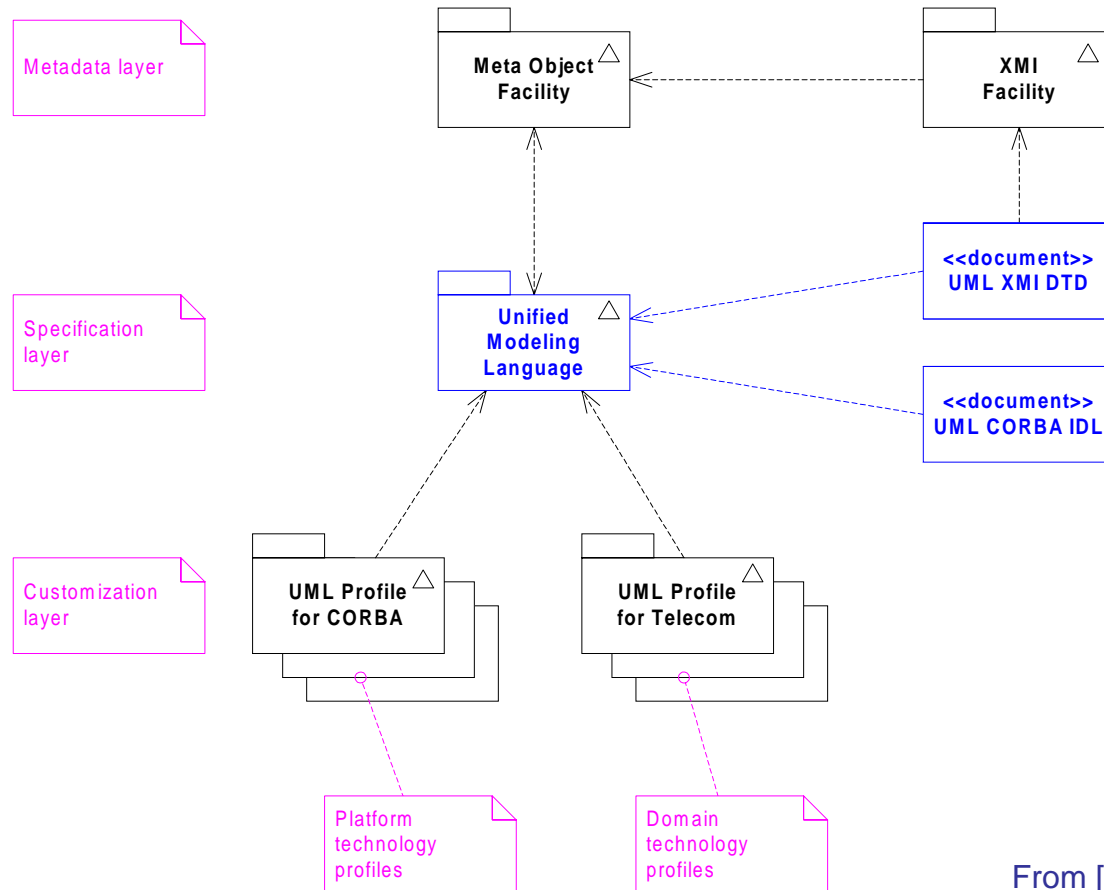
From [Kobryn 01b].

# UML Metamodel Layer



From [Kobryn 01b].

# Relationships to Other Modeling Technologies



From [Kobryn 01b].



# Structural Modeling

---

- What is structural modeling?
- Core concepts
- Diagram tour
- When to model structure
- Modeling tips
- Example: Interface-based design

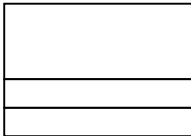


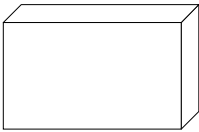


# What is structural modeling?

---


- Structural model: a view of an system that emphasizes the structure of the objects, including their classifiers, relationships, attributes and operations.

# Structural Modeling: Core Elements

Construct	Description	Syntax
<b>class</b>	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
<b>interface</b>	a named set of operations that characterize the behavior of an element.	
<b>component</b>	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
<b>node</b>	a run-time physical object that represents a computational resource.	







## *Structural Modeling: Core Elements* (cont'd)

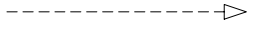
<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>constraint<sup>1</sup></b>	a semantic condition or restriction.	 {constraint}

<sup>1</sup> An extension mechanism useful for specifying structural elements.

# Structural Modeling: Core Relationships

Construct	Description	Syntax
<b>association</b>	a relationship between two or more classifiers that involves connections among their instances.	
<b>aggregation</b>	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	
<b>generalization</b>	a taxonomic relationship between a more general and a more specific element.	
<b>dependency</b>	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	

## *Structural Modeling: Core Relationships* (cont'd)

<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>realization</b>	a relationship between a specification and its implementation.	



# Structural Diagram Tour

---

- Show the static structure of the model
  - the entities that exist (e.g., classes, interfaces, components, nodes)
  - internal structure
  - relationship to other entities
- Do not show
  - temporal information
- Kinds
  - static structural diagrams
    - class diagram
    - object diagram
  - implementation diagrams
    - component diagram
    - deployment diagram



# Static Structural Diagrams

---

- Shows a graph of classifier elements connected by static relationships.
- kinds
  - class diagram: classifier view
  - object diagram: instance view



# Classes

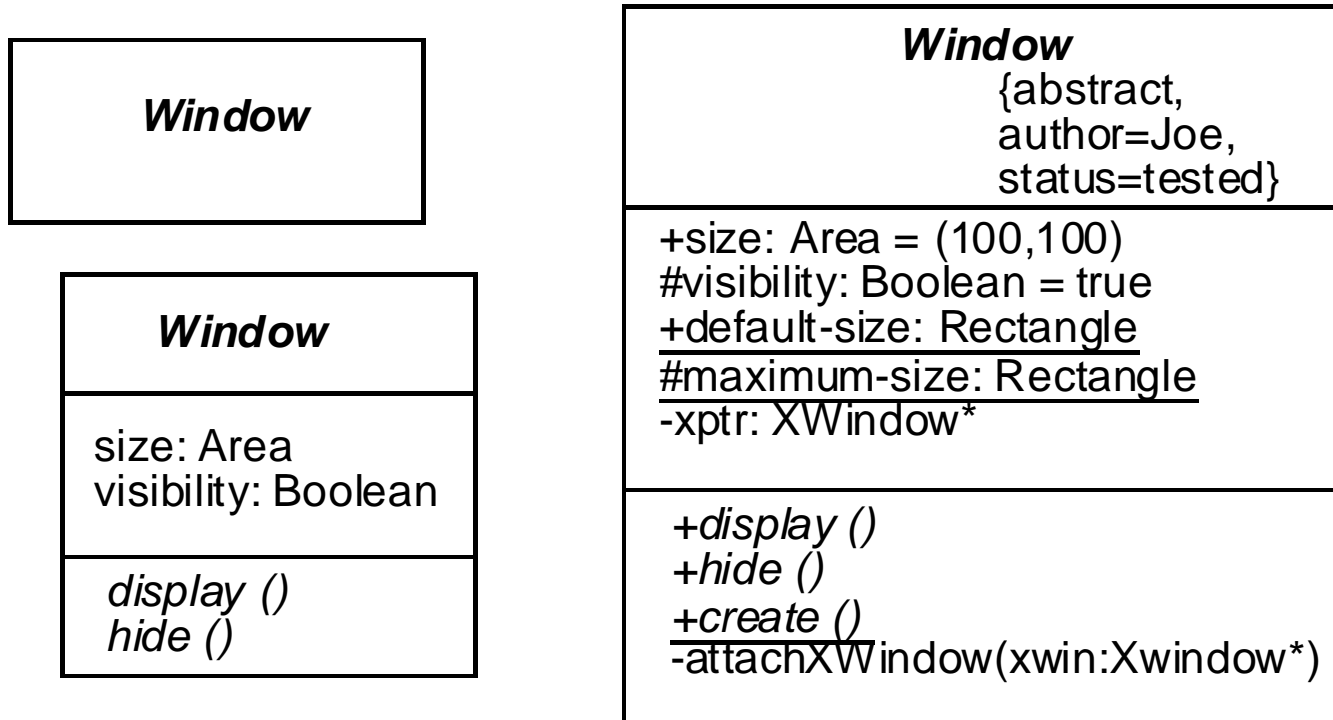


Fig. 3-20, *UML Notation Guide*



# Classes: compartments with names

---

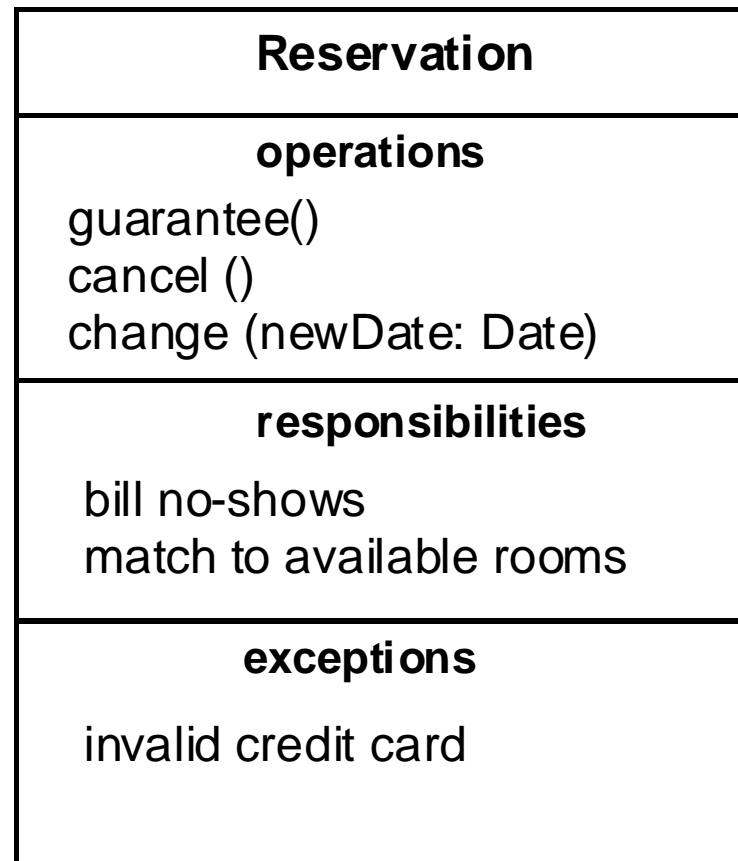


Fig. 3-23, *UML Notation Guide*

# Classes: method body

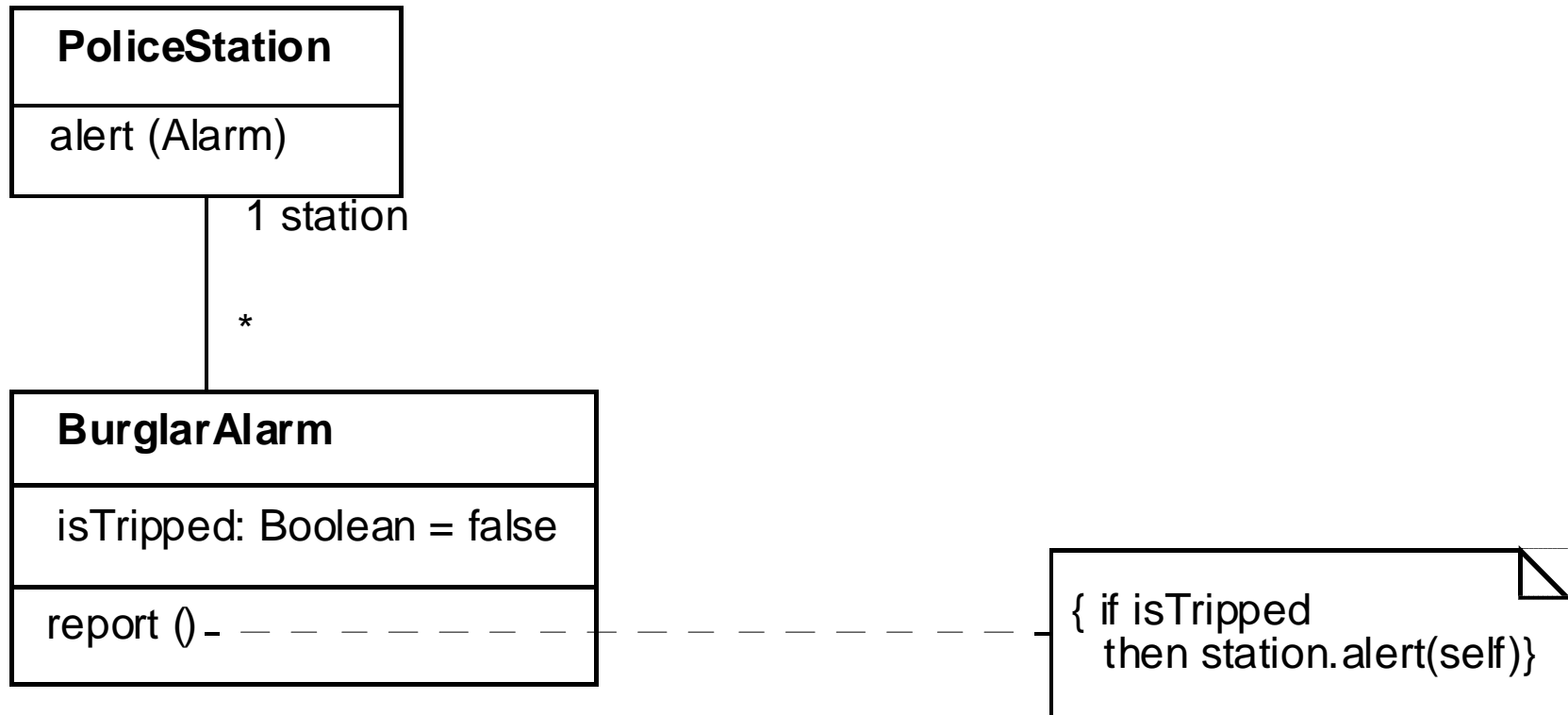


Fig. 3-24, *UML Notation Guide*

# Types and Implementation Classes

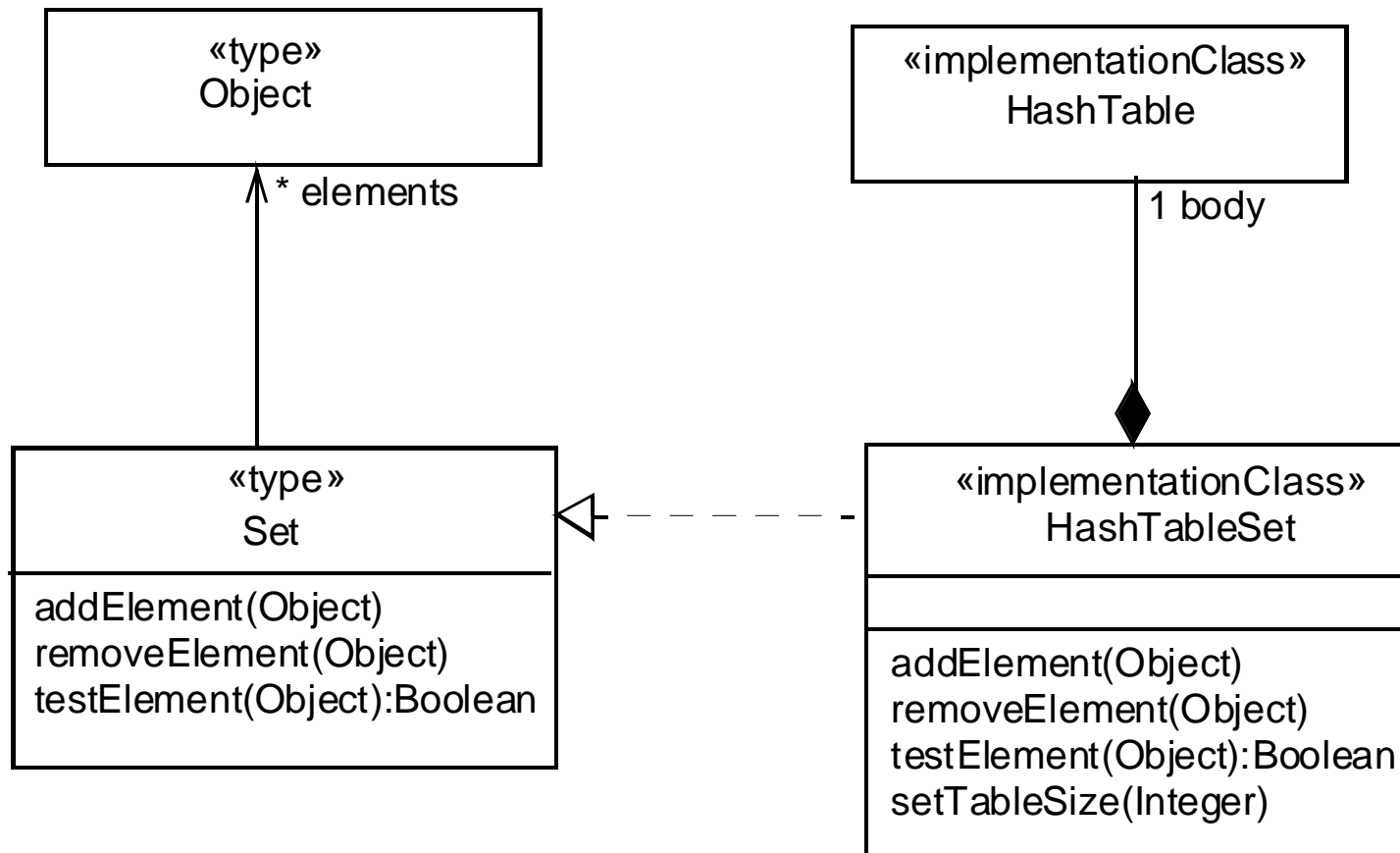


Fig. 3-27, *UML Notation Guide*

# Interfaces: Shorthand Notation

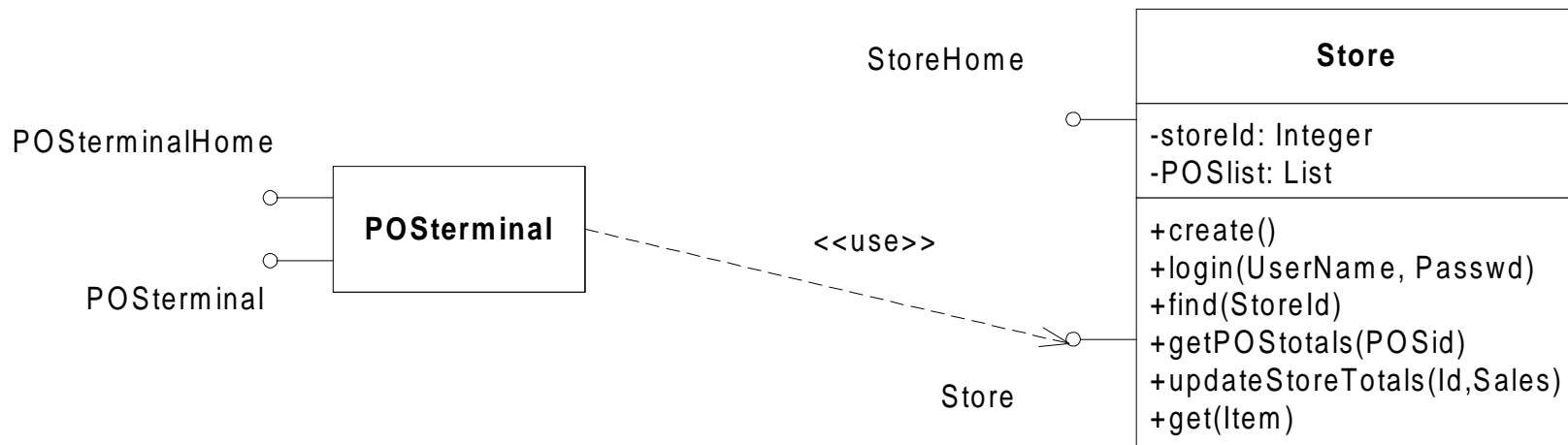


Fig. 3-29, *UML Notation Guide*

# Interfaces: Longhand Notation

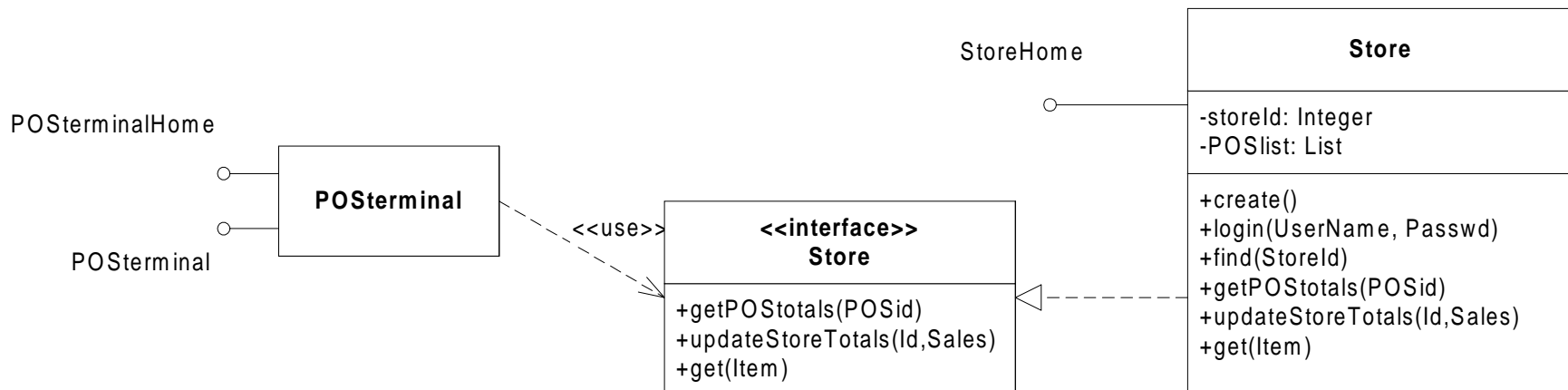


Fig. 3-29, *UML Notation Guide*



# Association Ends

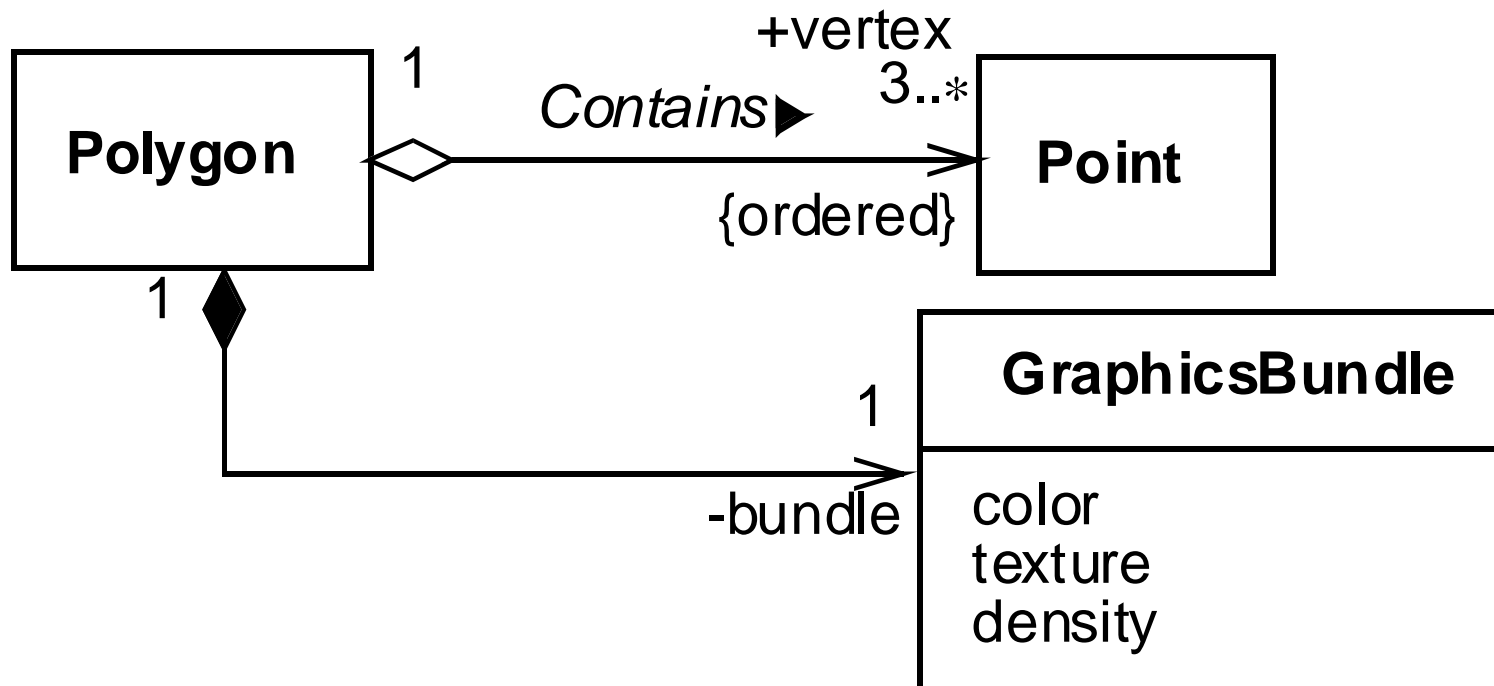


Fig. 3-41, *UML Notation Guide*

# Ternary Associations

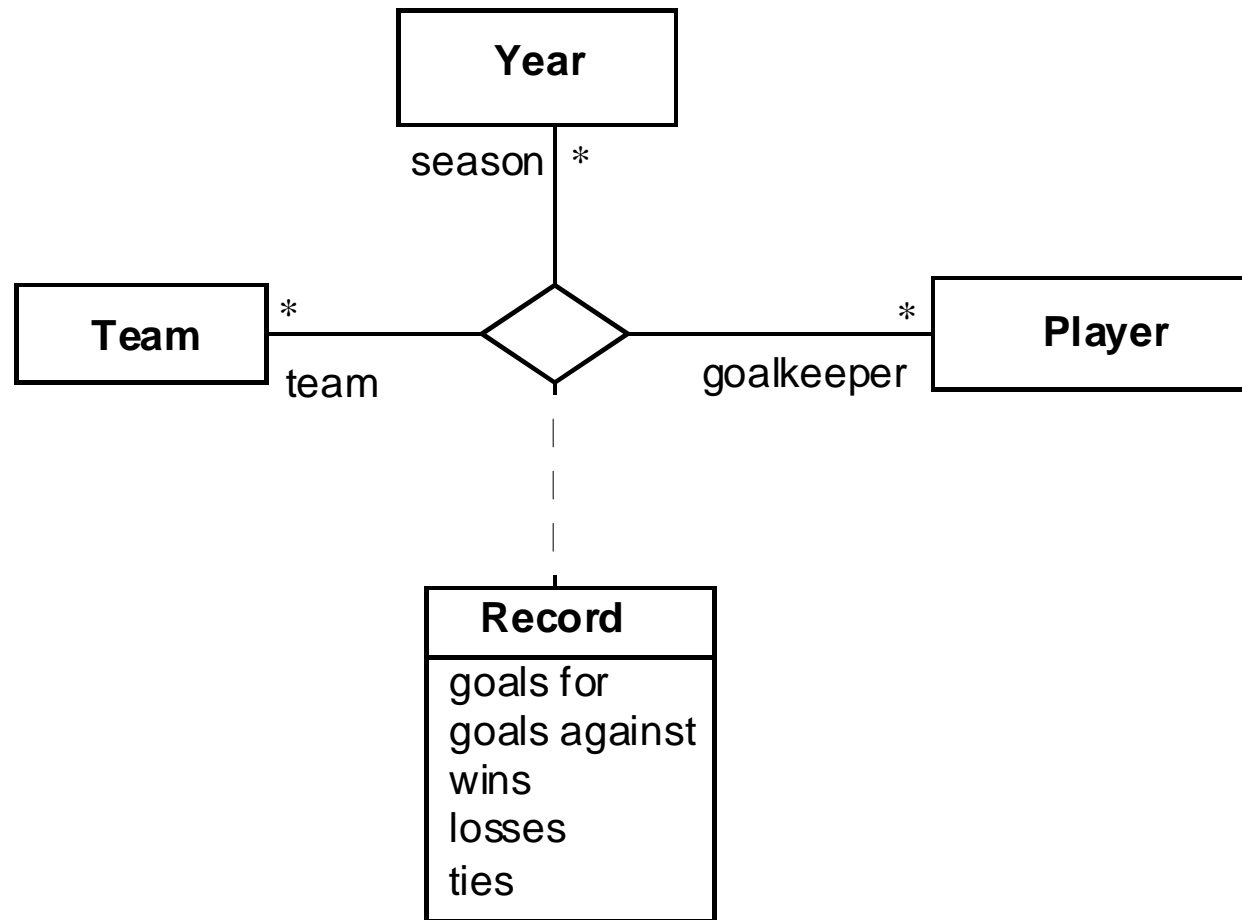


Fig. 3-44, *UML Notation Guide*

# Composition

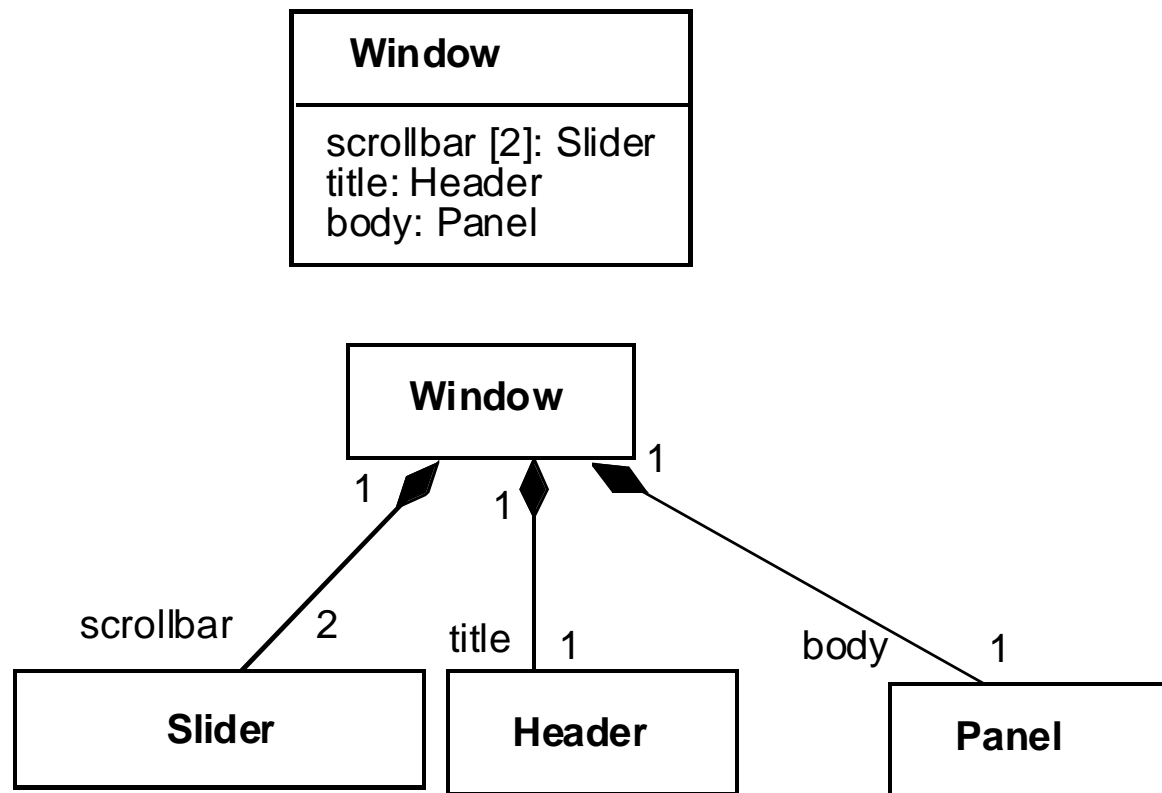


Fig. 3-45, *UML Notation Guide*

# Composition (cont'd)

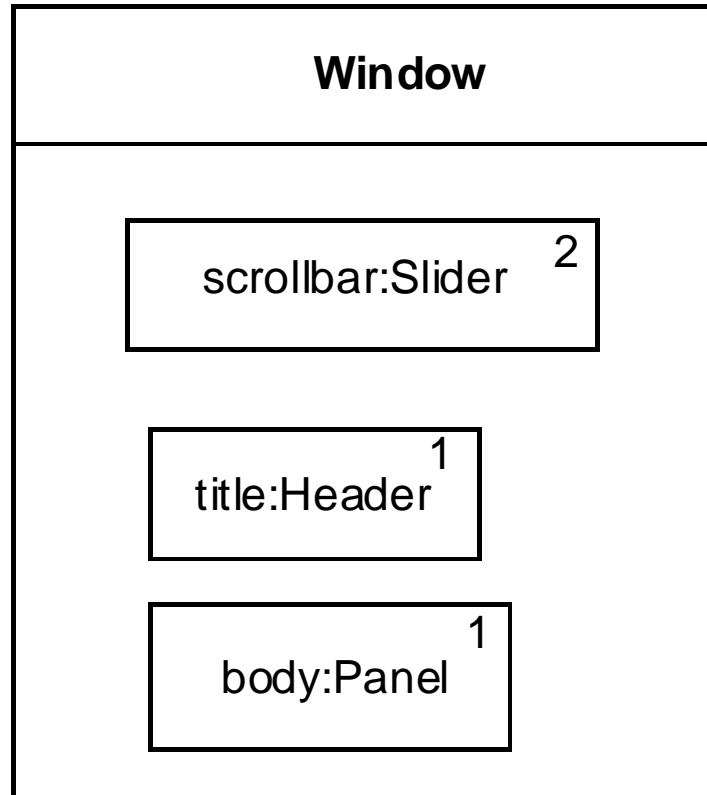


Fig. 3-45, *UML Notation Guide*

# Generalization

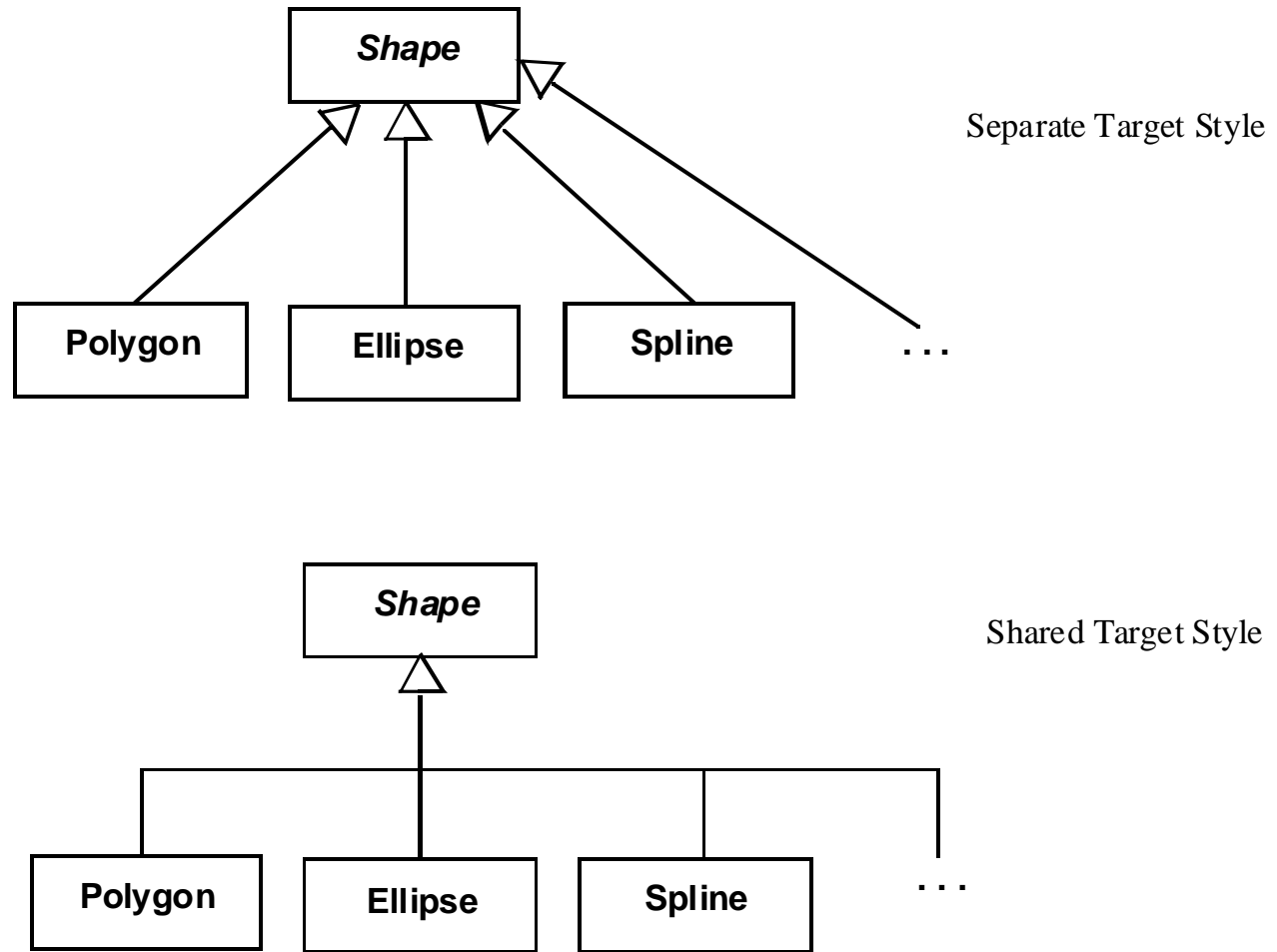


Fig. 3-47, *UML Notation Guide*

# Generalization

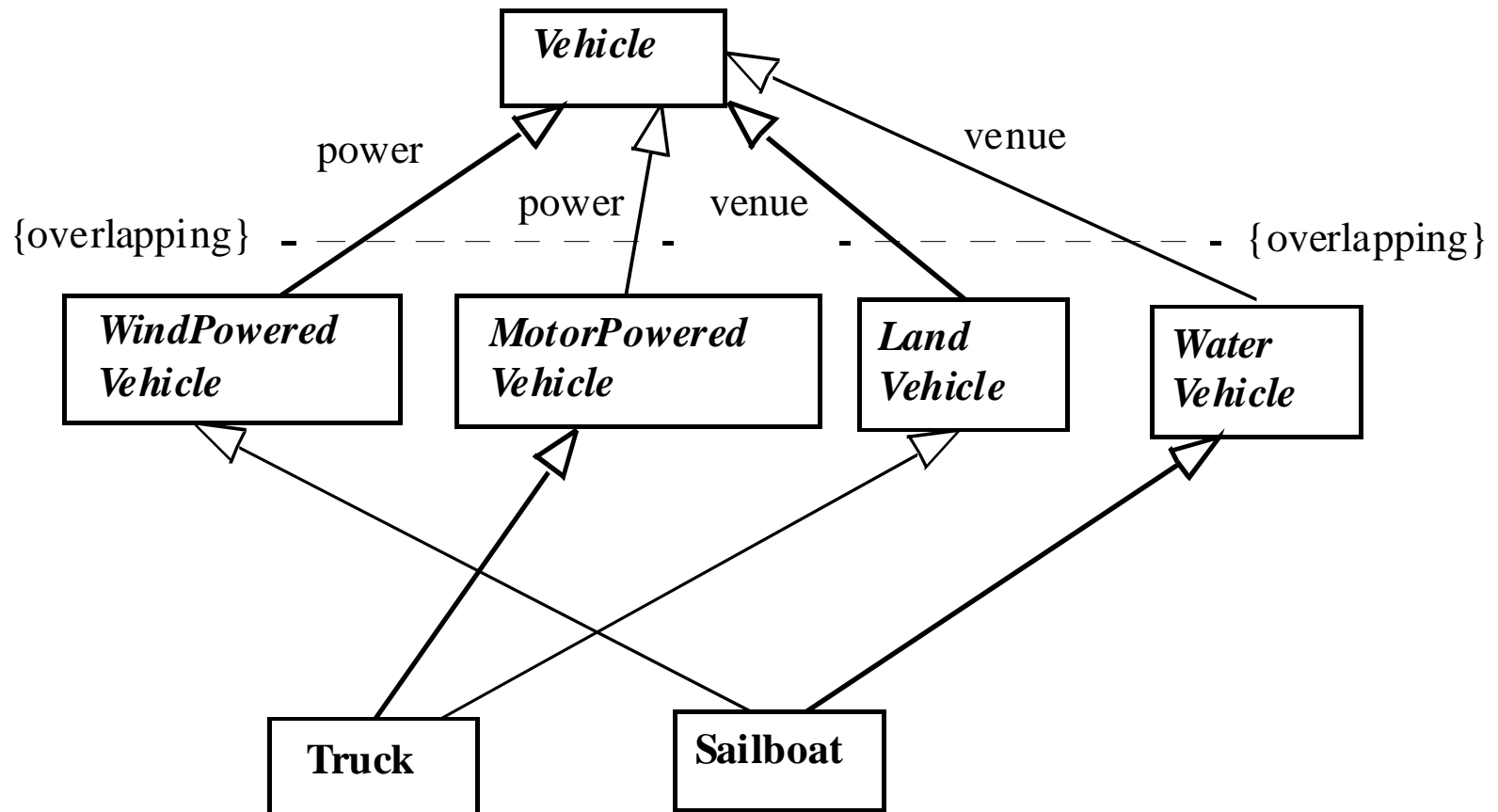


Fig. 3-48, *UML Notation Guide*

# Dependencies

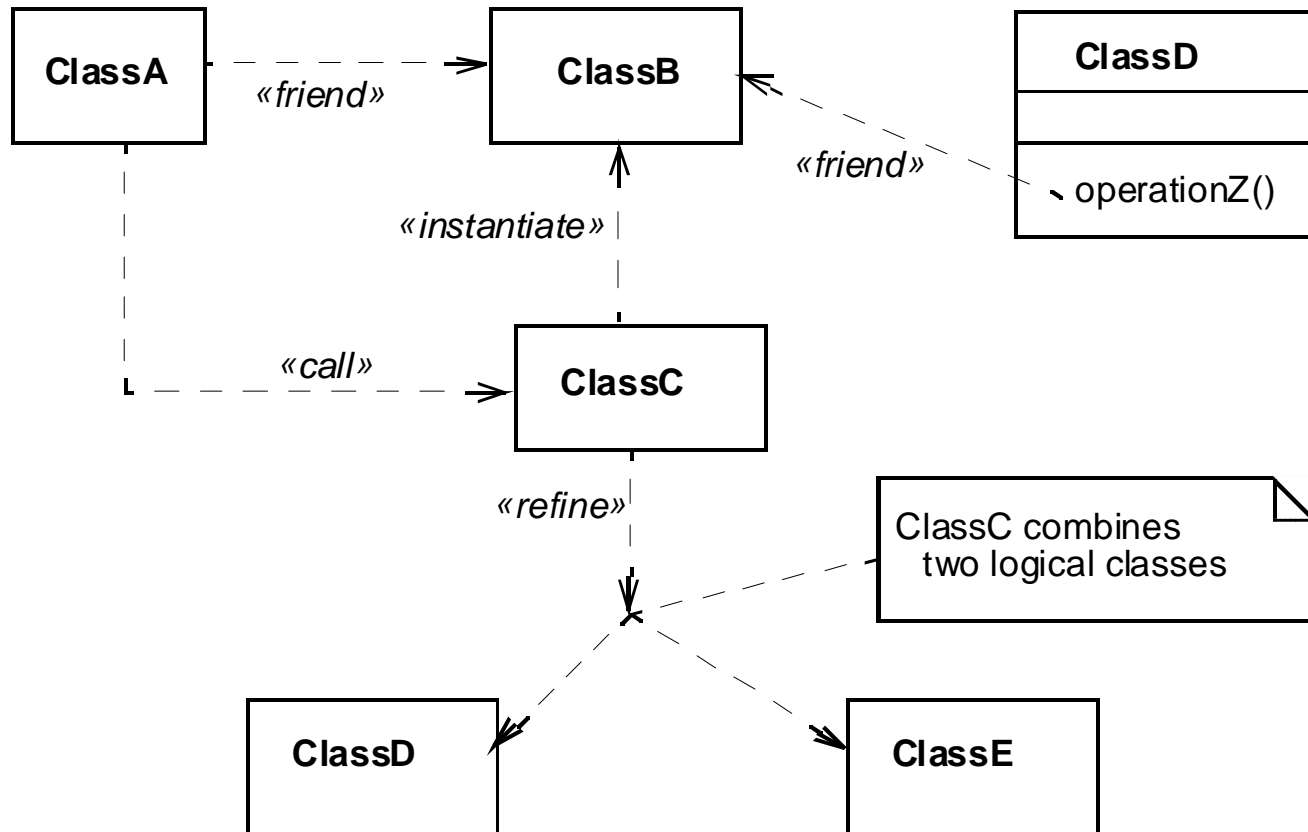


Fig. 3-50, UML Notation Guide

# Dependencies

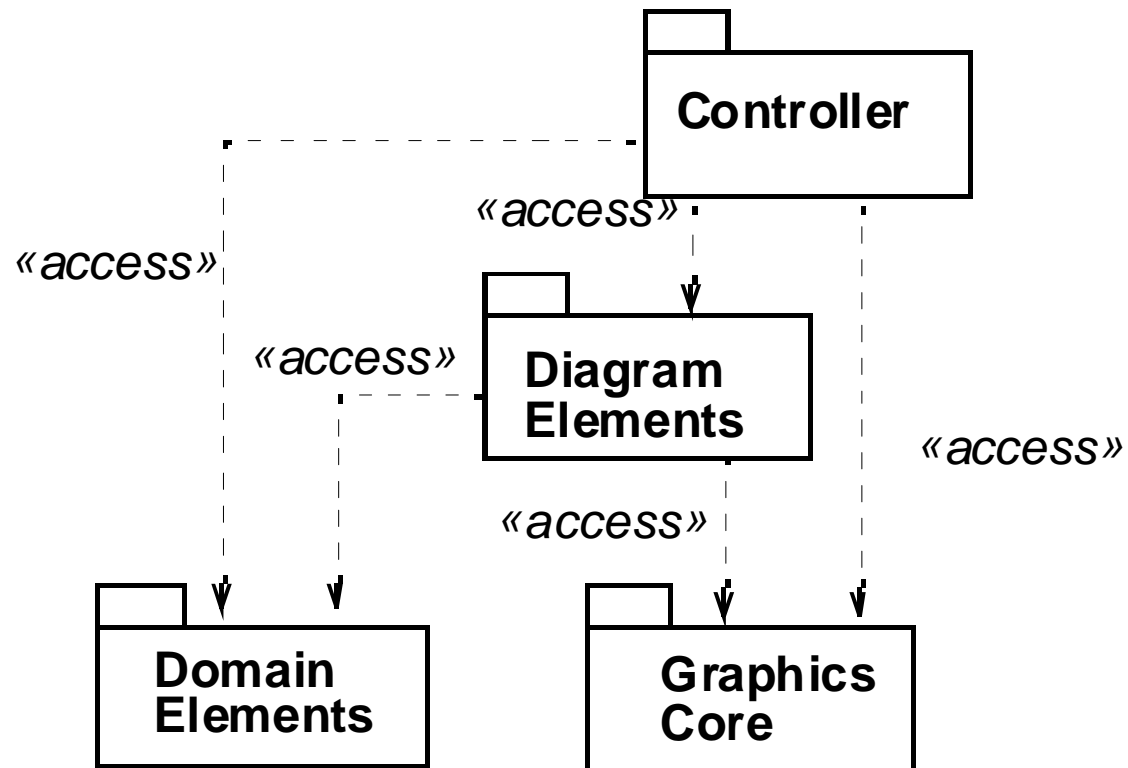


Fig. 3-51, *UML Notation Guide*

# Derived Attributes and Associations

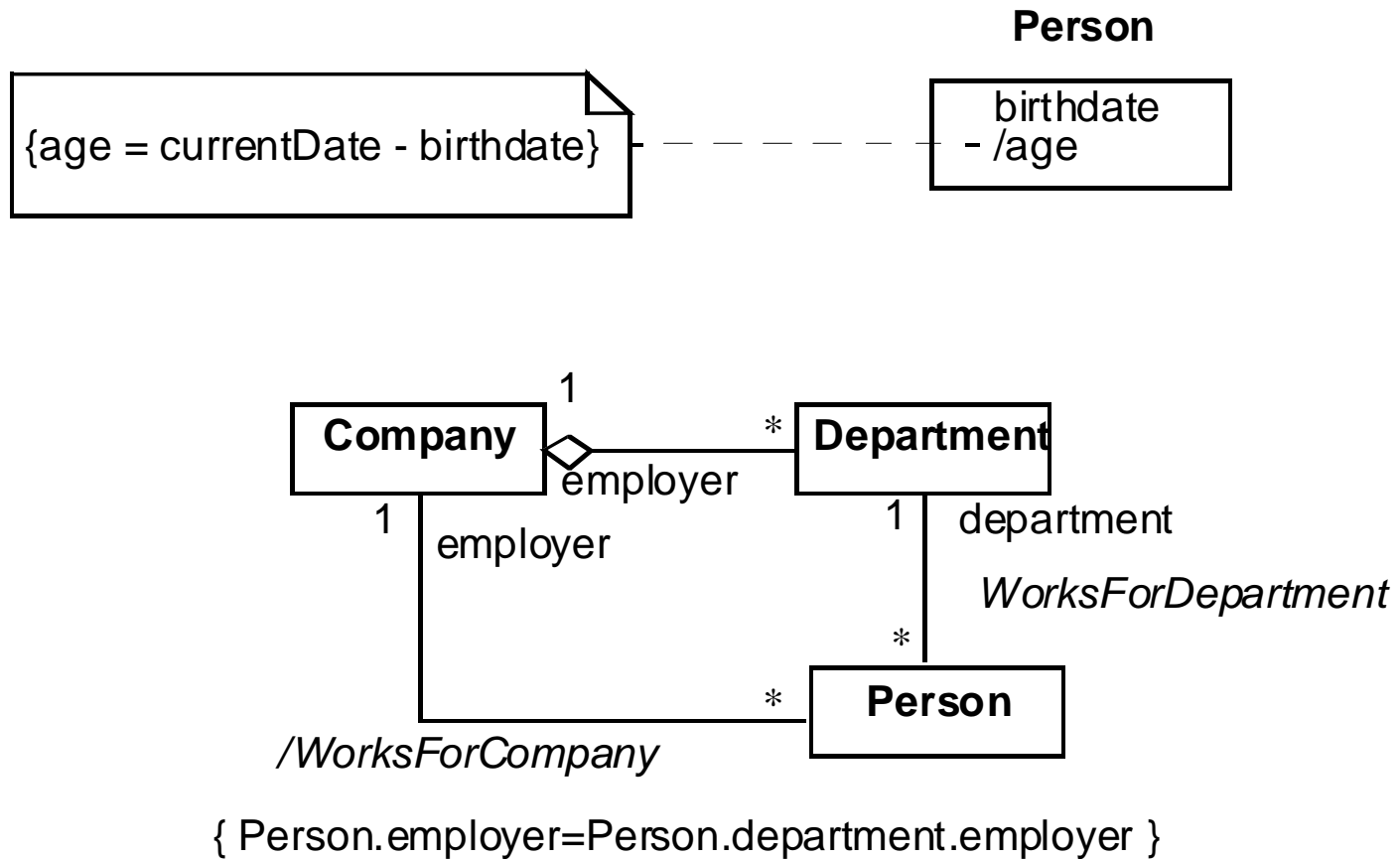


Fig. 3-52, UML Notation Guide

# Objects

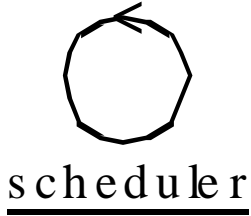
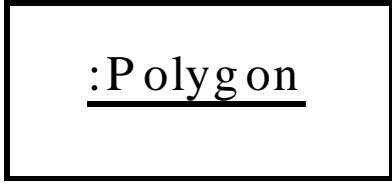
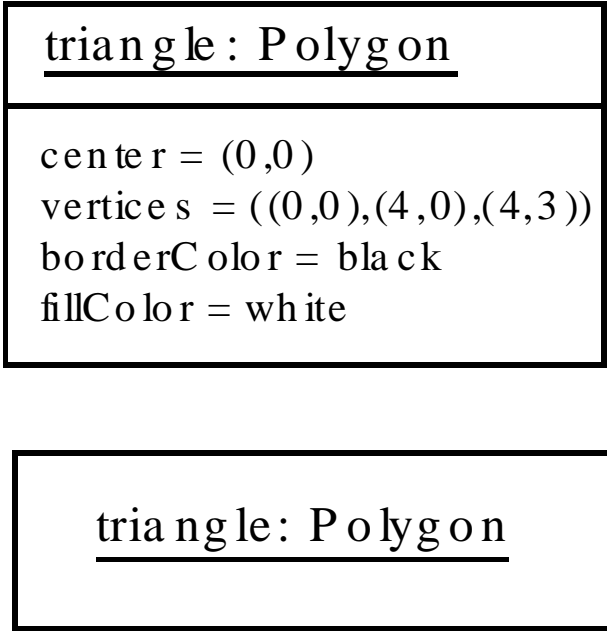


Fig. 3-38, UML Notation Guide

# Composite objects

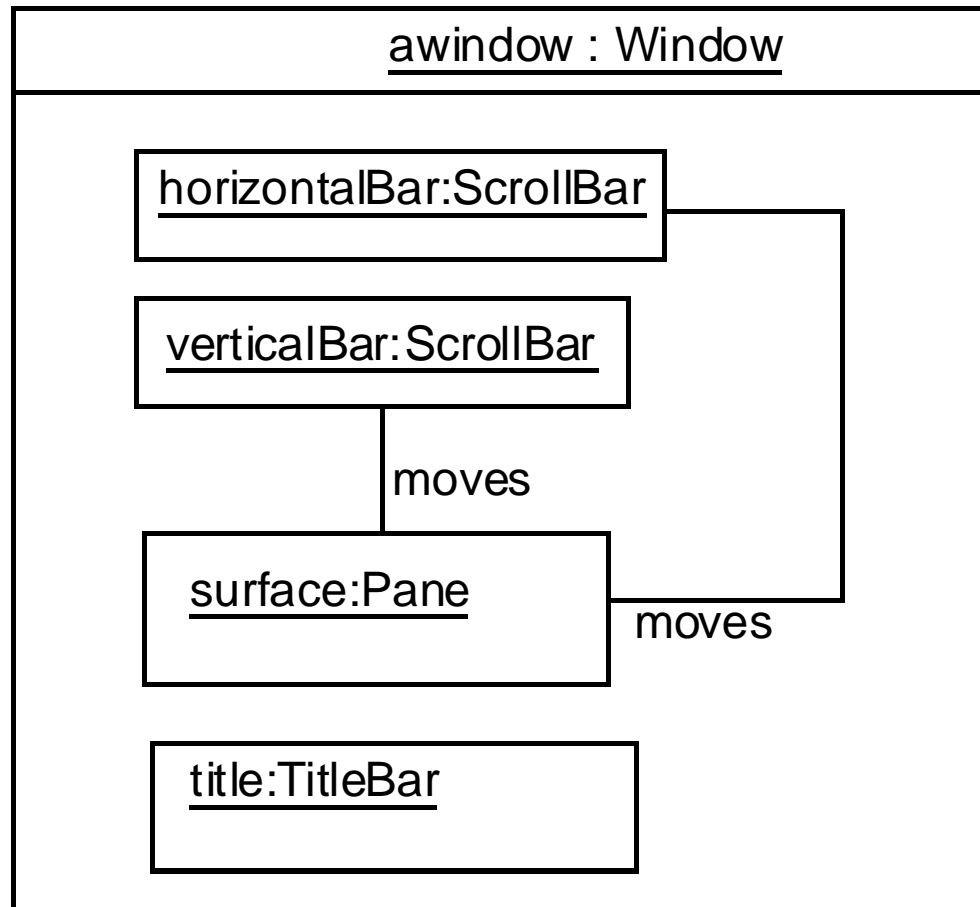


Fig. 3-39, *UML Notation Guide*

# Links

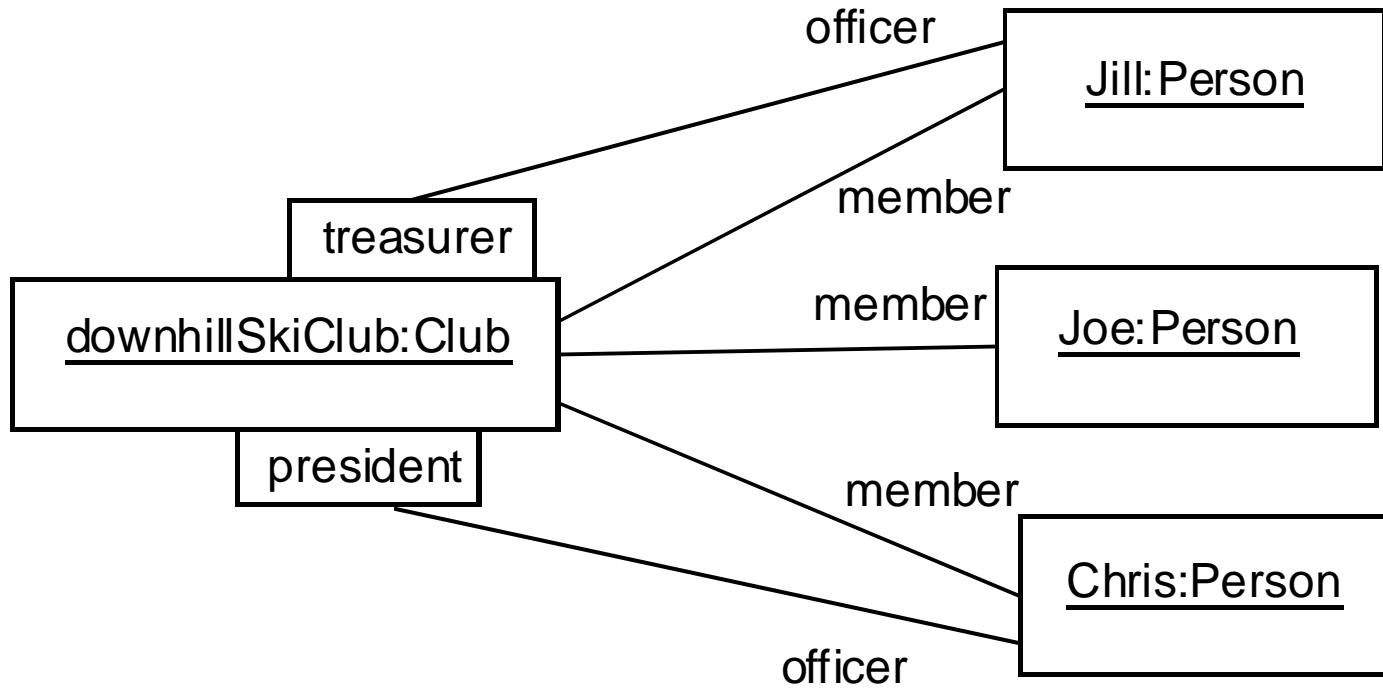


Fig. 3-46, *UML Notation Guide*

# Constraints and Comments

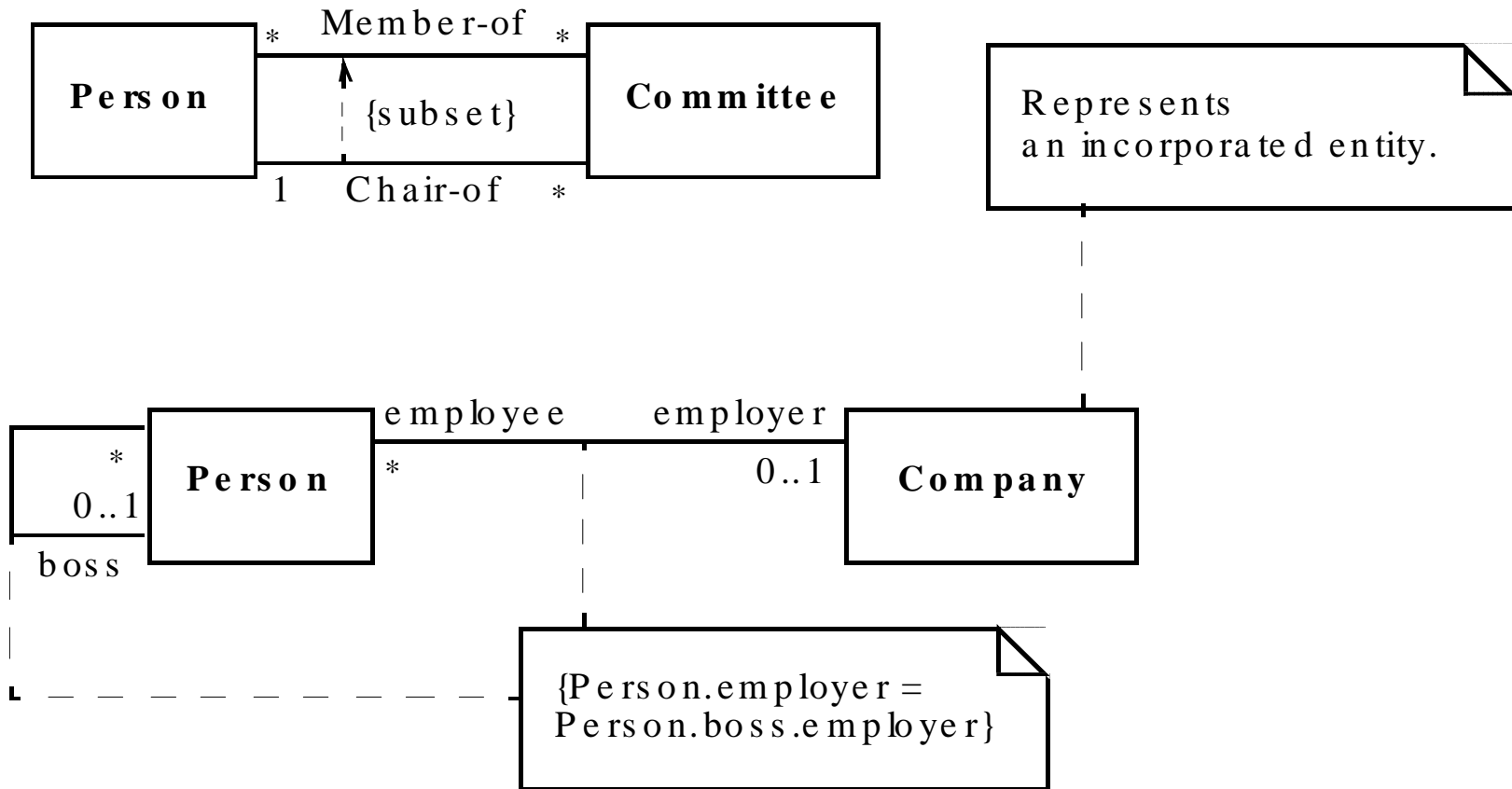
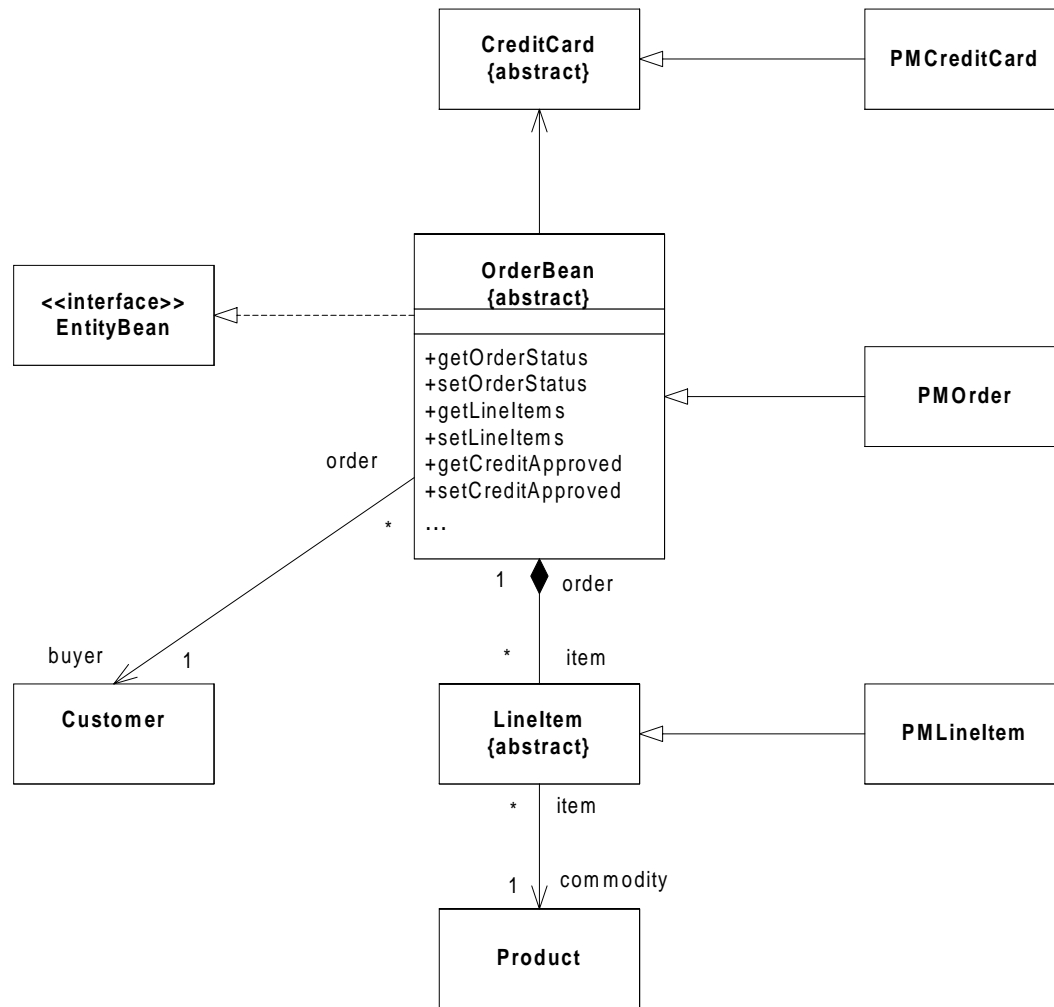


Fig. 3-17, UML Notation Guide

# Class Diagram Example



Adapted from Fig. 23 [EJB 2.0].



# Implementation Diagrams

---

- Show aspects of model implementation, including source code structure and run-time implementation structure
- Kinds
  - component diagram
  - deployment diagram



# Component Diagram

---

- Shows the organizations and dependencies among software components
- Components may be
  - specified by classifiers (e.g., implementation classes)
  - implemented by artifacts (e.g., binary, executable, or script files)

# Components

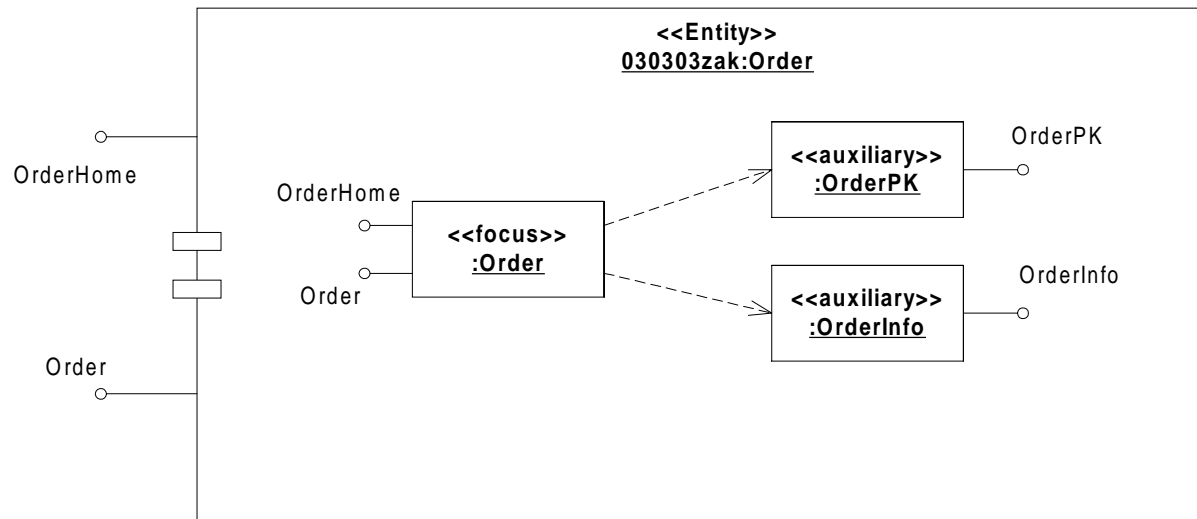
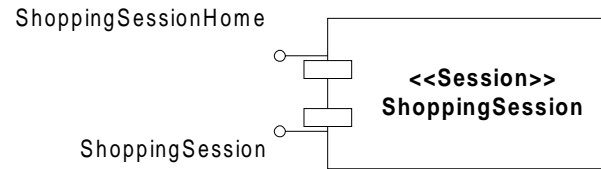


Fig. 3-99, UML Notation Guide

# Component Diagram

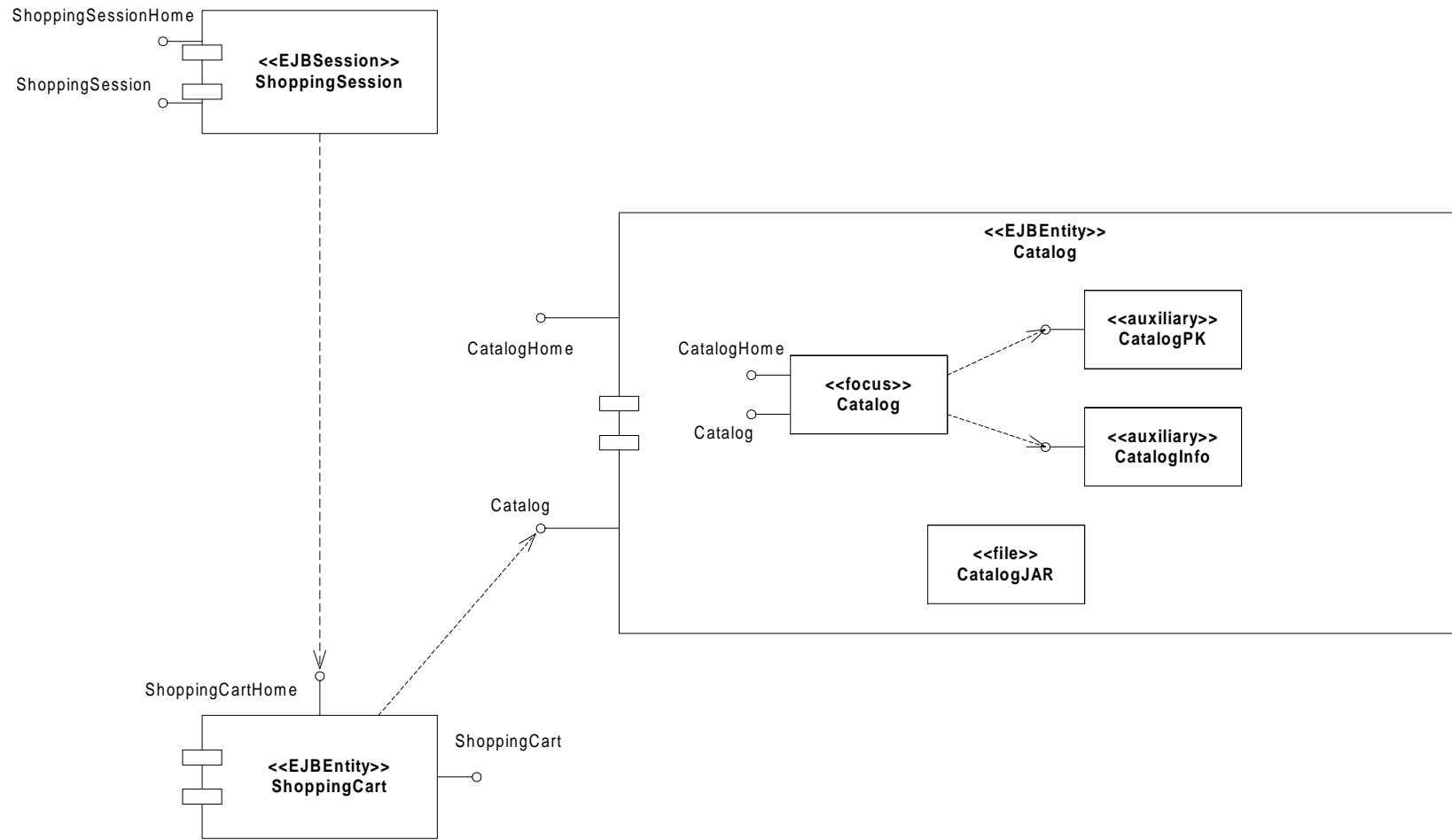


Fig. 3-95, UML Notation Guide

# Component Diagram with Relationships

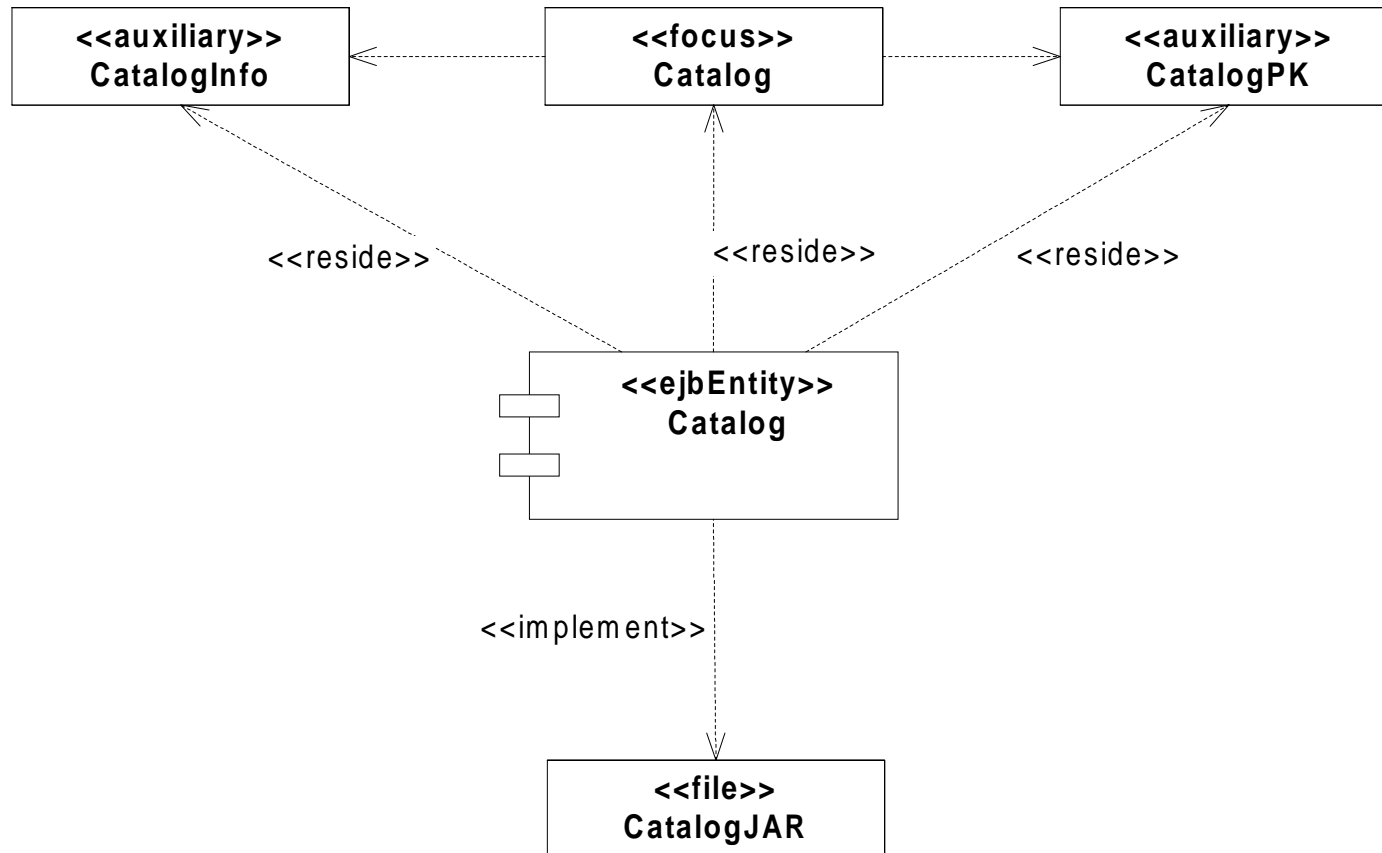


Fig. 3-96, *UML Notation Guide*



# Deployment Diagram

---

- Shows the configuration of run-time processing elements and the software components, processes and objects that live on them
- Deployment diagrams may be used to show which components may run on which nodes

# Deployment Diagram (1/2)

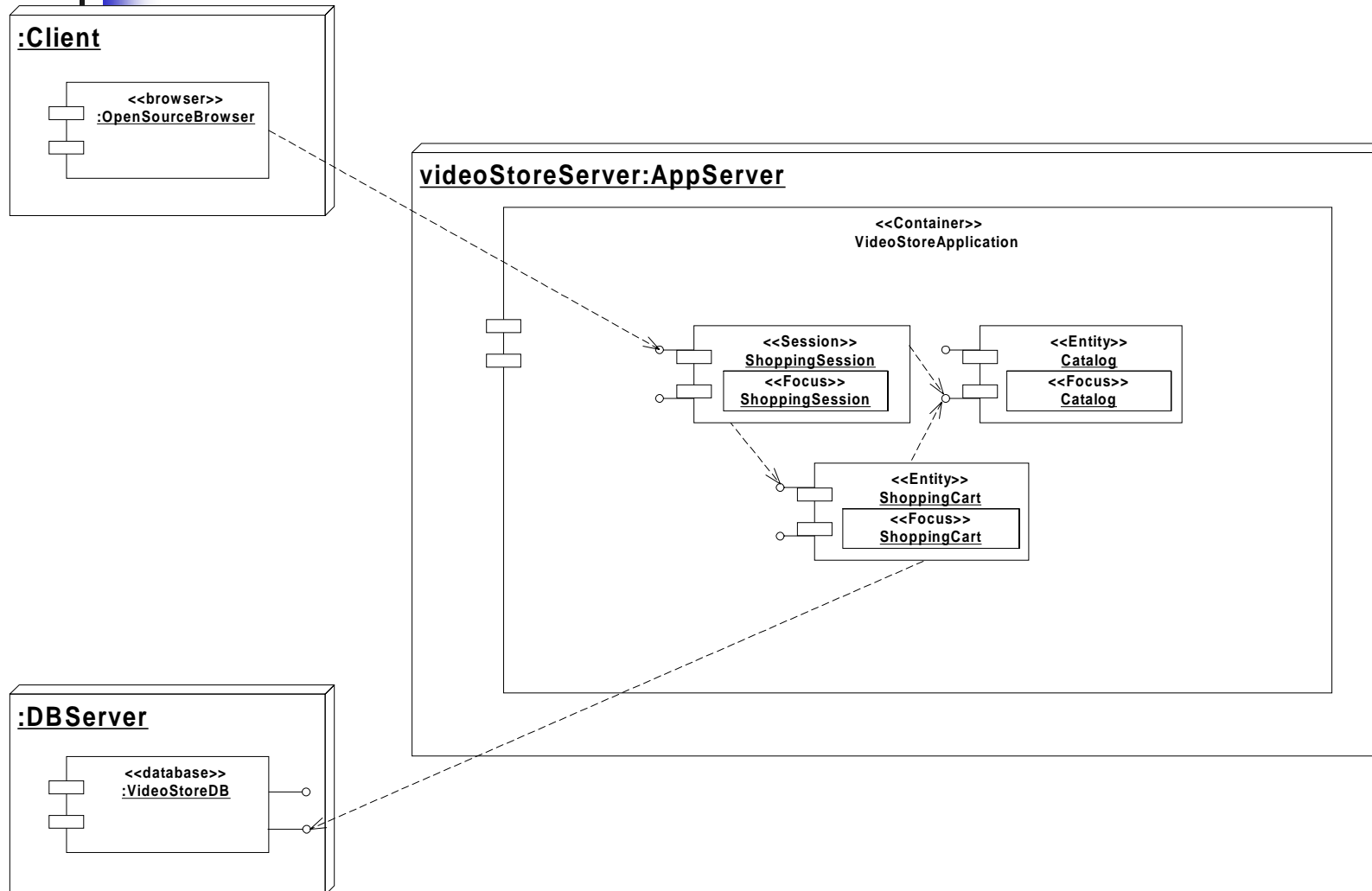


Fig. 3-97, UML Notation Guide

# Deployment Diagram (2/2)

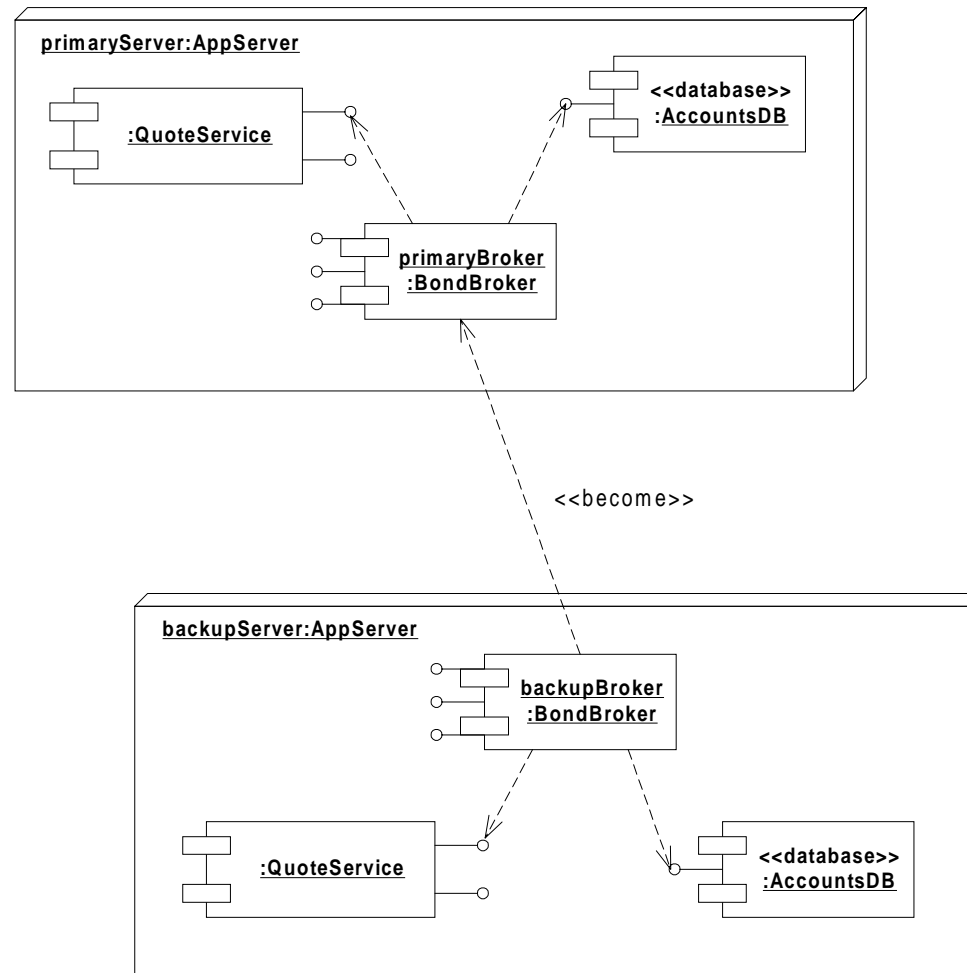


Fig. 3-98, *UML Notation Guide*



# When to model structure

---

- Adopt an opportunistic top-down+bottom-up approach to modeling structure
  - Specify the top-level structure using “architecturally significant” classifiers and model management constructs (packages, models, subsystems; see Tutorial 3)
  - Specify lower-level structure as you discover detail re classifiers and relationships
- If you understand your domain well you can frequently start with structural modeling; otherwise
  - If you start with use case modeling (as with a use-case driven method) make sure that your structural model is consistent with your use cases
  - If you start with role modeling (as with a collaboration-driven method) make sure that your structural model is consistent with your collaborations



# Structural Modeling Tips

---

- Define a “skeleton” (or “backbone”) that can be extended and refined as you learn more about your domain.
- Focus on using basic constructs well; add advanced constructs and/or notation only as required.
- Defer implementation concerns until late in the modeling process.
- Structural diagrams should
  - emphasize a particular aspect of the structural model
  - contain classifiers at the same level of abstraction
- Large numbers of classifiers should be organized into packages (see Lecture 3)



# Interface-Based Design

---

- Interface-based design is a design approach that
  - emphasizes the specification of system interfaces
  - separates the specification of service operations (interfaces) from their realization (implementation)
- CORBA IDL is typically used for interface-based design of CORBA applications
  - defines interfaces for business and system objects without constraining their implementations
  - defines the structure of an distributed application
  - doesn't allow you to specify object behavior or class relationships other than generalization



# Interface-Based Design (cont'd)

---

- The following example shows how UML can model the interfaces for a Point of Sale application originally specified in CORBA IDL. From [Kobryn 2000].



# Example: Interface-based design

---

```
module POS
{
    typedef long    POSId;
    typedef string Barcode;

    interface InputMedia
    {
        typedef string OperatorCmd;

        void        barcode_input(in Barcode    item);
        void        keypad_input( in OperatorCmd cmd);
    };

    interface OutputMedia
    {
        boolean     output_text( in string
string_to_print );
    };
    ...
}
```

Generic IDL Point of Sale (POS) example. [Siegel 00]



# Example: Interface-based design

---

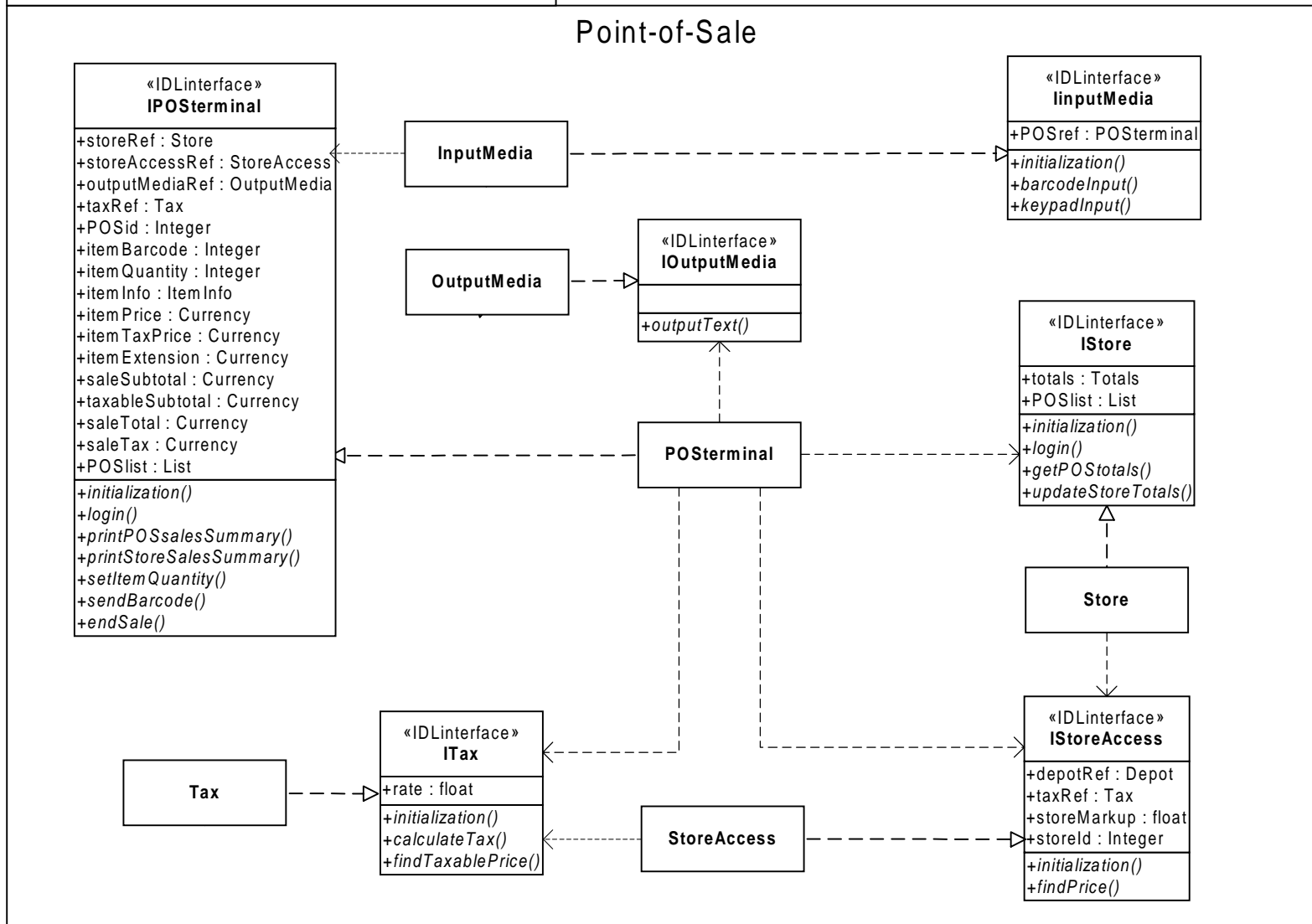
....

```
interface POSTerminal
{
    void login();
    void print_POS_sales_summary();
    void print_store_sales_summary();
    void send_barcode(          in Barcode          item);
    void item_quantity(          in long
quantity);
    void end_of_sale();
};

};

#endif /* _POS_IDL_ */
```

From [Kobryn 00].





# Use Case Modeling

---

- What is use case modeling?
- Core concepts
- Diagram tour
- When to model use cases
- Modeling tips
- Example: Online HR System

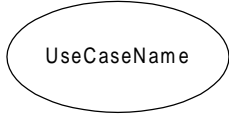
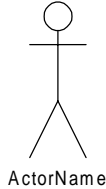



# What is use case modeling?


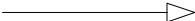
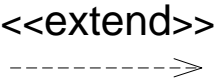
---

- use case model: a view of a system that emphasizes the behavior as it appears to outside users. A use case model partitions system functionality into transactions (‘use cases’) that are meaningful to users (‘actors’).

# Use Case Modeling: Core Elements

Construct	Description	Syntax
<b>use case</b>	A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.	
<b>actor</b>	A coherent set of roles that users of use cases play when interacting with these use cases.	
<b>system boundary</b>	Represents the boundary between the physical system and the actors who interact with the physical system.	

# Use Case Modeling: Core Relationships

Construct	Description	Syntax
<b>association</b>	The participation of an actor in a use case. i.e., instance of an actor and instances of a use case communicate with each other.	
<b>generalization</b>	A taxonomic relationship between a more general use case and a more specific use case.	
<b>extend</b>	A relationship from an <i>extension</i> use case to a <i>base</i> use case, specifying how the behavior for the extension use case can be inserted into the behavior defined for the base use case.	



## *Use Case Modeling: Core Relationships* (cont'd)

---

<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>include</b>	An relationship from a <i>base</i> use case to an <i>inclusion</i> use case, specifying how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.	<code>&lt;&lt;include&gt;&gt;</code> ----->



# Use Case Diagram Tour

---

- Shows use cases, actor and their relationships
- Use case internals can be specified by text and/or interaction diagrams (see Lecture 2)
- Kinds
  - use case diagram
  - use case description

# Use Case Diagram

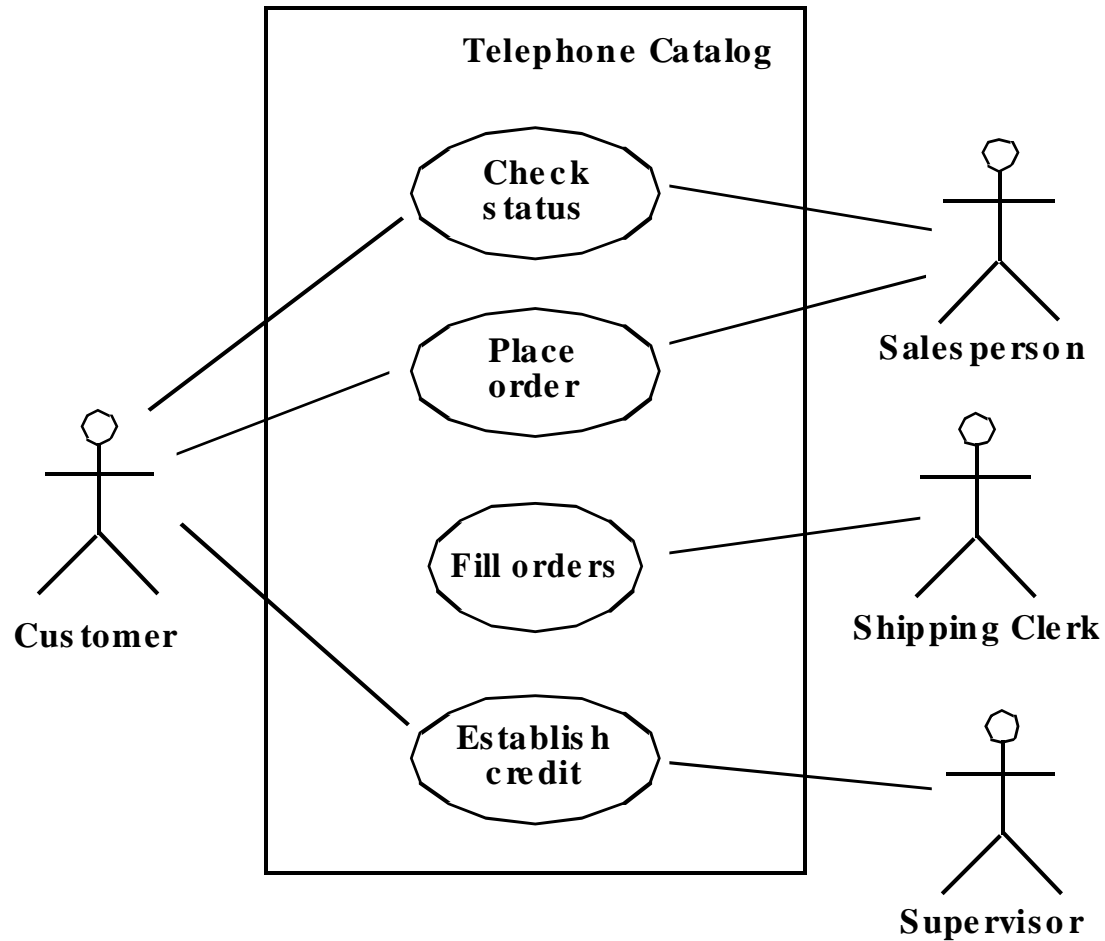


Fig. 3-53, *UML Notation Guide*

# Use Case Relationships

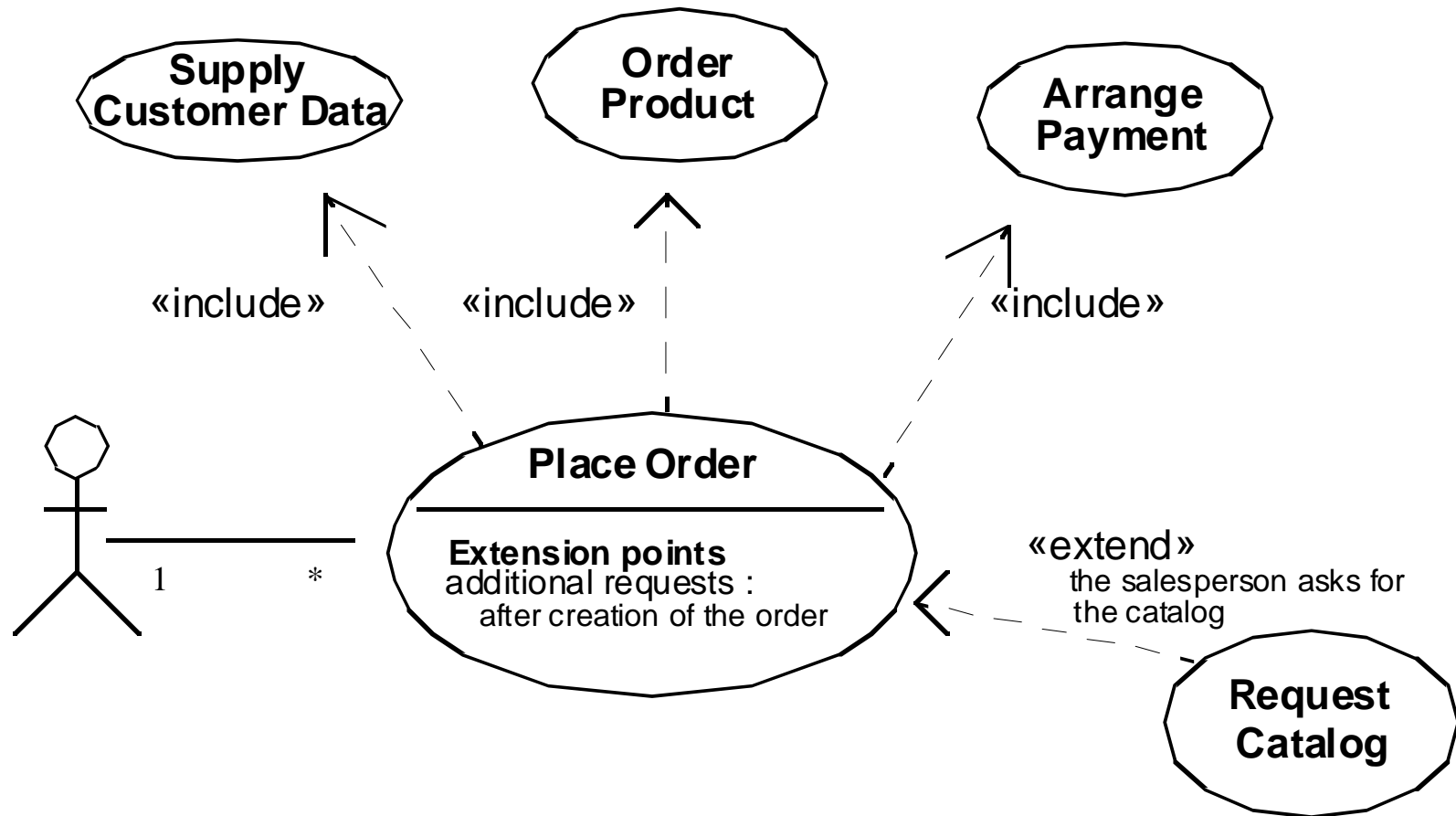


Fig. 3-54, *UML Notation Guide*

# Actor Relationships

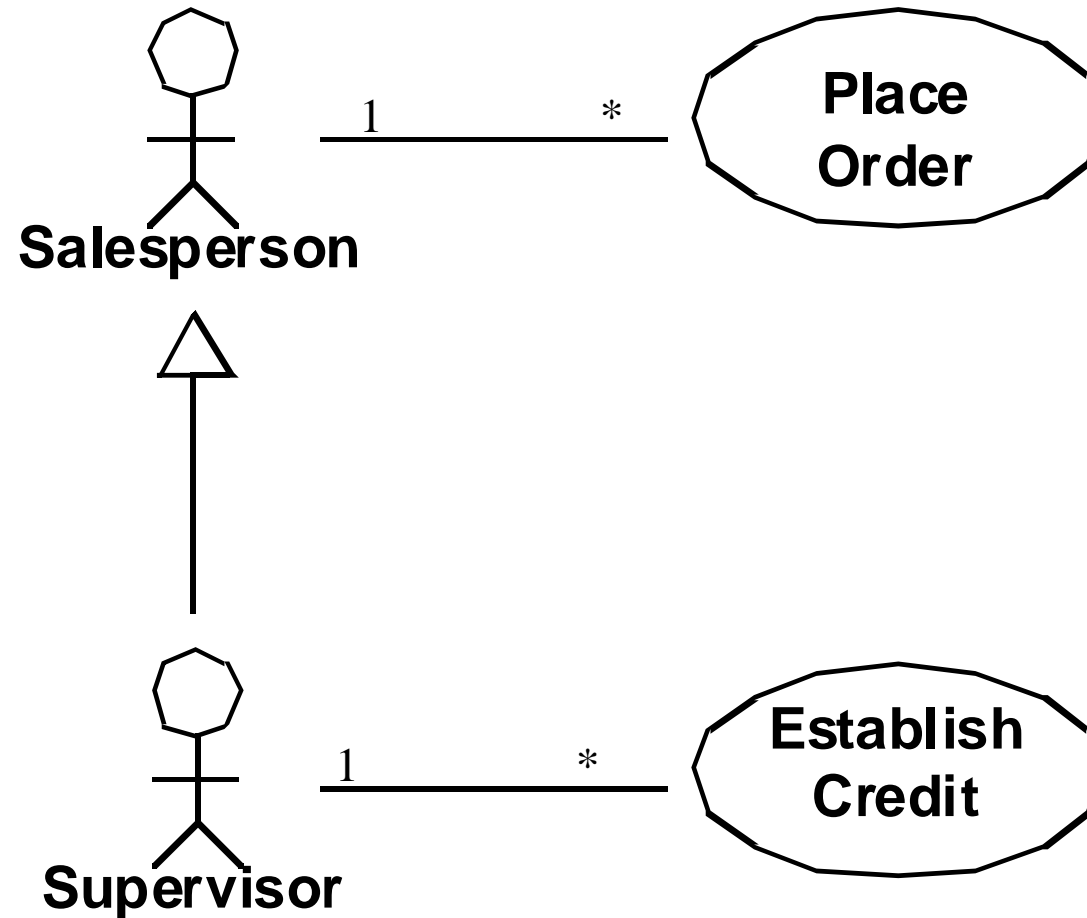


Fig. 3-55, *UML Notation Guide*



# Use Case Description: Change Flight

---

■ **Actors:** traveler, client account db, airline reservation system

■ **Preconditions:**

- Traveler has logged on to the system and selected 'change flight itinerary' option

■ **Basic course**

- System retrieves traveler's account and flight itinerary from client account database
- System asks traveler to select itinerary segment she wants to change; traveler selects itinerary segment.
- System asks traveler for new departure and destination information; traveler provides information.
- If flights are available then
- ...
- System displays transaction summary.

■ **Alternative courses**

- If no flights are available then ...



# When to model use cases

---

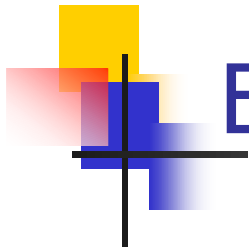
- Model user requirements with use cases.
- Model test scenarios with use cases.
- If you are using a use-case driven method
  - start with use cases and derive your structural and behavioral models from it.
- If you are not using a use-case driven method
  - make sure that your use cases are consistent with your structural and behavioral models.



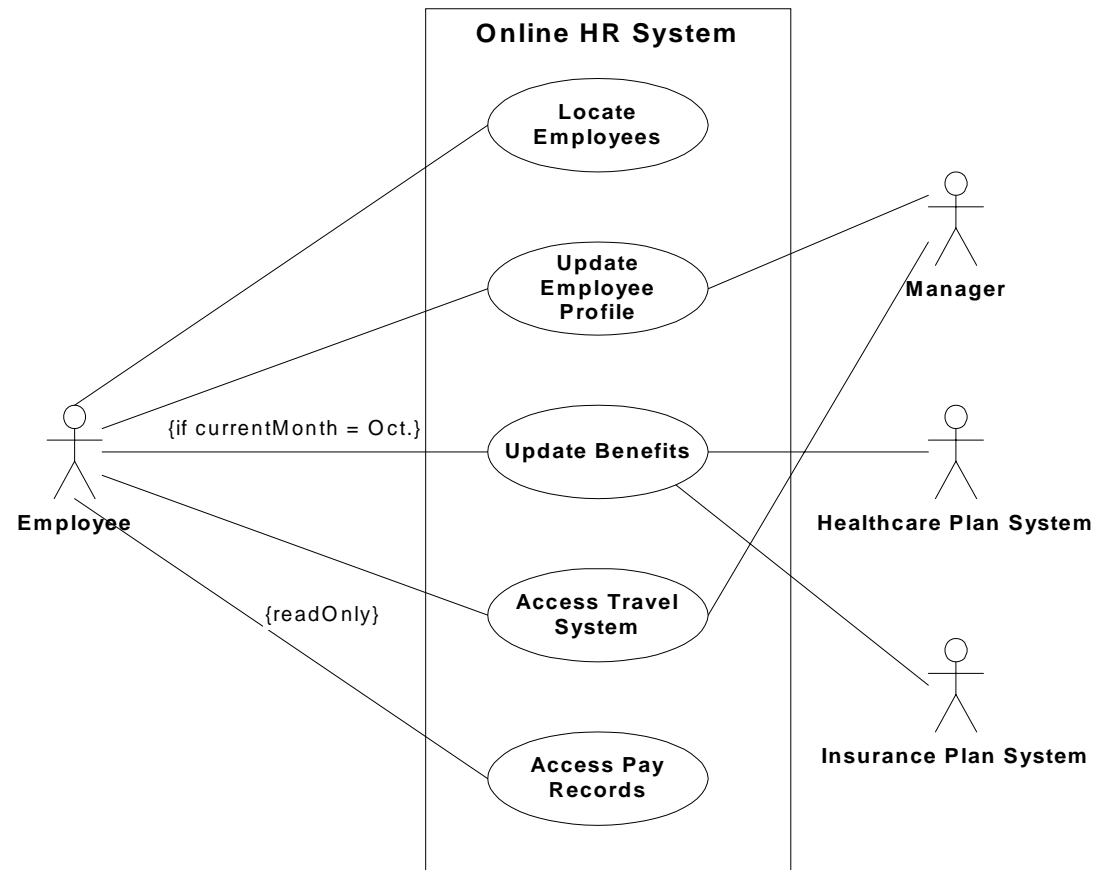
# Use Case Modeling Tips

---

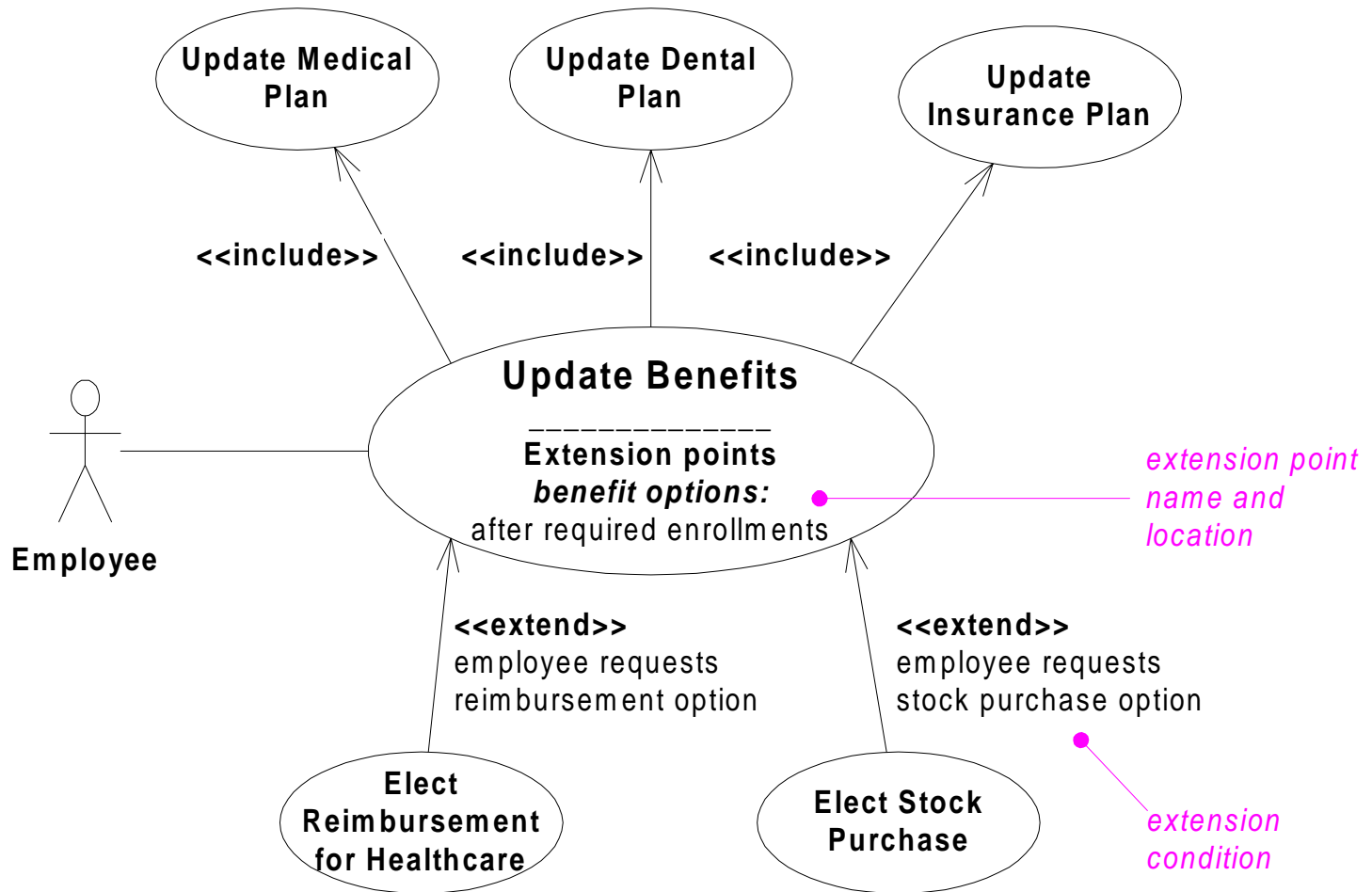
- Make sure that each use case describes a significant chunk of system usage that is understandable by both domain experts and programmers
- When defining use cases in text, use nouns and verbs accurately and consistently to help derive objects and messages for interaction diagrams (see Lecture 2)
- Factor out common usages that are required by multiple use cases
  - If the usage is required use <<include>>
  - If the base use case is complete and the usage may be optional, consider use <<extend>>
- A use case diagram should
  - contain only use cases at the same level of abstraction
  - include only actors who are required
- Large numbers of use cases should be organized into packages (see Lecture 3)



# Example: Online HR System



# Online HR System: Use Case Relationships





# Online HR System: Update Benefits Use Case

---

■ **Actors:** employee, employee account db, healthcare plan system, insurance plan system

■ **Preconditions:**

- Employee has logged on to the system and selected 'update benefits' option

■ **Basic course**

- System retrieves employee account from employee account db
- System asks employee to select medical plan type; **include** Update Medical Plan.
- System asks employee to select dental plan type; **include** Update Dental Plan.
- ...

■ **Alternative courses**

- If health plan is not available in the employee's area the employee is informed and asked to select another plan...



# Wrap Up

---

- Ideas to take away
- Preview of next tutorial
- References
- Further info



# Ideas to Take Away

---

- UML is effective for modeling large, complex software systems
- It is simple to learn for most developers, but provides advanced features for expert analysts, designers and architects
- It can specify systems in an implementation-independent manner
- 10-20% of the constructs are used 80-90% of the time
- Structural modeling specifies a skeleton that can be refined and extended with additional structure and behavior
- Use case modeling specifies the functional requirements of system in an object-oriented manner



# Preview - Next Tutorial

---

- Behavioral Modeling with UML
  - Behavioral modeling overview
  - Interactions
  - Collaborations
  - Statecharts
  - Activity Graphs



# References

---

- [UML 1.3] *OMG UML Specification v. 1.3*, OMG doc# ad/06-08-99
- [UML 1.4] *OMG UML Specification v. 1.4*, UML Revision Task Force recommended final draft, OMG doc# ad/01-02-13.
- [Kobryn 01a] C. Kobryn, "UML 2.0 Roadmap: Fast Track or Detours?," *Software Development*, April 2001. To appear.
- [Kobryn 01b] C. Kobryn, "Modeling Distributed Applications with UML," chapter in [Siegel 01] *Quick CORBA 3*, Wiley, 2001. To be published.
- [Kobryn 00] "Modeling CORBA Applications with UML," chapter 21 in [Siegel 00] *CORBA 3 Fundamentals and Programming* (2<sup>nd</sup> ed.), Wiley, 2000.
- [Kobryn 99] *UML 2001: A Standardization Odyssey*, Communications of the ACM, Oct. 1999.
- [EJB 2.0] *Enterprise JavaBeans Specification v. 2.0*, Sun Microsystems, March 31, 2000.



# Further Info

---

- Web:

- UML 1.4 RTF: [www.celigent.com/omg/umlrtf](http://www.celigent.com/omg/umlrtf)
- OMG UML Tutorials: [www.celigent.com/omg/umlrtf/tutorials.htm](http://www.celigent.com/omg/umlrtf/tutorials.htm)
- UML 2.0 Working Group: [www.celigent.com/omg/adptf/wgs/uml2wg.htm](http://www.celigent.com/omg/adptf/wgs/uml2wg.htm)
- OMG UML Resources: [www.omg.org/uml/](http://www.omg.org/uml/)

- Email

- [uml-rtf@omg.org](mailto:uml-rtf@omg.org)
- [ckobryn@acm.org](mailto:ckobryn@acm.org)

- Conferences & workshops

- UML World 2001, New York, June 11-14, 2001
- UML 2001, Toronto, Canada, Oct. 1-5, 2001
- OMG UML Workshop 2001, San Francisco, Dec. 3-6, 2001