Object-Oriented Software Engineering

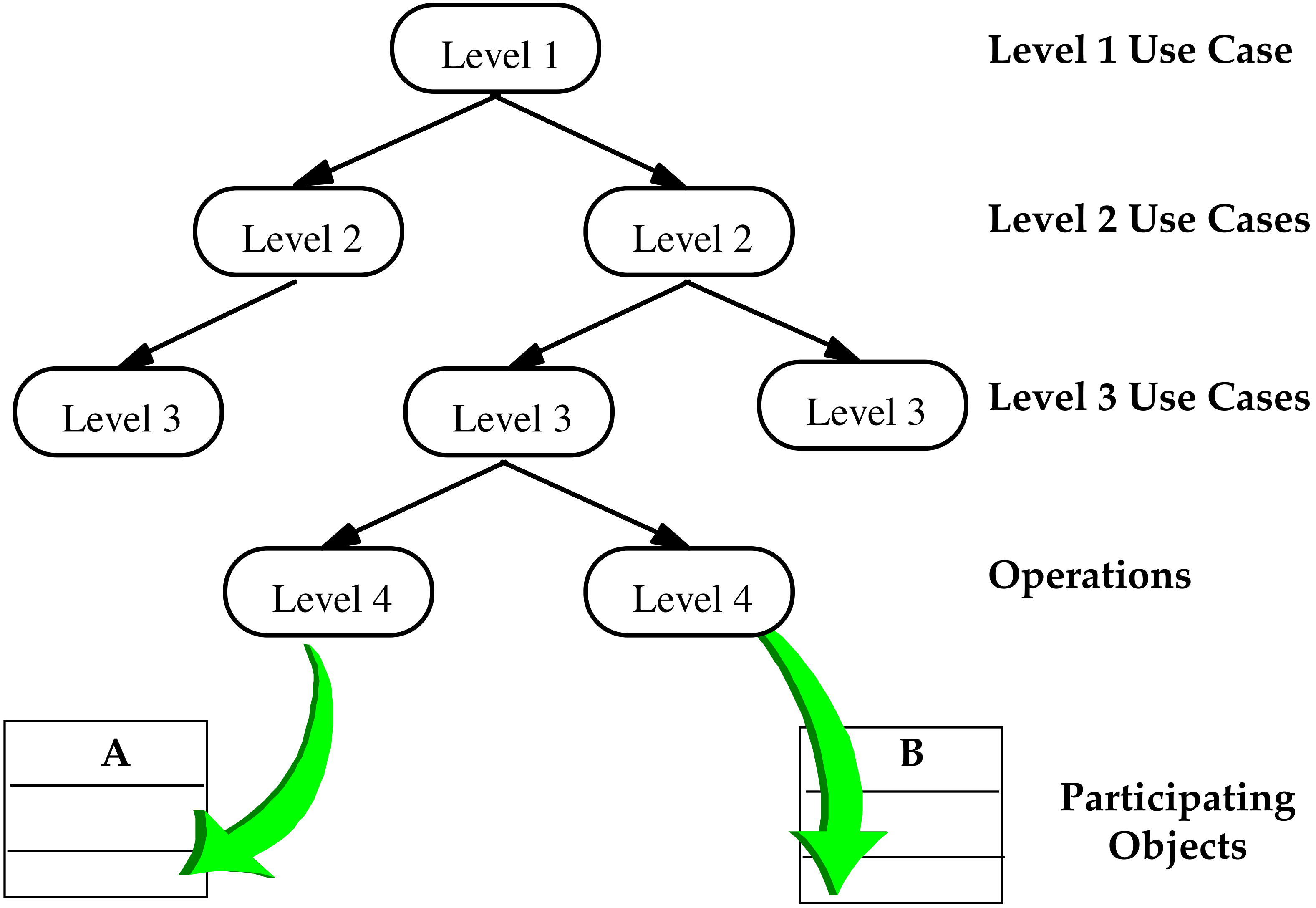
Using UML, Patterns, and Java

Chapter 6
Object Modeling

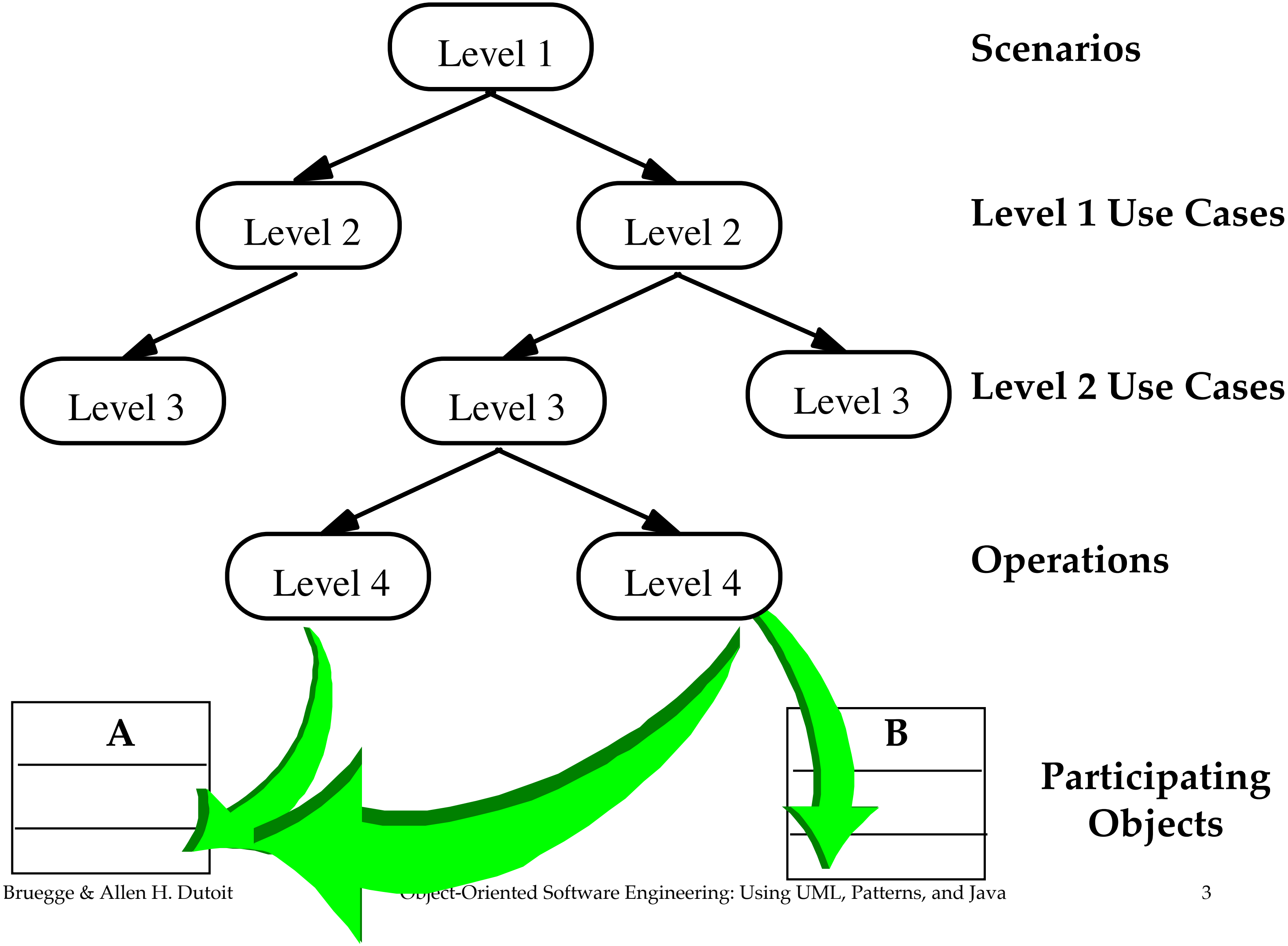
(Textbook Chapter 5)

Course instructor : TA.Nada Alamoudi

From Use Cases to Objects



From Use Cases to Objects: Why Functional Decomposition is not Enough



Activities during Object Modeling

Main goal: Find the important abstractions

- Steps during object modeling
  1. Class identification
     - Based on the fundamental assumption that we can find abstractions
  2. Find the attributes
  3. Find the methods
  4. Find the associations between classes
- Order of steps
  - Goal: get the desired abstractions
  - Order of steps secondary, only a heuristic
- What happens if we find the wrong abstractions?
  - We iterate and revise the model

Class Identification

Class identification is crucial to object-oriented modeling

- Helps to identify the important entities of a system

- Basic assumptions:
  1. We can find the classes for a new software system (Forward Engineering)
  2. We can identify the classes in an existing system (Reverse Engineering)

- Why can we do this?
  - Philosophy, science, experimental evidence.

Class Identification

- Approaches
  - Application domain approach
    - Ask application domain experts to identify relevant abstractions
  - Syntactic approach
    - Start with use cases
    - Analyze the text to identify the objects
    - Extract participating objects from flow of events
  - Design patterns approach
    - Use reusable design patterns
  - Component-based approach
    - Identify existing solution classes.

Class identification is a Hard Problem

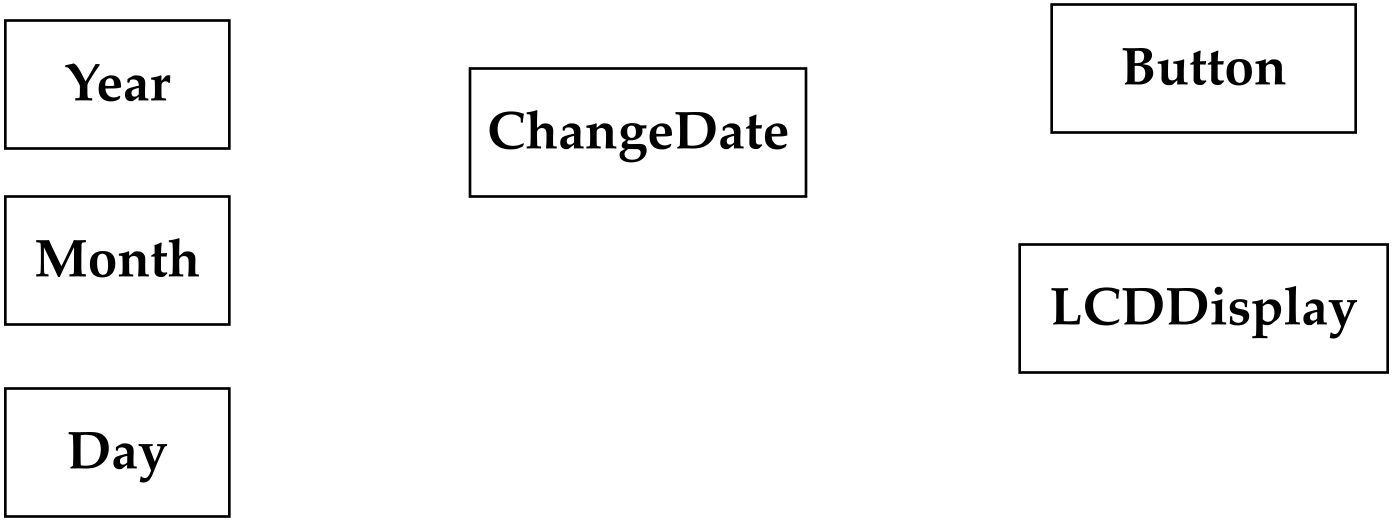
- One problem: Definition of the system boundary:
  - Which abstractions are outside, which abstractions are inside the system boundary?
    - Actors are outside the system
    - Classes/Objects are inside the system.
- An other problem: Classes/Objects are not just found by taking a picture of a scene or domain
  - The application domain has to be analyzed
  - Depending on the purpose of the system different objects might be found
    - How can we identify the purpose of a system?
    - Scenarios and use cases => Functional model

There are different types of Objects

- Entity Objects
  - Represent the persistent information tracked by the system (Application domain objects, also called "Business objects")
- Boundary Objects
  - Represent the interaction between the user and the system
- Control Objects
  - Represent the control tasks performed by the system.

Example: 2BWatch Modeling

To distinguish different object types in a model we can use the UML Stereotype mechanism



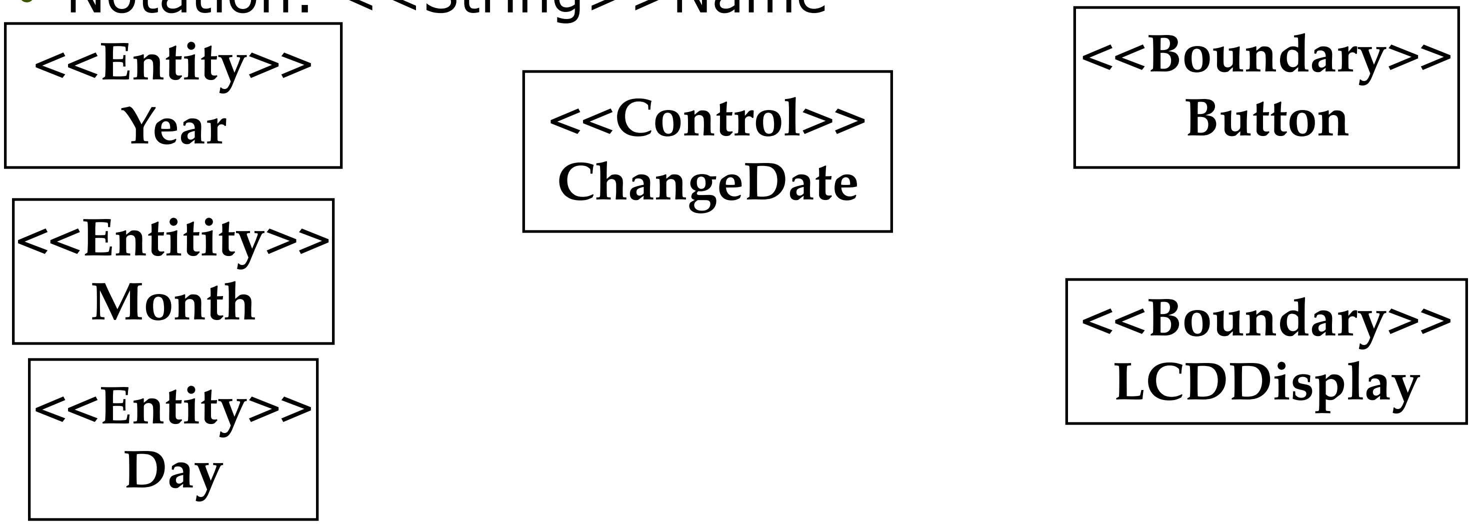
Entity Objects

Control Object

Boundary Objects

Naming Object Types in UML

- UML provides the stereotype mechanism to introduce new types of modeling elements
  - A stereotype is drawn as a name enclosed by angled double-quotes (“guillemets”) (<<, >>) and placed before the name of a UML element (class, method, attribute, ….)
  - Notation: <<String>>Name



<<Entity>> Year

<<Entitity>> Month

<<Entity>> Day

<<Control>> ChangeDate

<<Boundary>> Button

<<Boundary>> LCDDisplay

Entity Object

Control Object

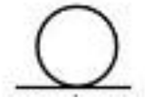
Boundary Object

UML is an Extensible Language

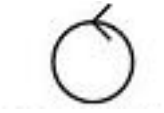
- Stereotypes allow you to extend the vocabulary of the UML so that you can create new model elements, derived from existing ones
- Examples:
  - Stereotypes can also be used to classify method behavior such as <<constructor>>, <<getter>> or <<setter>>
  - To indicate the interface of a subsystem or system, one can use the stereotype <<interface>> (Lecture System Design)
- Stereotypes can be represented with icons and graphics:
  - This can increase the readability of UML diagrams.

Icons for Stereotypes

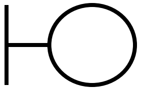
- One can use icons to identify a stereotype
  - When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



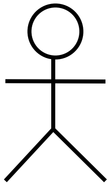
Entity Object



Control Object



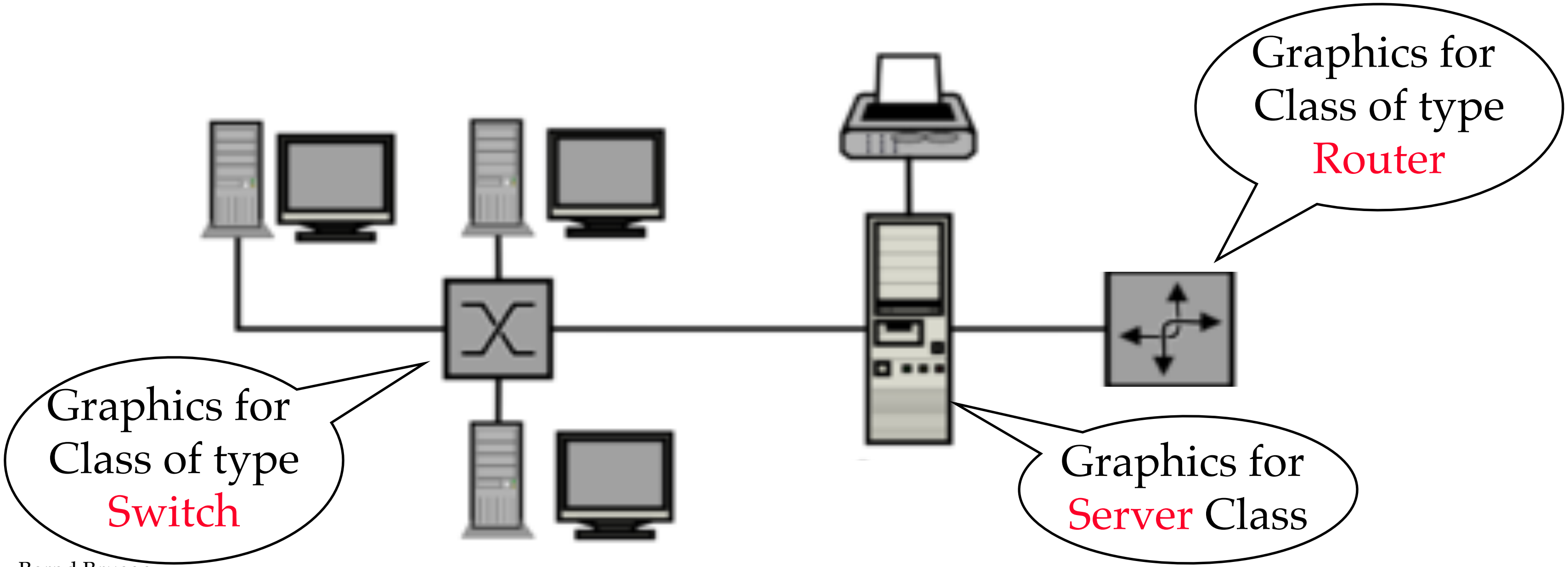
Boundary Object



Actor

Graphics for Stereotypes

- One can also use graphical symbols to identify a stereotype
    - When the stereotype is applied to a UML model element, the graphic replaces the default graphic for the diagram element.
- Example: When modeling a network, define graphics for representing classes of type Switch, Server, Router, Printer,etc.



Graphics for Class of type Router

Graphics for Class of type Switch

Graphics for Server Class

Pros and Cons of Stereotype Graphics

- Advantages:
  - UML diagrams may be easier to understand if they contain graphics and icons for stereotypes
    - This can increase the readability of the diagram, especially if the client is not trained in UML
    - And they are still UML diagrams!
- Disadvantages:
  - If developers are unfamiliar with the symbols being used, it can become much harder to understand what is going on
  - Additional symbols add to the burden of learning to read the diagrams.

Object Types allow us to deal with Change

- Having three types of object leads to models that are more resilient to change
  - The interface of a system changes more likely than the control
  - The way the system is controlled changes more likely than entities in the application domain
- Object types originated in Smalltalk:
  - Model, View, Controller (MVC)

Model <-> Entity Object

View <-> Boundary Object

Controller <-> Control Object

- Next topic: Finding objects.

Finding Participating Objects in Use Cases

- Pick a use case and look at flow of events
- Do a textual analysis (noun-verb analysis)
  - Nouns are candidates for objects/classes
  - Verbs are candidates for operations
  - This is also called Abbott's Technique

- After objects/classes are found, identify their types
  - Identify real world entities that the system needs to keep track of (FieldOfficer → Entity Object)
  - Identify real world procedures that the system needs to keep track of (EmergencyPlan → Control Object)
  - Identify interface artifacts (PoliceStation → Boundary Object).

Example for using the Technique

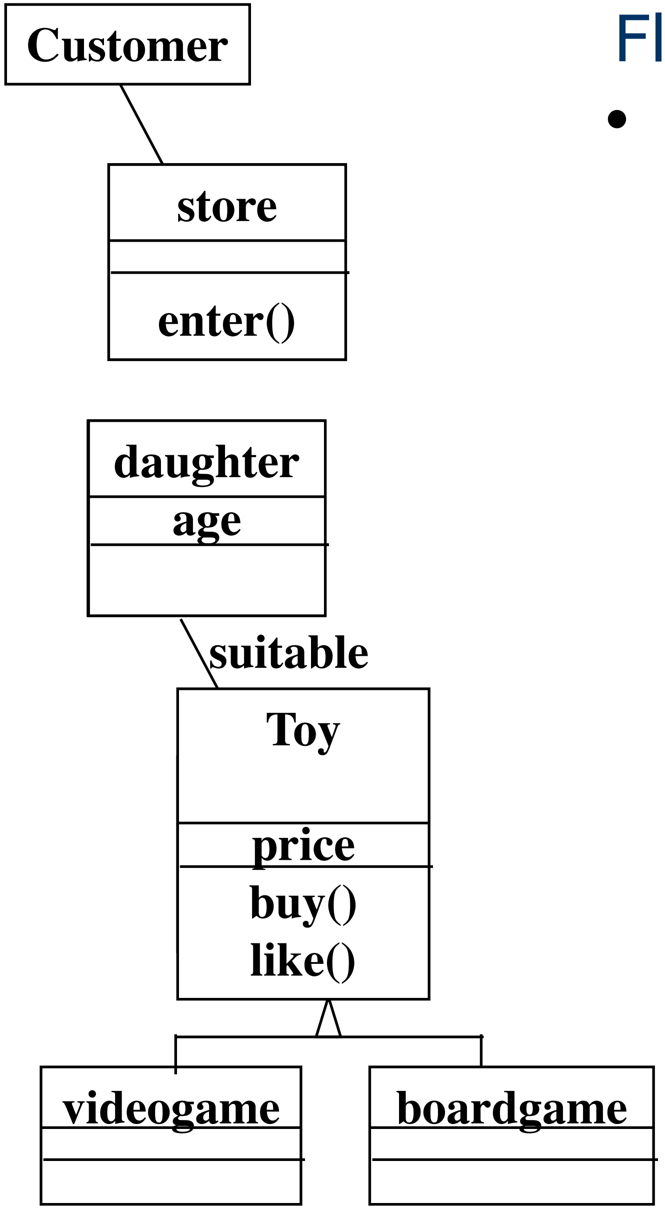
Flow of Events:

- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display. He likes it.
- An assistant helps him.
- The suitability of the game depends on the age of the child.
- His daughter is only 3 years old.
- The assistant recommends another type of toy, namely the boardgame “Monopoly".

Mapping parts of speech to model components (Abbot's Technique)

Example	*Part of speech*	*UML model component*
"Monopoly"	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
is-a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb	operation
depends on	intransitive verb	Constraint, class, association

Generating a Class Diagram from Flow of Events



Flow of events:

- The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

  An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store

Ways to find Objects

- Syntactical investigation with Abbot‘s technique:
  - Flow of events in use cases
  - Problem statement

- Use other knowledge sources:
  - Application knowledge: End users and experts know the abstractions of the application domain
  - Solution knowledge: Abstractions in the solution domain
  - General world knowledge: Your generic knowledge and intution

Order of Activities for Object Identification

1. Formulate a few scenarios with help from an end user or application domain expert
2. Extract the use cases from the scenarios, with the help of an application domain expert
3. Then proceed in parallel with the following:
   - Analyse the flow of events in each use case using Abbot's textual analysis technique
   - Generate the UML class diagram.

Steps in Generating Class Diagrams

1. Class identification (textual analysis, domain expert)
2. Identification of attributes and operations (sometimes before the classes are found!)
3. Identification of associations between classes
4. Identification of multiplicities
5. Identification of roles
6. Identification of inheritance

Who uses Class Diagrams?

- Purpose of class diagrams
  - The description of the static properties of a system
- The main users of class diagrams:
  - The application domain expert
    - uses class diagrams to model the application domain (including taxonomies)
      - during requirements elicitation and analysis
  - The developer
    - uses class diagrams during the development of a system
      - during analysis, system design, object design and implementation.

Who does not use Class Diagrams?

- The client and the end user are usually not interested in class diagrams
    - Clients focus more on project management issues
    - End users are more interested in the functionality of the system.

Pieces of an Object Model

- Classes and their instances (“objects”)
- Attributes
- Operations
- Associations between classes and objects

Associations

- Types of Associations
  - Canonical associations
    - Part-of Hierarchy (Aggregation)
    - Kind-of Hierarchy (Inheritance)
  - Generic associations

Attributes

- Detection of attributes is application specific
- Attributes in one system can be classes in another system
- Turning attributes to classes and vice versa

Operations

- Source of operations
  - Use cases in the functional model
  - General world knowledge
  - Generic operations: Get/Set
  - Design Patterns
  - Application domain specific operations
  - Actions and activities in the dynamic model

Object vs Class

- Object (instance): Exactly one thing
  - This lecture on object modeling
- A class describes a group of objects with similar properties
  - Game, Tournament, mechanic, car, database
- Object diagram: A graphical notation for modeling objects, classes and their relationships
  - Class diagram: Template for describing many instances of data. Useful for taxonomies, patters, schemata...
  - Instance diagram: A particular set of objects relating to each other. Useful for discussing scenarios, test cases and examples

Developers have different Views on Class Diagrams

- According to the development activity, a developer plays different roles:
  - Analyst
  - System Designer
  - Object Designer
  - Implementor

- Each of these roles has a different view about the class diagram (the object model).

The View of the Analyst

- The analyst is interested
  - in application classes: The associations between classes are relationships between abstractions in the application domain
  - operations and attributes of the application classes (difference to E/R models!)
- The analyst uses inheritance in the model to reflect the taxonomies in the application domain
  - Taxonomy: An is-a-hierarchy of abstractions in an application domain
- The analyst is not interested
  - in the exact signature of operations
  - in solution domain classes

The View of the Designer

- The designer focuses on the solution of the problem, that is, the solution domain
- The associations between classes are now references (pointers) between classes in the application or solution domain
- An important design task is the specification of interfaces:
  - The designer describes the interface of classes and the interface of subsystems
  - Subsystems originate from modules (term often used during analysis):
    - Module: a collection of classes
    - Subsystem: a collection of classes with an interface
- Subsystems are modeled in UML with a package.

Goals of the Designer

- The most important design goals for the designer are design usability and design reusability
- Design usability: the interfaces are usable from as many classes as possible within in the system
- Design reusability: The interfaces are designed in a way, that they can also be reused by other (future) software systems

    => Class libraries

    => Frameworks

    => Design patterns.

The View of the Implementor

- Class implementor
  - Must realize the interface of a class in a programming language
  - Interested in appropriate data structures (for the attributes) and algorithms (for the operations)
- Class extender
  - Interested in how to extend a class to solve a new problem or to adapt to a change in the application domain
- Class user
  - The class user is interested in the signatures of the class operations and conditions, under which they can be invoked
  - The class user is not interested in the implementation of the class.

Why do we distinguish different Users of Class Diagrams?

- Models often don�much distinguish between application classes and solution classes
    - Reason: Modeling languages like UML allow the use of both types of classes in the same model
        - “address book‟, “array"
    - Preferred: No solution classes in the analysis model
- Many systems don‵t distinguish between the specification and the implementation of a class
    - Reason: Object-oriented programming languages allow the simultaneous use of specification and implementation of a class
    - Preferred: We distinguish between analysis model and object design model. The analysis design model does not contain any implementation specification.

Analysis Model vs. Object Design model

- The analysis model is constructed during the analysis phase
  - Main stake holders: End user, customer, analyst
  - The class diagrams contains only application domain classes

- The object design model (sometimes also called specification model) is created during the object design phase
  - Main stake holders: class specifiers, class implementors, class users and class extenders
  - The class diagrams contain application domain as well as solution domain classes.

Analysis Model vs Object Design Model (2)

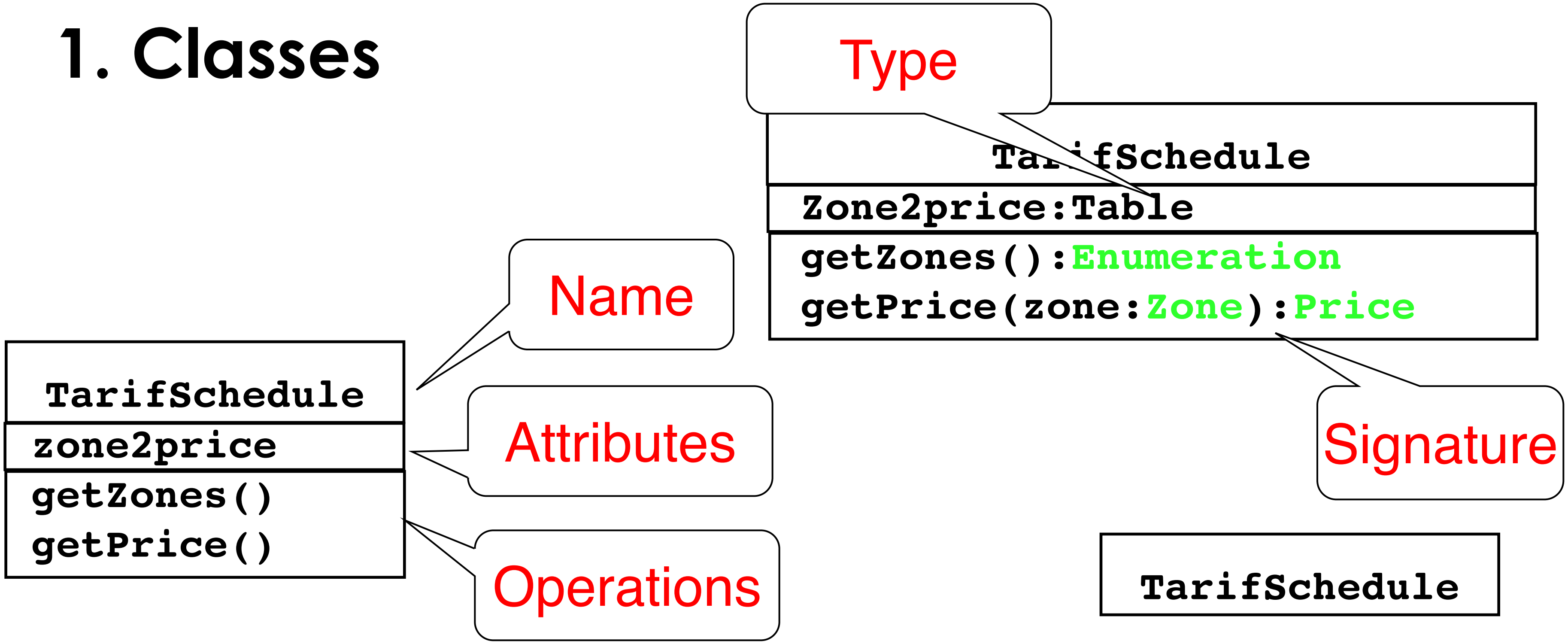
- The analysis model is the basis for communication between analysts, application domain experts and end users.

- The object design model is the basis for communication between designers and implementors.

Class Diagrams

- Class diagrams represent the structure of the system
- Used during:
  - requirements analysis to model application domain concepts
  - system design to model subsystems
  - object design to specify the detailed behavior and attributes of classes

1. Classes

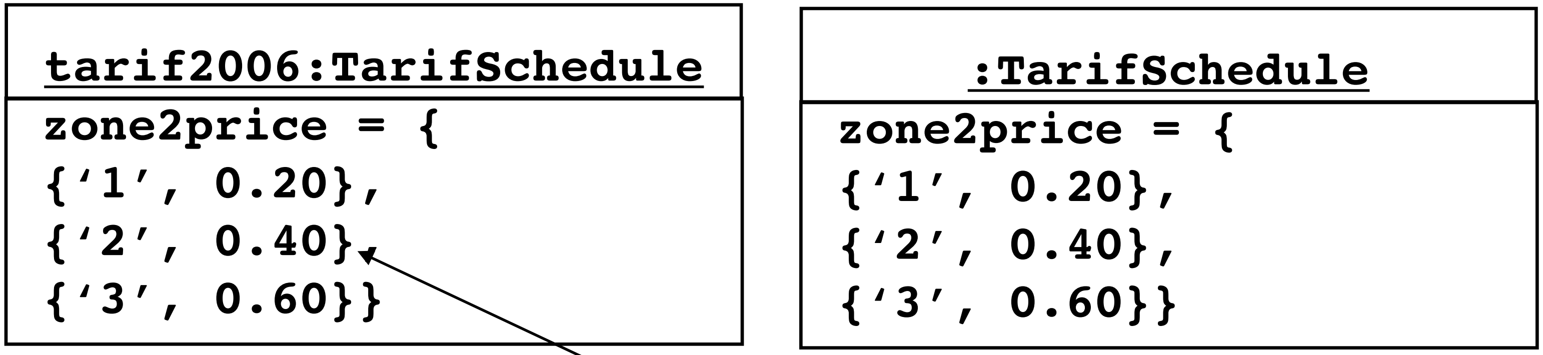


- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

  Each attribute has a ***type***
  Each operation has a ***signature***

The class name is the only mandatory information

Instances



<u>tarif2006:TarifSchedule</u>

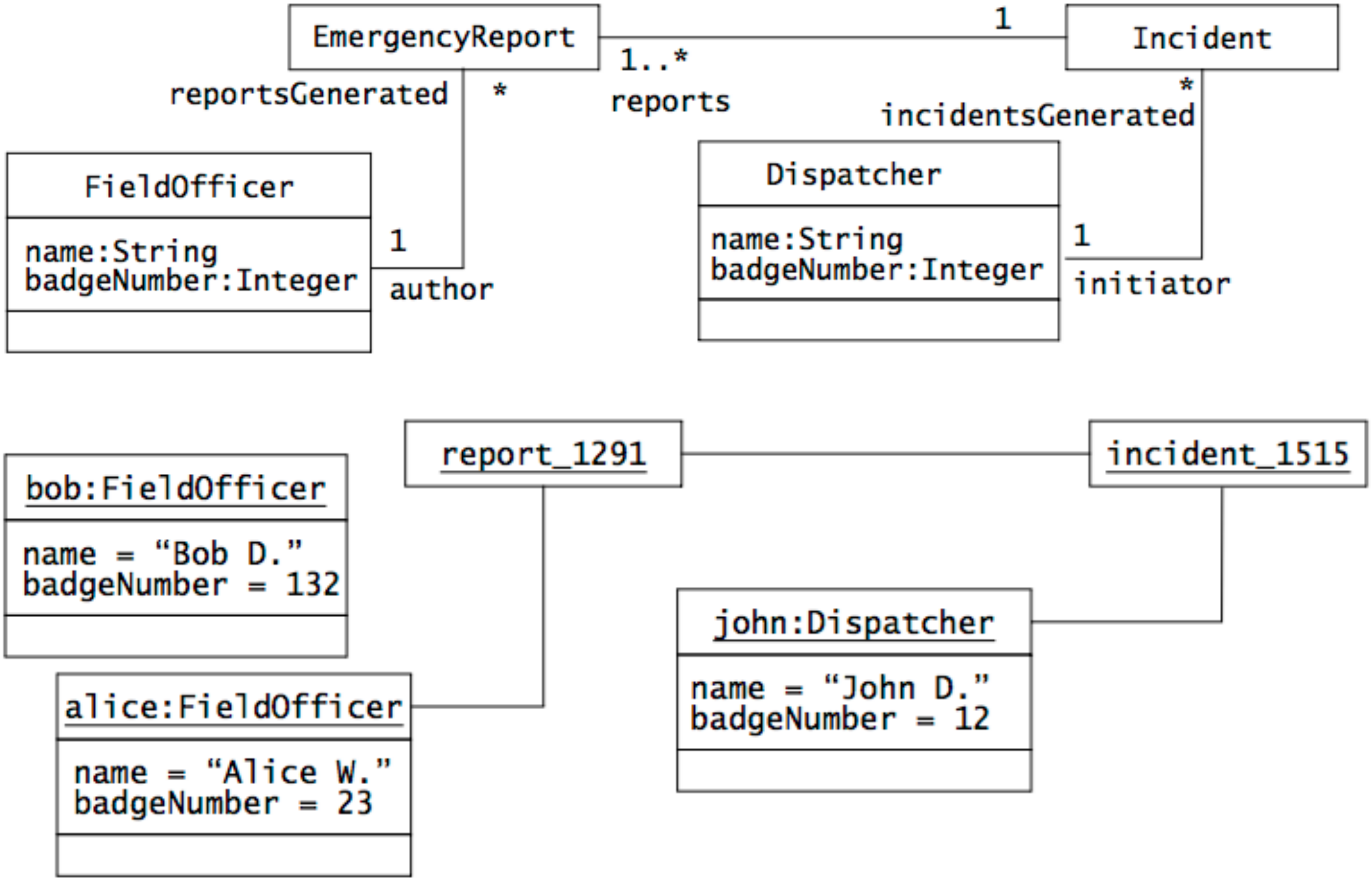
```
zone2price = {
{‘1’, 0.20},
{‘2’, 0.40},
{‘3’, 0.60}}
```

<u>:TarifSchedule</u>

```
zone2price = {
{‘1’, 0.20},
{‘2’, 0.40},
{‘3’, 0.60}}
```

- An ***instance*** represents a phenomenon
- The attributes are represented with their ***values***
- The name of an instance is <u>underlined</u>
- The name can contain only the class name of the instance (anonymous instance)

Classes and Objects—another example



Actor vs Class vs Object

Actor

- An entity outside the system to be modeled, interacting with the system (“Passenger”)

Class

- An abstraction modeling an entity in the application or solution domain
- The class is part of the system model (“User”, “Ticket distributor”, “Server”)

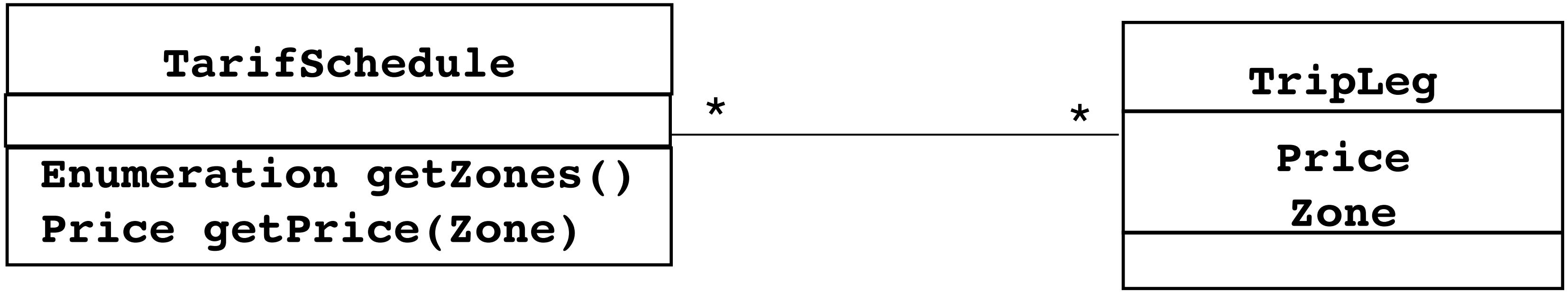
Object

- A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

2. Associations and links

- Associations are relationships between classes
- A link represents a connection between two objects.
- For example, each FieldOfficer object also has a list of EmergencyReports that has been written by the FieldOfficer.

Associations



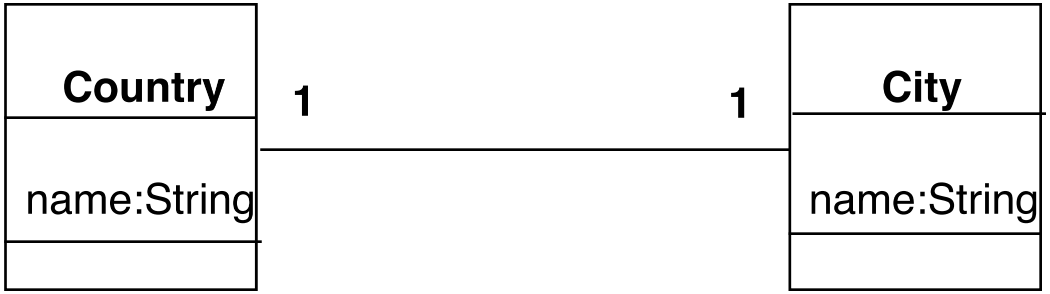
Associations denote relationships between classes

The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

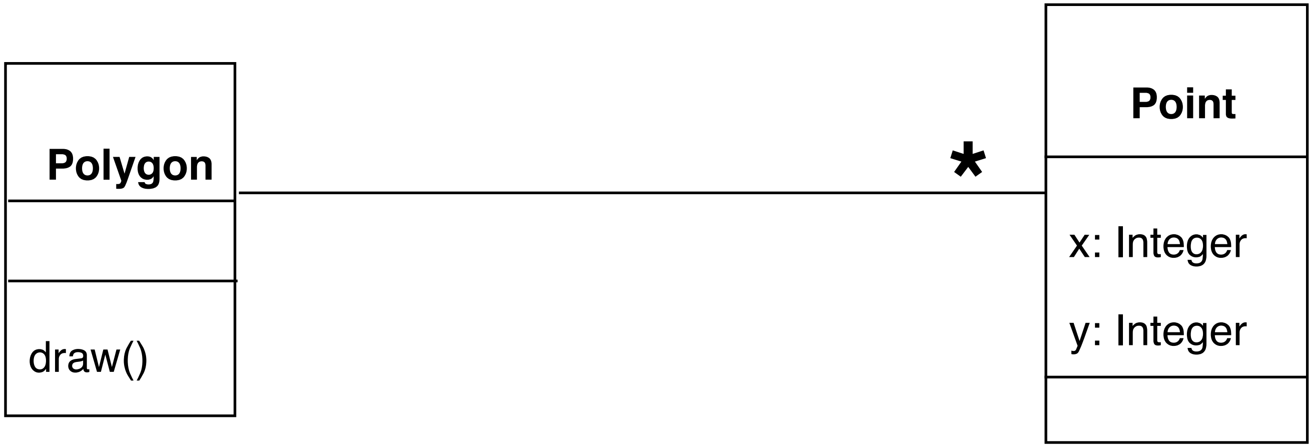
Multiplicity

- Each end of an association can be labeled by a set of integers indicating the number of links that can legitimately originate from an instance of the class connected to the association end.
- This set of integers is called the multiplicity of the association end
- The “many” multiplicity is shorthand for 0..n and is shown as a star.
- An association end can have an arbitrary set of integers as a multiplicity.
- In practice, most of the associations we encounter belong to one of the following three types.

Multiplicity: 1-to-1 and 1-to-many Associations

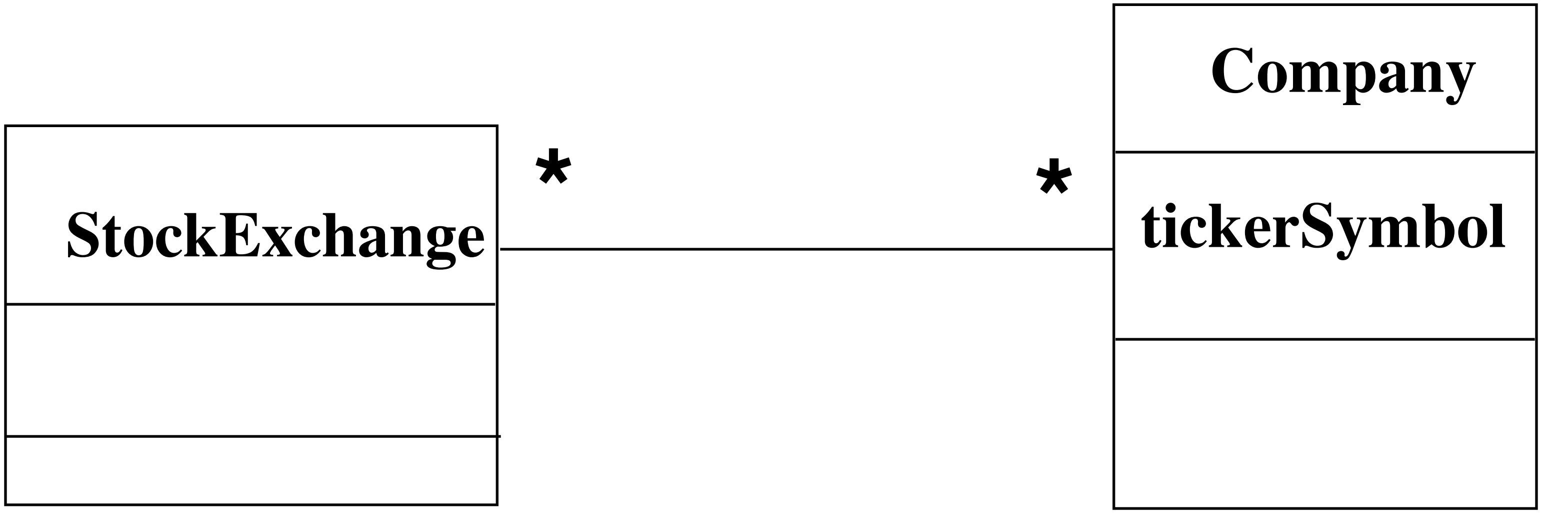


1-to-1 association

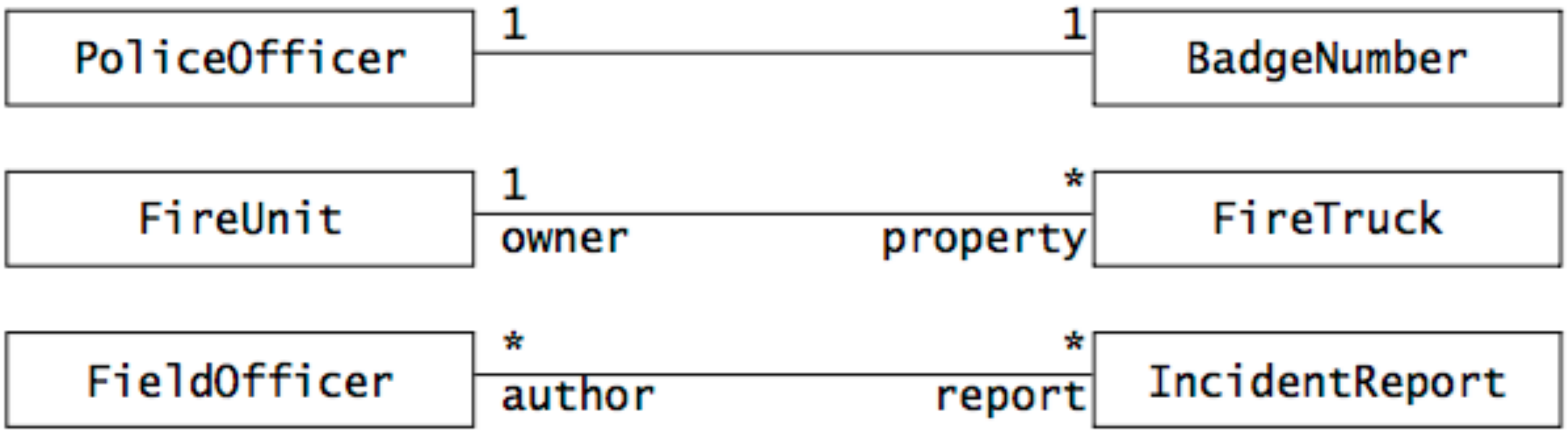


1-to-many association

Multiplicity: Many-to-Many Associations



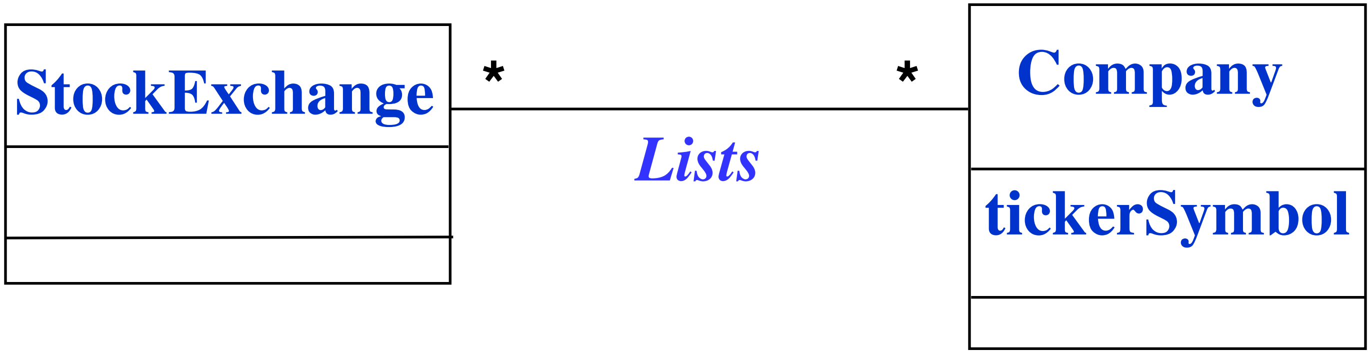
Multiplicity—Another Example



From Problem Statement To Object Model

Problem Statement: A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol

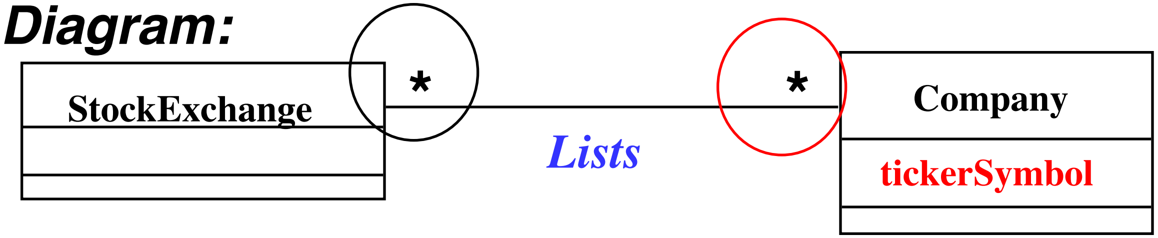
Class Diagram:



From Problem Statement to Code

Problem Statement : A stock exchange lists many companies. Each company is identified by a ticker symbol

Class Diagram:



Java Code

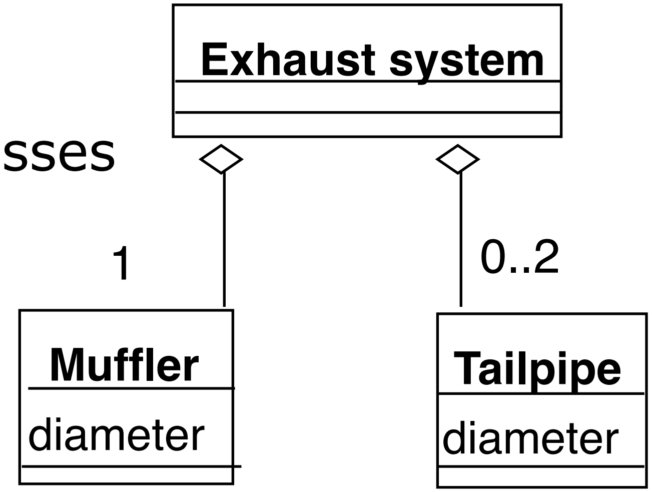
```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

public class Company
{
 public int m_tickerSymbol;
 private Vector m_StockExchange = new Vector();
};
```

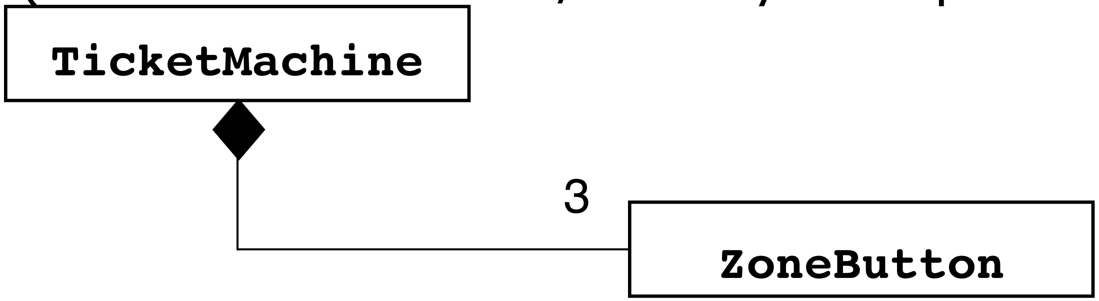
Associations are mapped to Attributes!

Aggregation

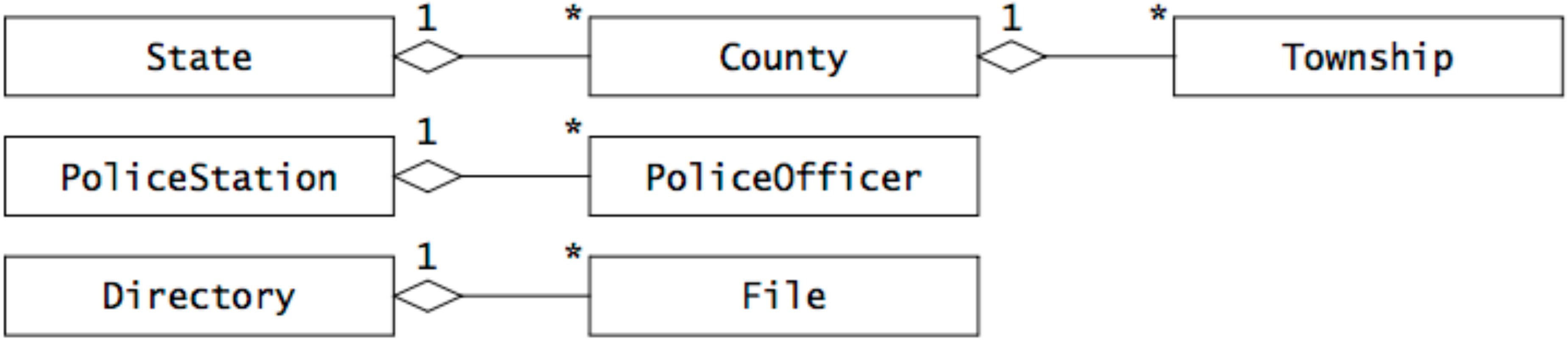
- An *aggregation* is a special case of association denoting a “consists-of” hierarchy
- The *aggregate* is the parent class, the components are the children classes



A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don’t exist on their won (“the whole controls/destroys the parts”)

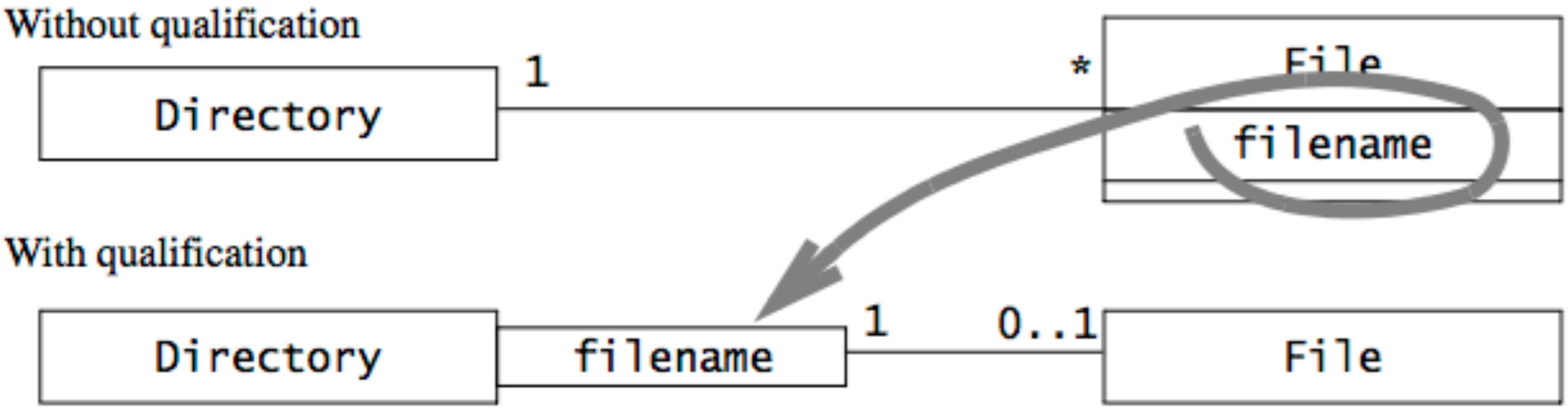


Aggregation—Another Example



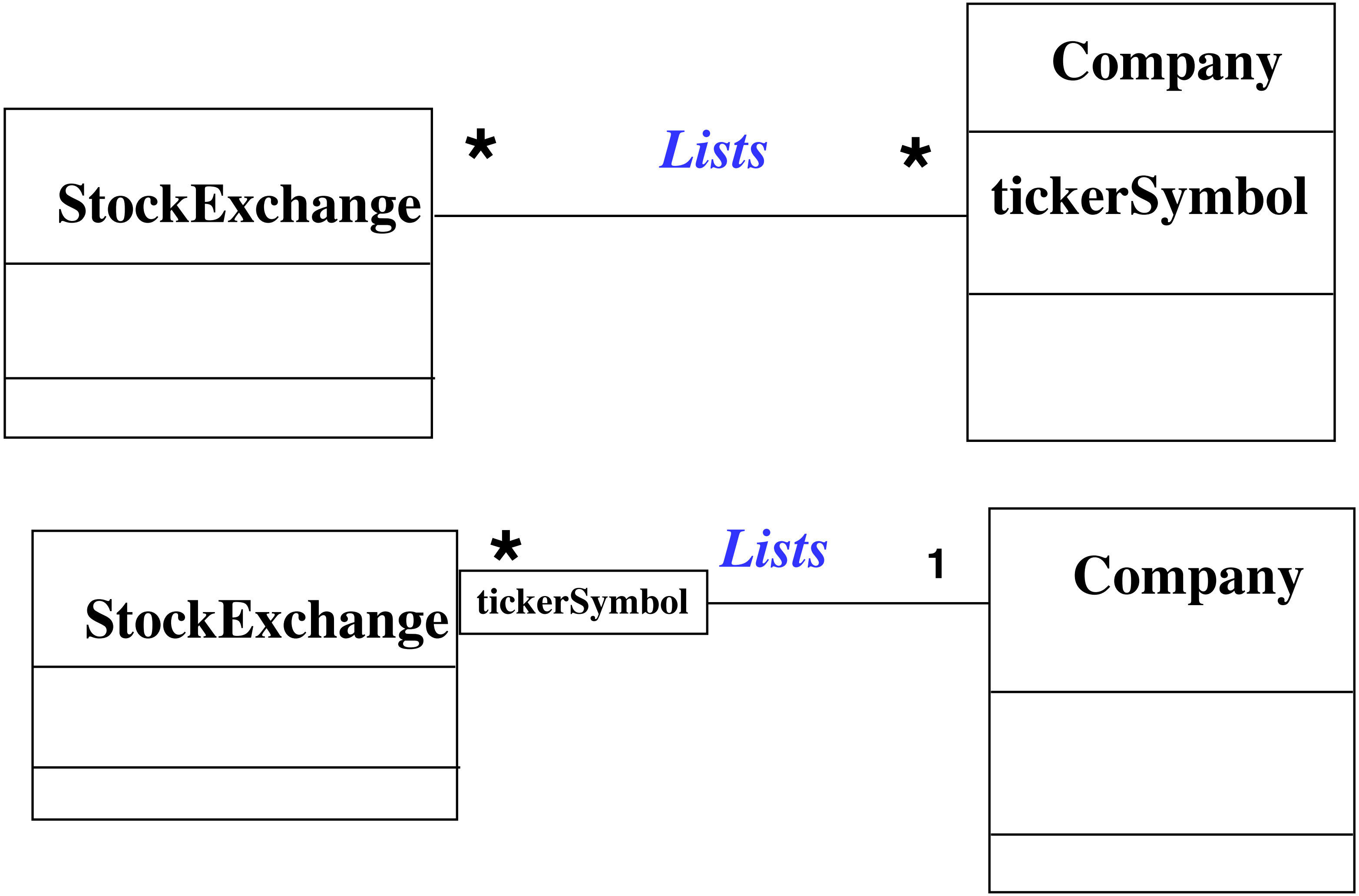
Qualifiers

Qualification is a technique for reducing multiplicity by using keys.

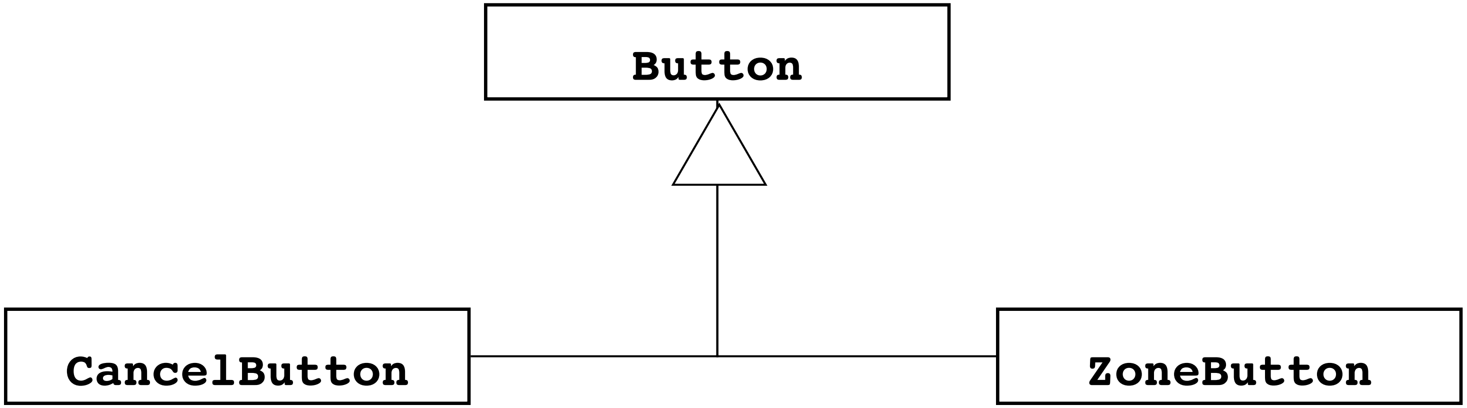


The relationship between Directory and File is called a qualified association.

Qualification: Another Example



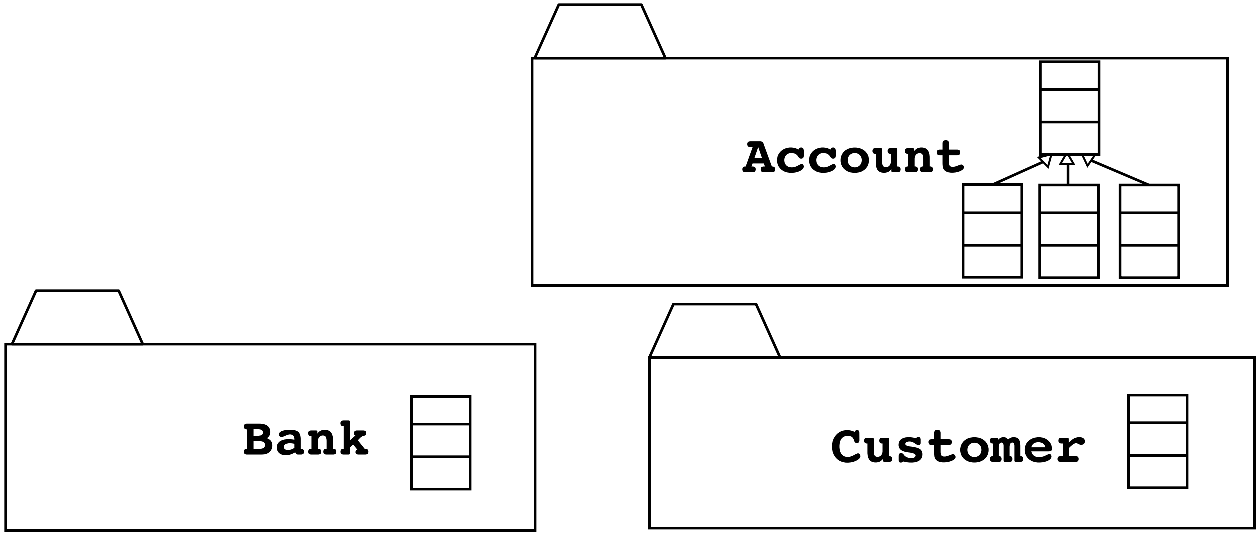
Inheritance



- *Inheritance* is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The **children classes** inherit the attributes and operations of the **parent class.**

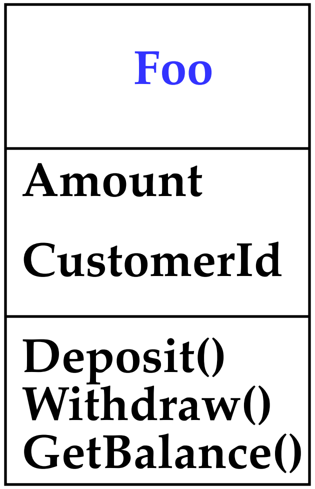
Packages

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems



- Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package

Object Modeling in Practice



Foo

Amount

CustomerId

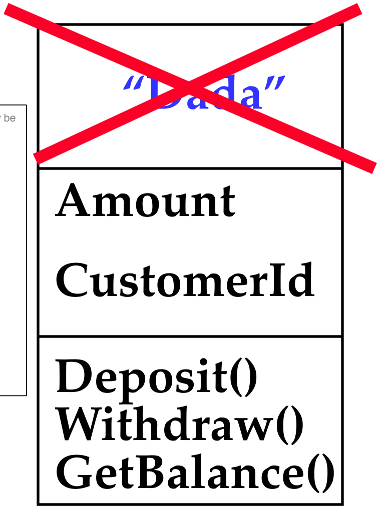
Deposit()
Withdraw()
GetBalance()

Class Identification: Name of Class, Attributes and Methods

Is Foo the right name?

Object Modeling in Practice: Brainstorming

This image cannot currently be displayed.

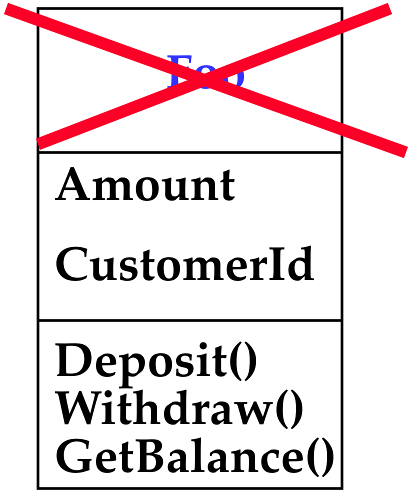


~~"Dada"~~

Amount

CustomerId

Deposit()
Withdraw()
GetBalance()



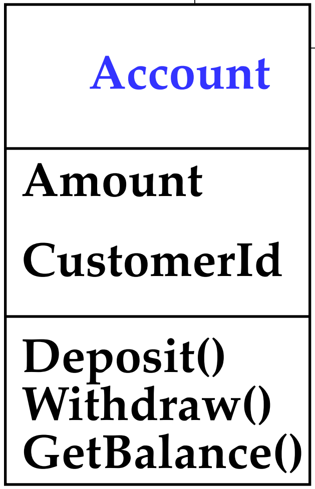
~~Foo~~

Amount

CustomerId

Deposit()
Withdraw()
GetBalance()

This image cannot currently be displayed.



Account

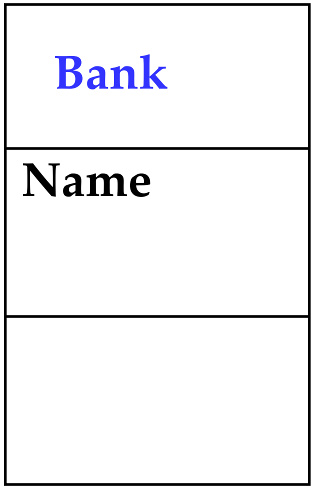
Amount

CustomerId

Deposit()
Withdraw()
GetBalance()

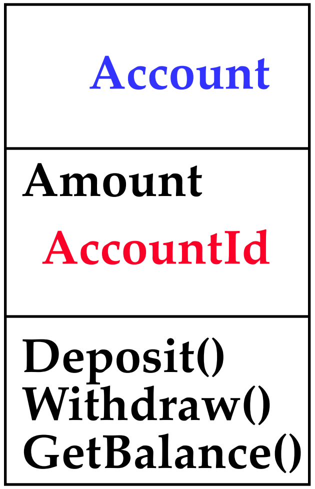
Is Foo the right name?

Object Modeling in Practice: More classes



Bank

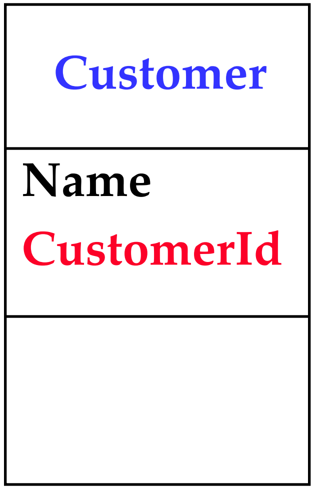
Name



Account

Amount
AccountId

Deposit()
Withdraw()
GetBalance()



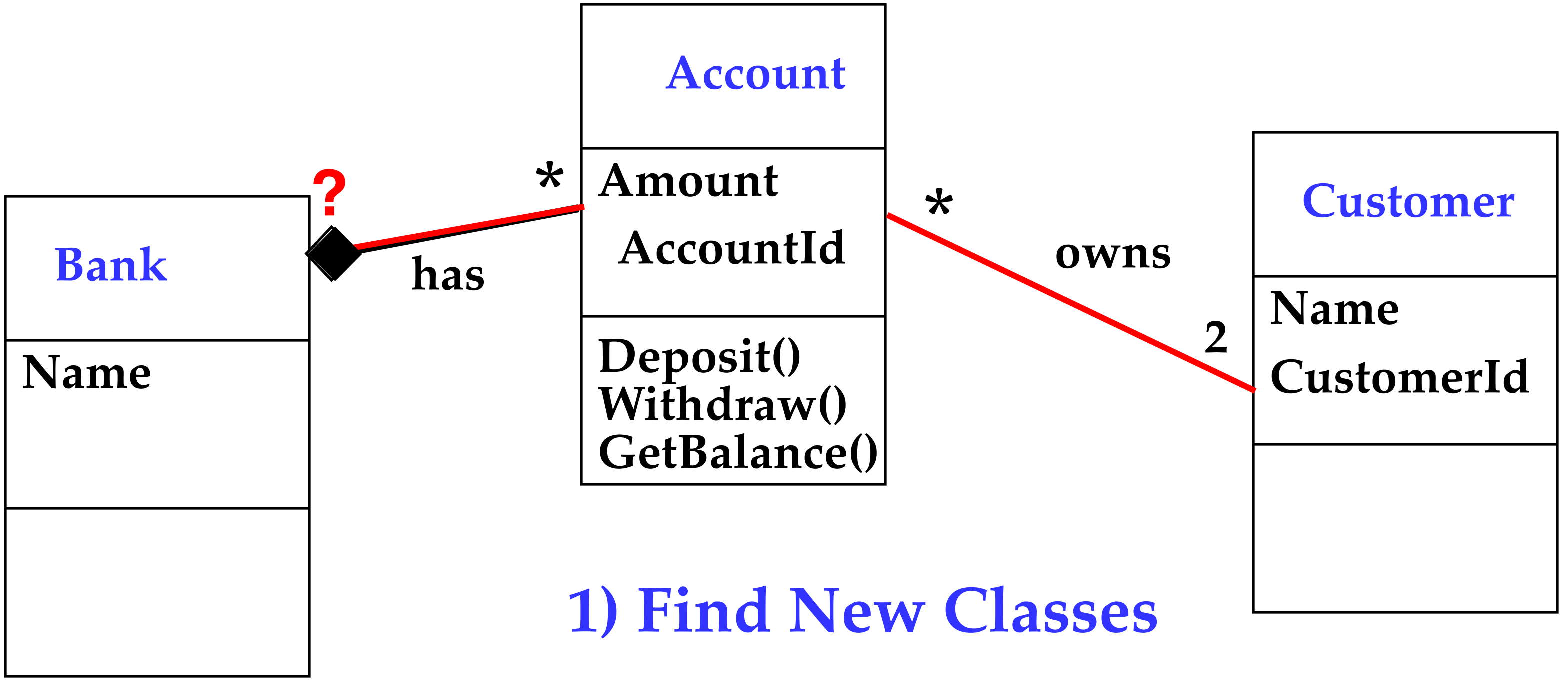
Customer

Name
CustomerId

1) Find New Classes

2) Review Names, Attributes and Methods

Object Modeling in Practice: Associations



1) Find New Classes

2) Review Names, Attributes and Methods

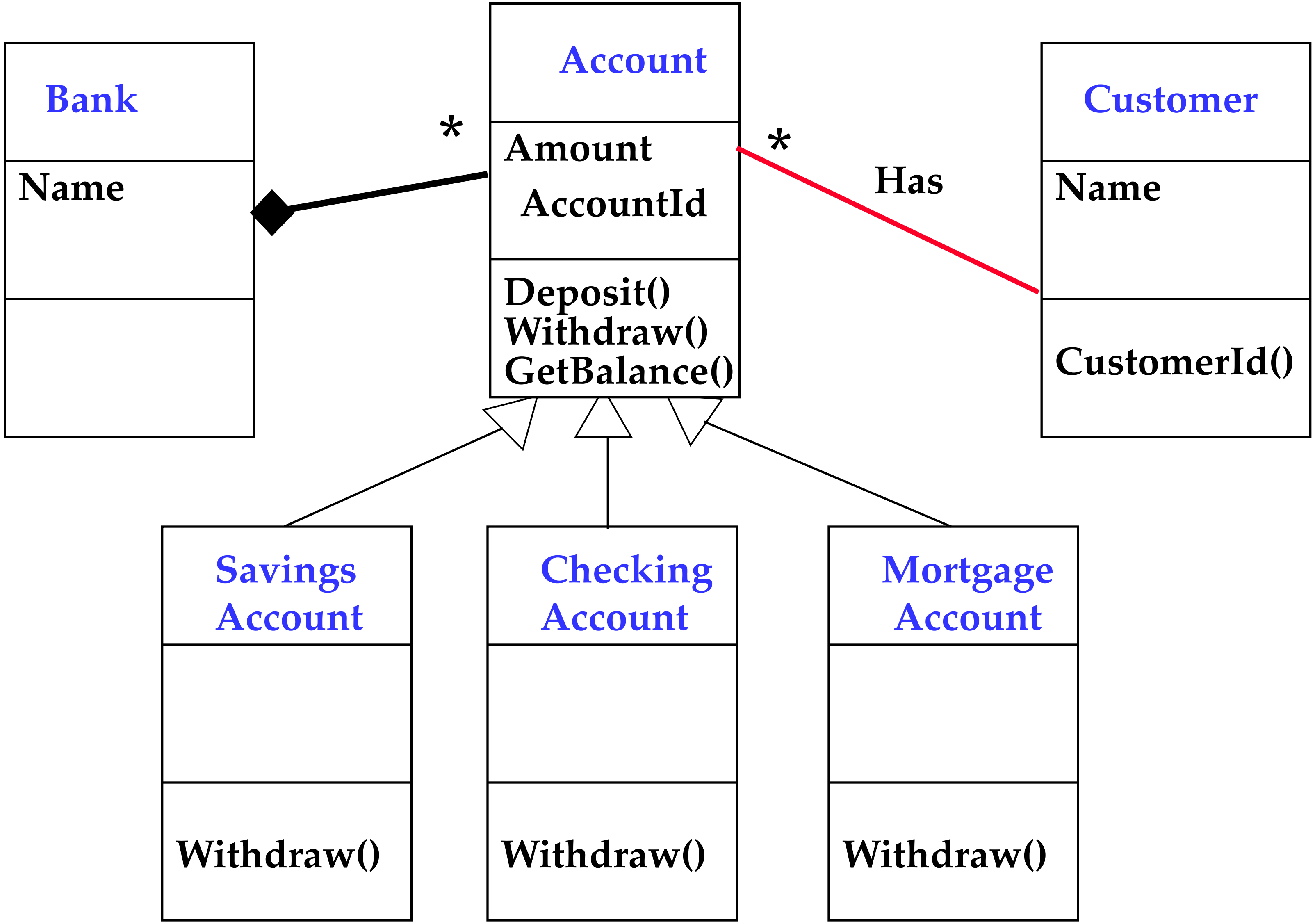
3) Find Associations between Classes

4) Label the generic associations

5) Determine the multiplicity of the assocations

6) Review associations

Practice Object Modeling: Find Taxonomies



Bank

Name

Account

Amount
AccountId

Deposit()
Withdraw()
GetBalance()

Customer

Name

CustomerId()

Has

Savings Account

Withdraw()

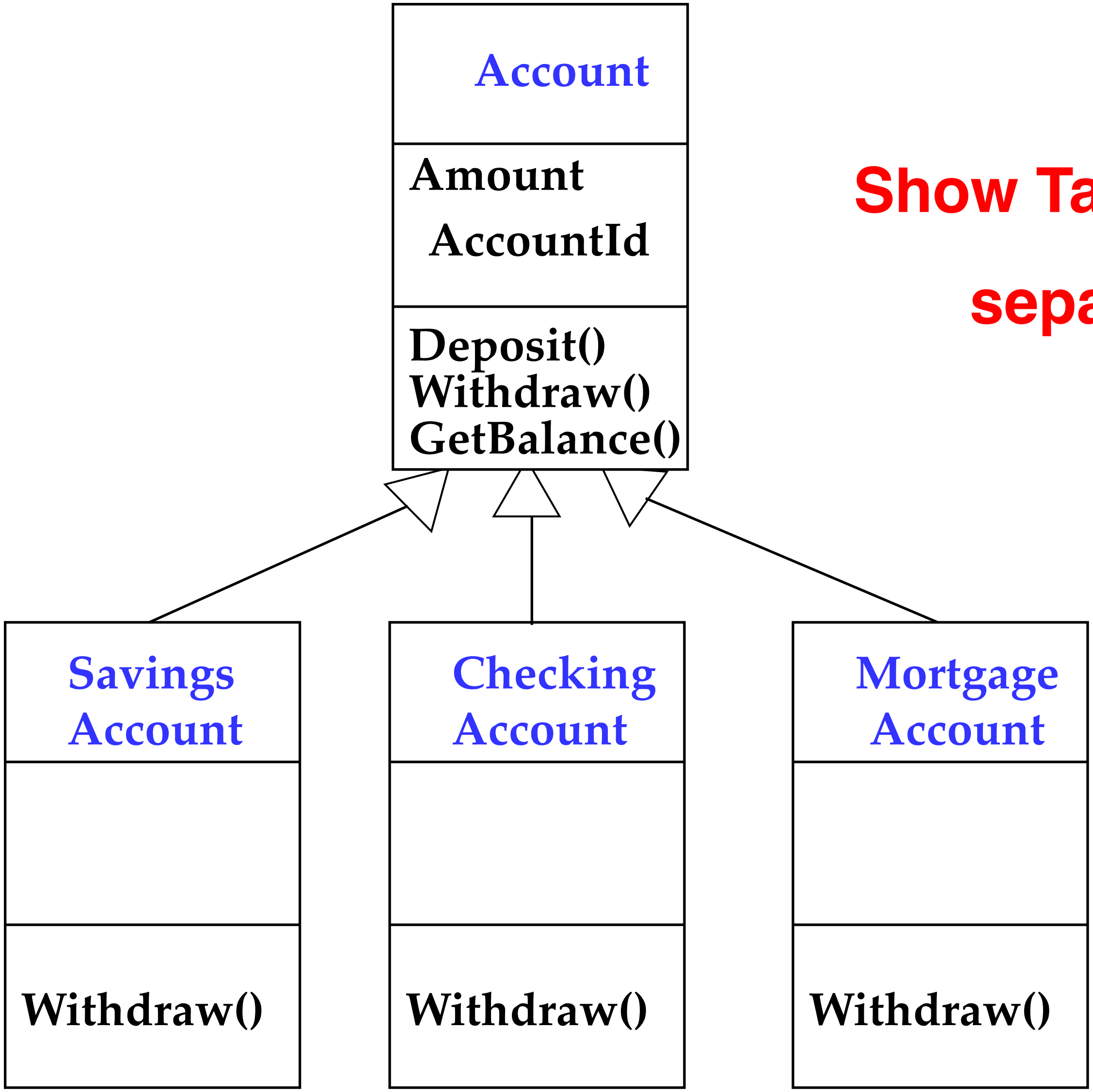
Checking Account

Withdraw()

Mortgage Account

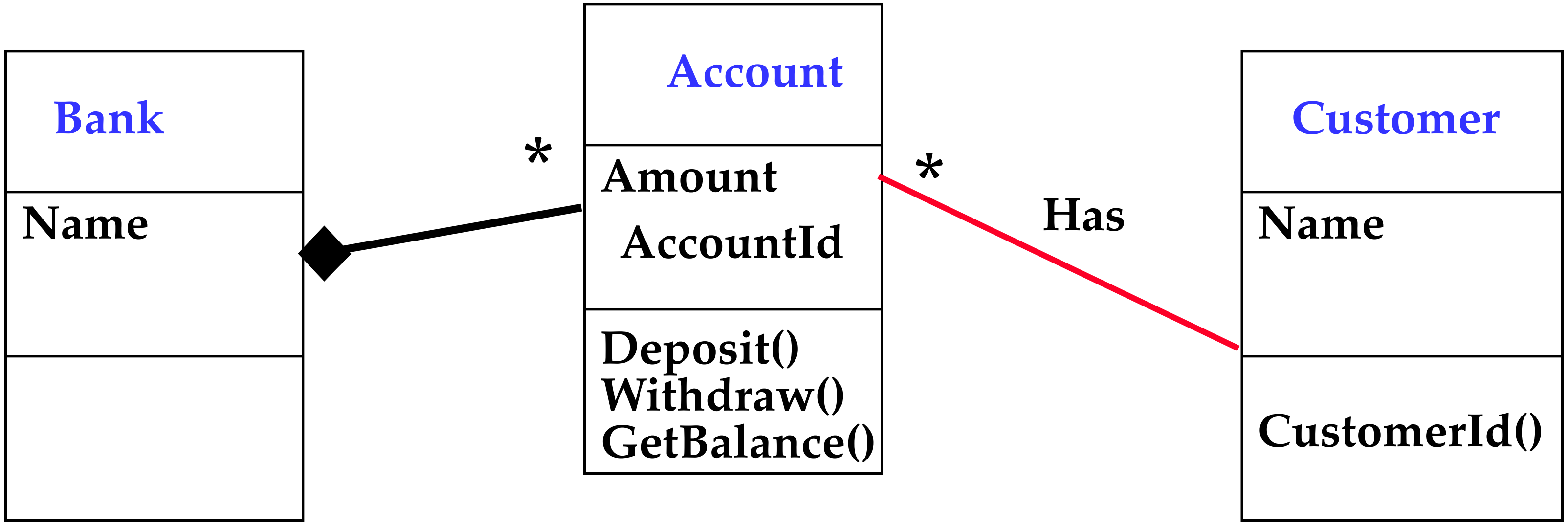
Withdraw()

Practice Object Modeling: Simplify, Organize



Show Taxonomies separately

Practice Object Modeling: Simplify, Organize



Use the 7±2 heuristics

or better yet, 5±2!