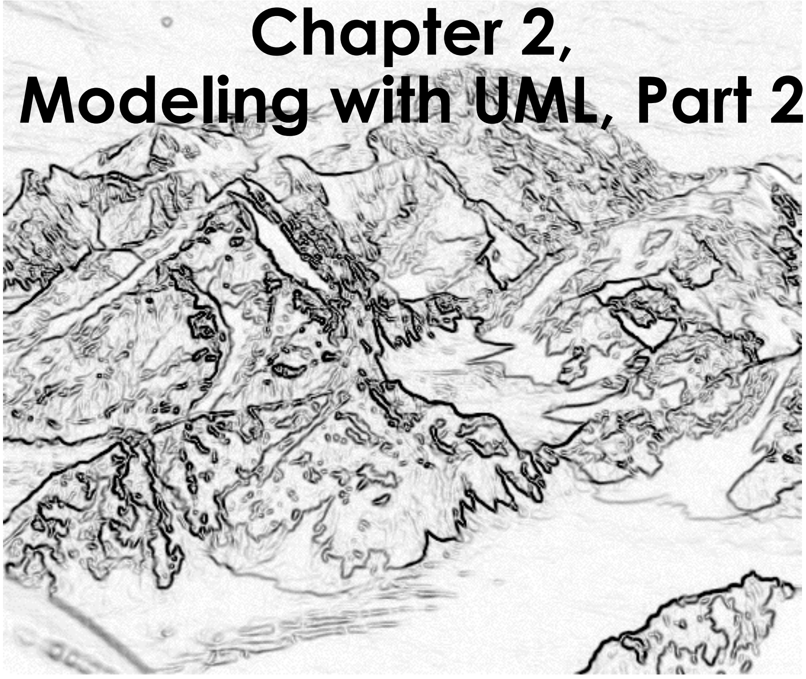
Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 2, Modeling with UML, Part 2



Outline of this Class

- Use case diagrams
  - Describe the functional behavior of the system as seen by the user
- Class diagrams
  - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the dynamic behavior between objects of the system
- Statechart diagrams
  - Describe the dynamic behavior of an individual object
- Activity diagrams
  - Describe the dynamic behavior of a system, in particular the workflow.

What is UML? Unified Modeling Language

- Convergence of different notations used in object-oriented methods, mainly
  - OMT (James Rumbaugh and collegues), OOSE (Ivar Jacobson), Booch (Grady Booch)
- They also developed the Rational Unified Process, which became the Unified Process in 1999

25 year at GE Research, where he developed OMT, joined (IBM) Rational in 1994, CASE tool OMTool

At Ericsson until 1994, developed use cases and the CASE tool Objectory, at IBM Rational since 1995, http://www.ivarjacobson.com

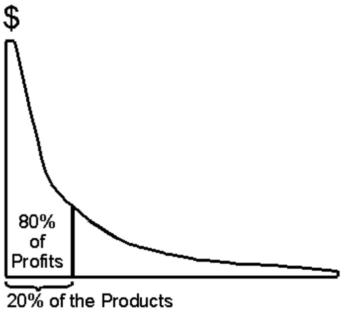
Developed the Booch method ("clouds"), ACM Fellow 1995, and IBM Fellow 2003 http://www.booch.com/

UML

- Nonproprietary standard for modeling systems
- Current Version: UML 2.2
  - Information at the OMG portal http://www.uml.org/
- Commercial tools:
  - Rational (IBM),Together (Borland), Visual Architect (Visual Paradigm), Enterprise Architect (Sparx Systems)
- Open Source tools http://www.sourceforge.net/
  - ArgoUML, StarUML, Umbrello (for KDE), PoseidonUML
- Example of research tools: Unicase, Sysiphus
  - Based on a unified project model for modeling, collaboration and project organization
  - http://unicase.org
  - http://sysiphus.in.tum.de/

UML: First Pass

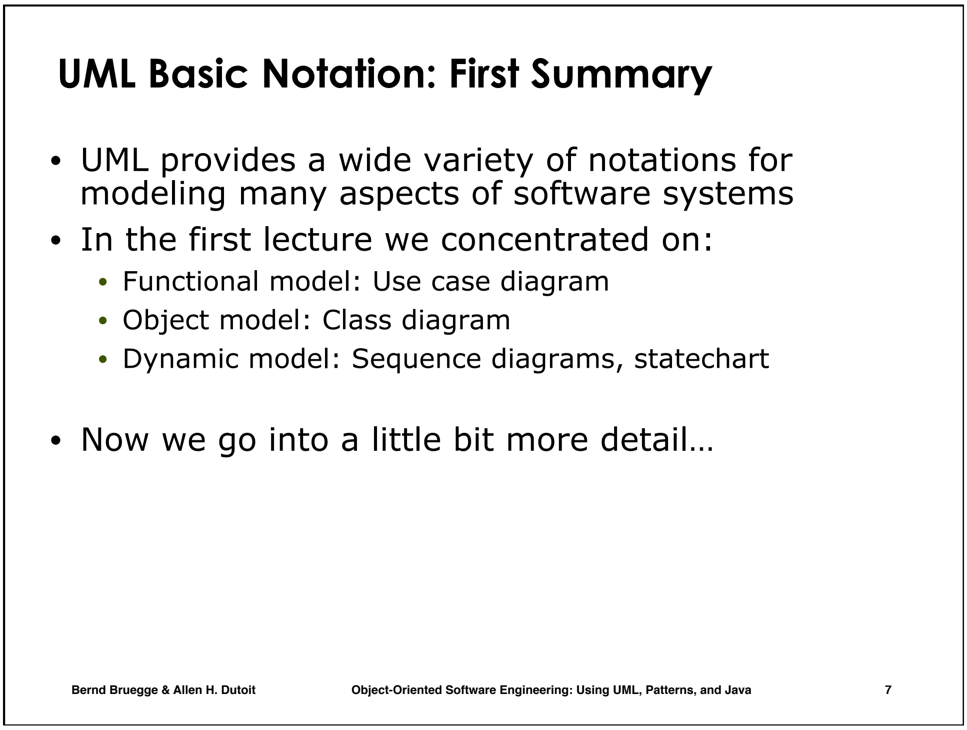
- You can solve 80% of the modeling problems by using 20 % UML
- We teach you those 20%
- 80-20 rule: Pareto principle



Vilfredo Pareto, 1848-1923
Introduced the concept of Pareto Efficiency,
Founder of the field of microeconomics.

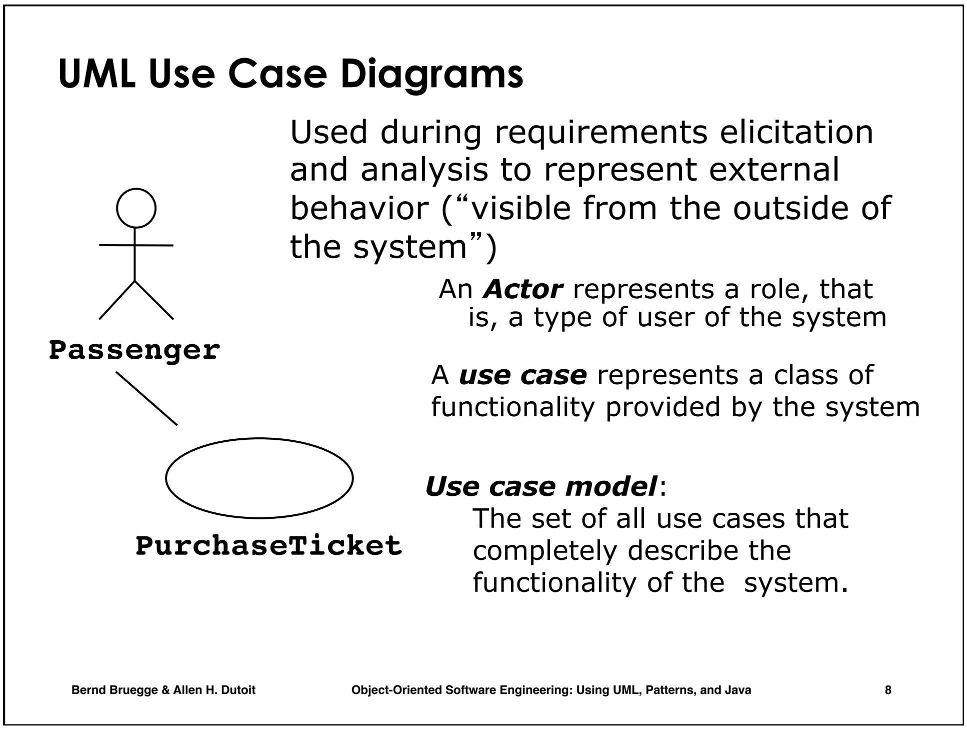
UML First Pass (covered in Last Lecture)

- Use case diagrams
  - Describe the functional behavior of the system as seen by the user
- Class diagrams
  - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the dynamic behavior between objects of the system
- Statechart diagrams
  - Describe the dynamic behavior of an individual object
- Activity diagrams
  - Describe the dynamic behavior of a system, in particular the workflow.



UML Basic Notation: First Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- In the first lecture we concentrated on:
  - Functional model: Use case diagram
  - Object model: Class diagram
  - Dynamic model: Sequence diagrams, statechart
- Now we go into a little bit more detail…



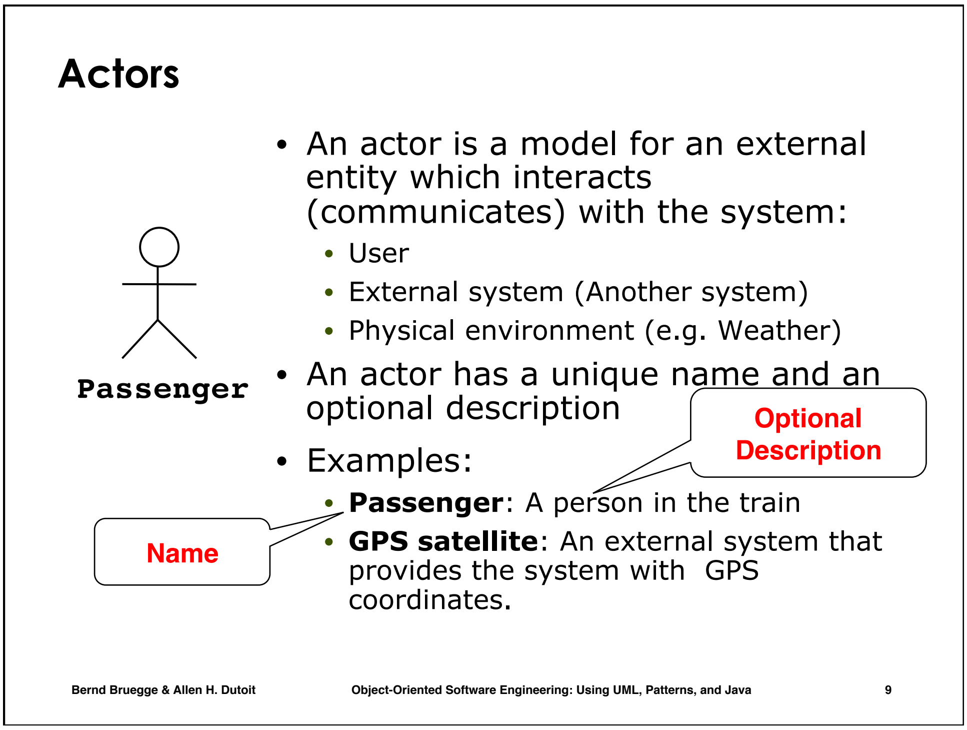
UML Use Case Diagrams

Used during requirements elicitation and analysis to represent external behavior (“visible from the outside of the system”)

An ***Actor*** represents a role, that is, a type of user of the system

A ***use case*** represents a class of functionality provided by the system

Use case model:
The set of all use cases that completely describe the functionality of the system.

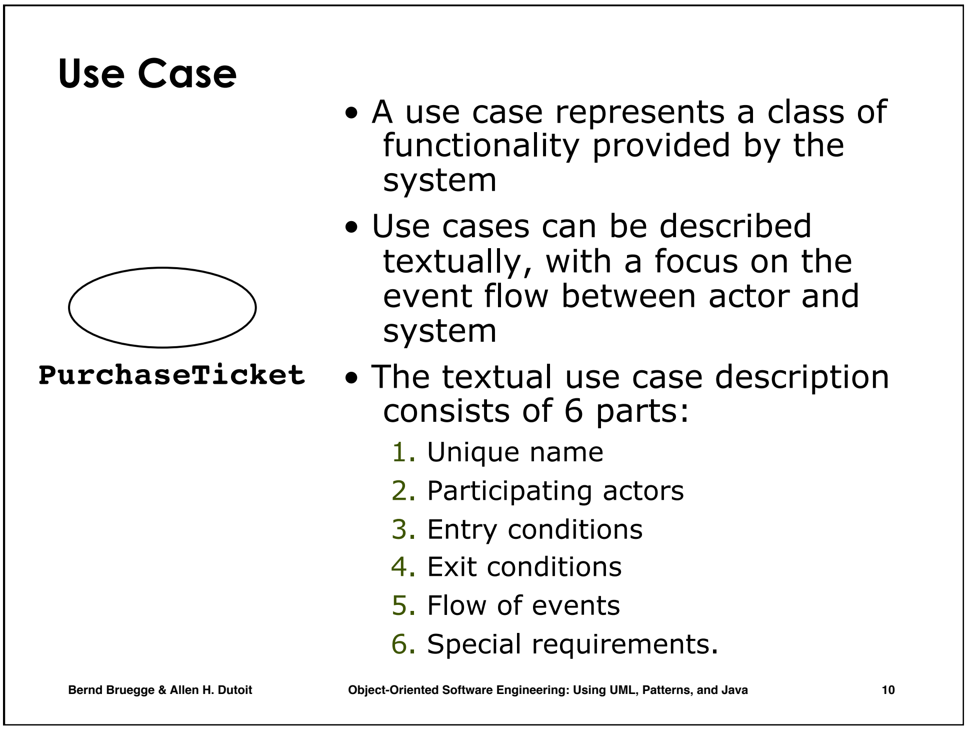


Actors

- An actor is a model for an external entity which interacts (communicates) with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)
- An actor has a unique name and an optional description
- Examples:
  - **Passenger**: A person in the train
  - **GPS satellite**: An external system that provides the system with GPS coordinates.

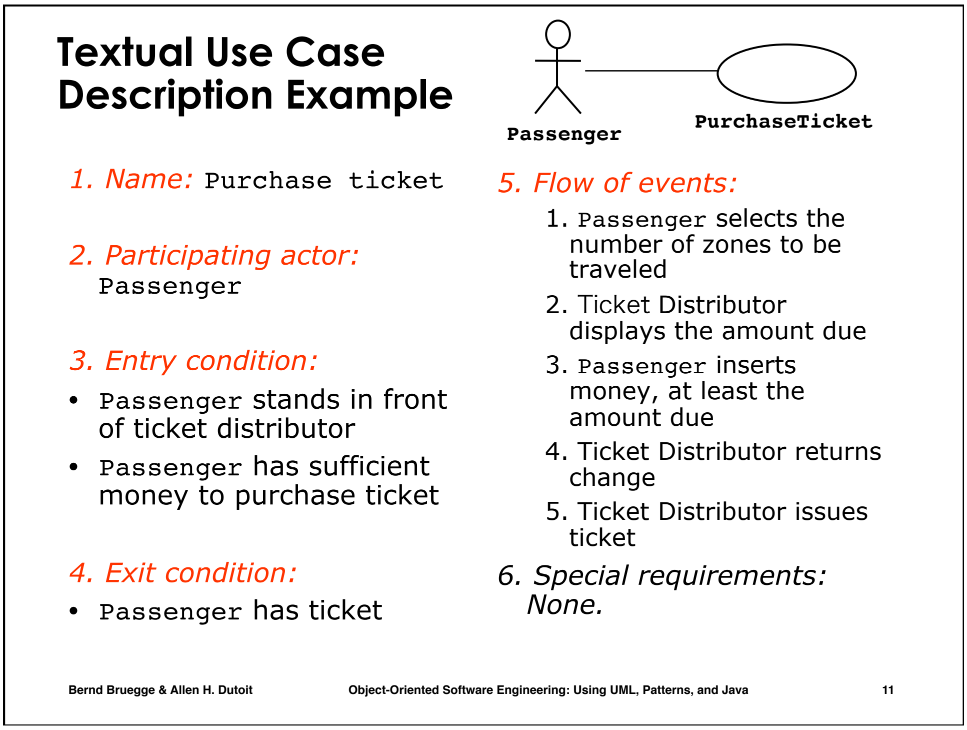
Optional Description

Name



Use Case

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
  1. Unique name
  2. Participating actors
  3. Entry conditions
  4. Exit conditions
  5. Flow of events
  6. Special requirements.



Textual Use Case Description Example

Passenger
PurchaseTicket

1. Name: `Purchase ticket`

2. Participating actor:
`Passenger`

3. Entry condition:

- `Passenger` stands in front of ticket distributor
- `Passenger` has sufficient money to purchase ticket

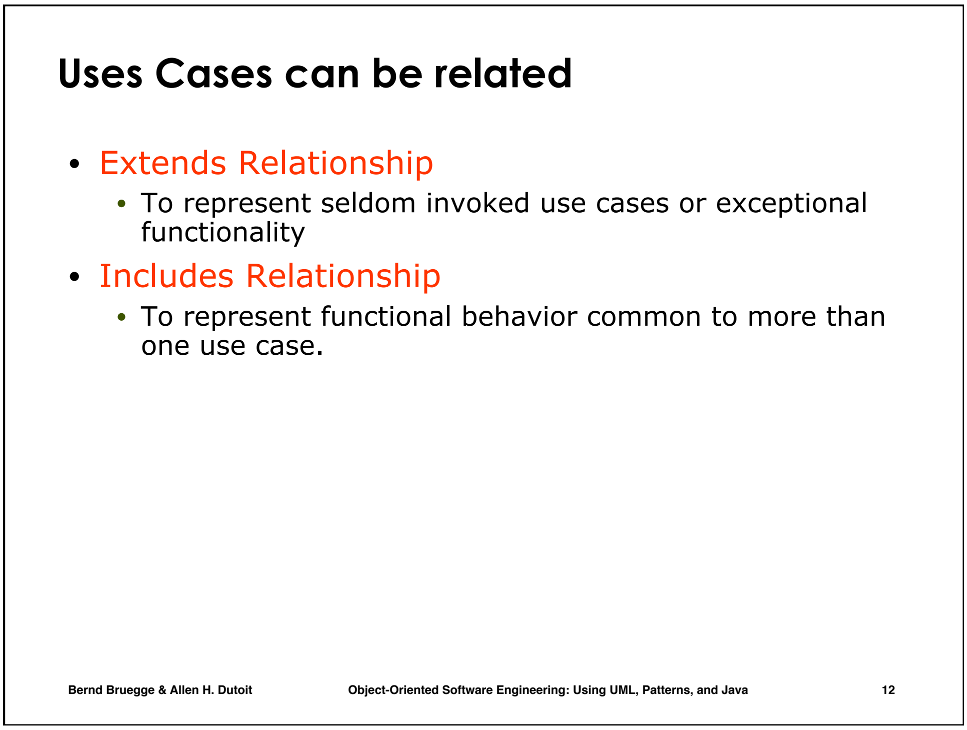
4. Exit condition:

- `Passenger` has ticket

5. Flow of events:

1. `Passenger` selects the number of zones to be traveled
2. Ticket Distributor displays the amount due
3. `Passenger` inserts money, at least the amount due
4. Ticket Distributor returns change
5. Ticket Distributor issues ticket

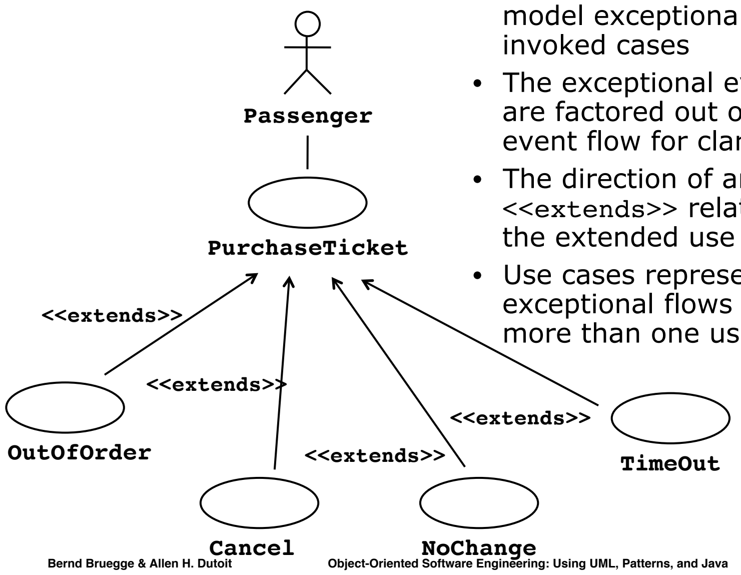
6. Special requirements:
None.



Uses Cases can be related

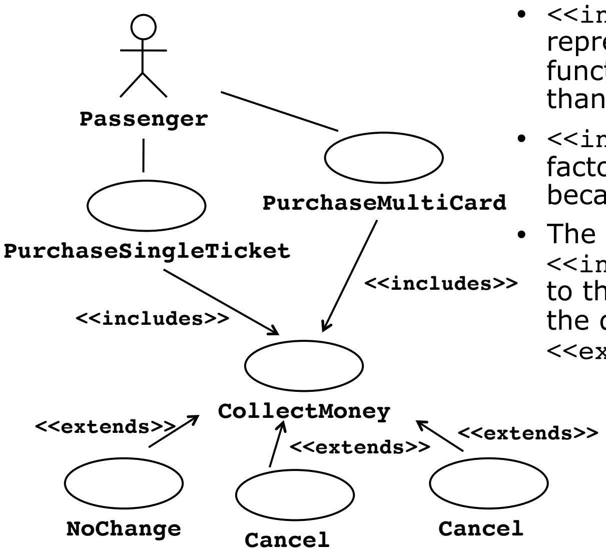
- Extends Relationship
  - To represent seldom invoked use cases or exceptional functionality
- Includes Relationship
  - To represent functional behavior common to more than one use case.

The <<*extends*>> Relationship



- <<extends>> relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an <<extends>> relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

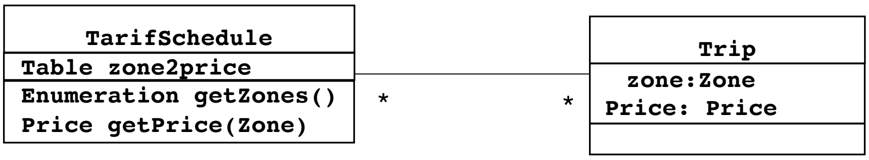
The <<*includes*>> Relationship



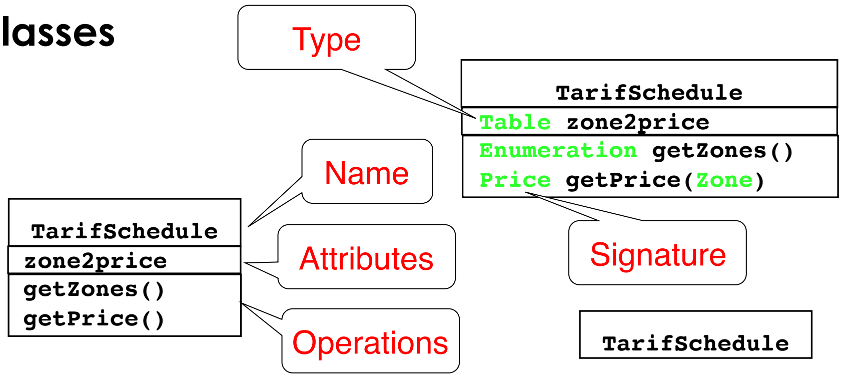
- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

Class Diagrams

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



Classes

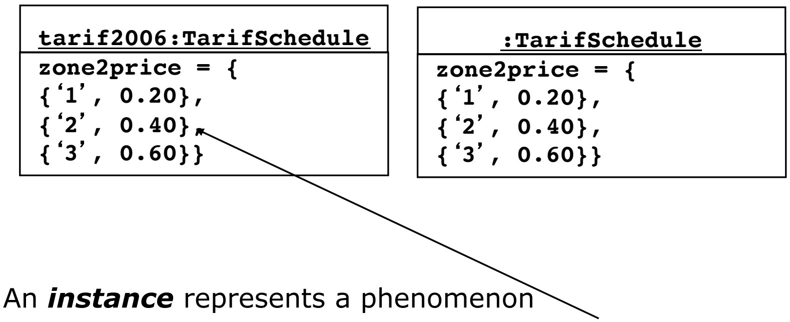


- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

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

  The class name is the only mandatory information

Instances



```
tarif2006:TarifSchedule
zone2price = {
{'1', 0.20},
{'2', 0.40},
{'3', 0.60}}
```

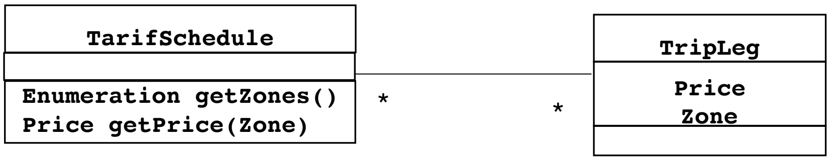
```
:TarifSchedule
zone2price = {
{'1', 0.20},
{'2', 0.40},
{'3', 0.60}}
```

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

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").

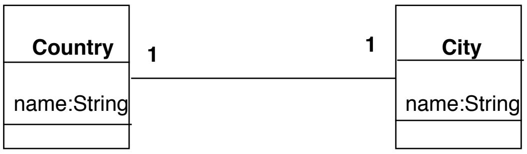
Associations



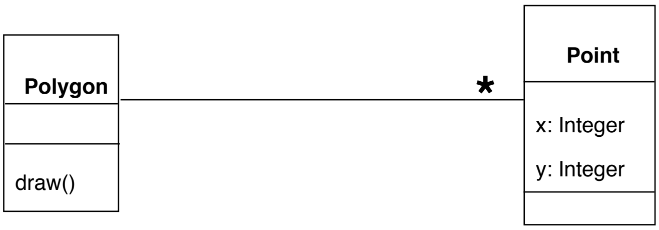
Associations denote relationships between classes

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

1-to-1 and 1-to-many Associations

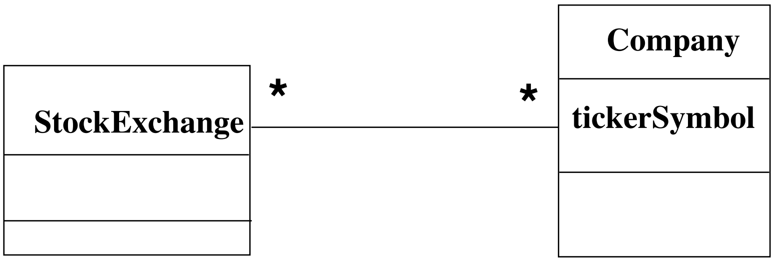


1-to-1 association



1-to-many association

Many-to-Many Associations



StockExchange

*

*

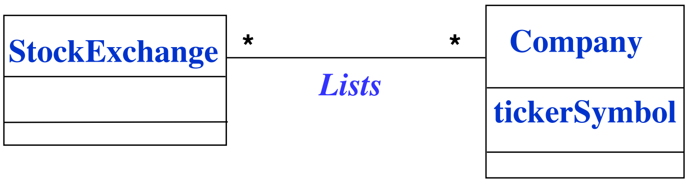
Company

tickerSymbol

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:



StockExchange

*

Lists

*

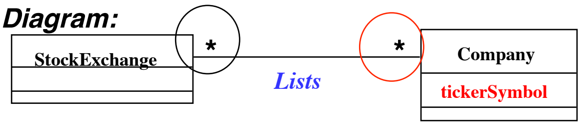
Company

tickerSymbol

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

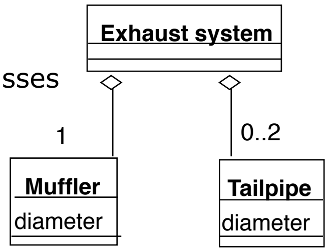
```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

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

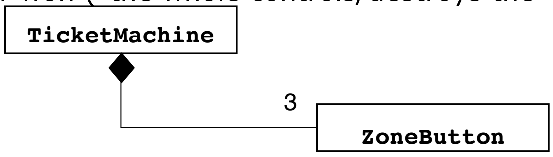
Associations are mapped to Attributes!

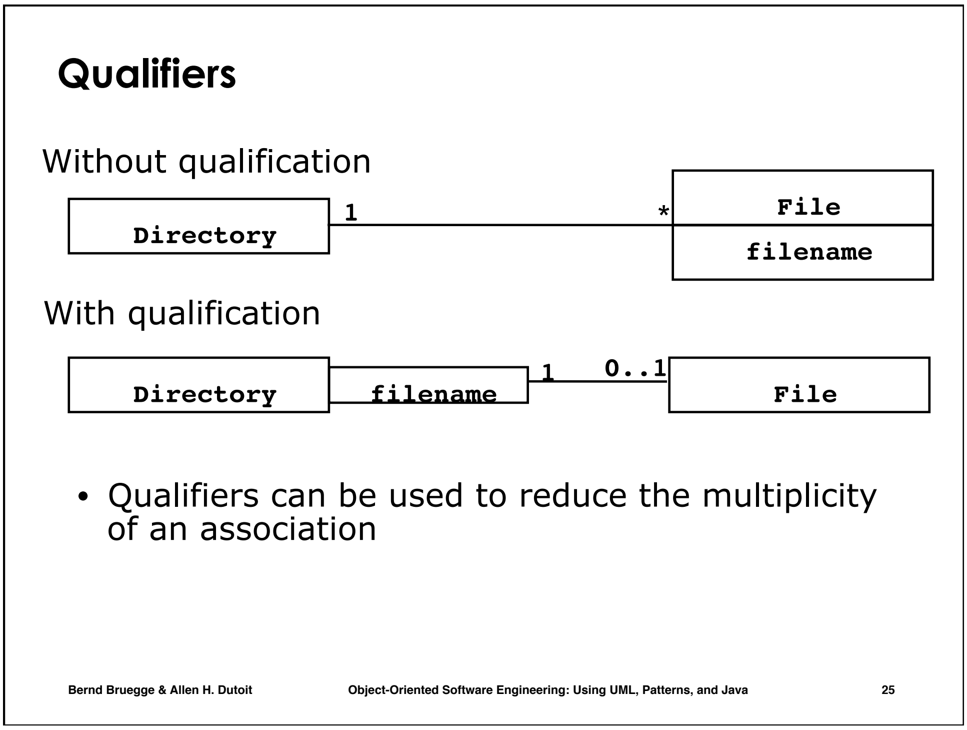
Aggregation

- 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")





Qualifiers

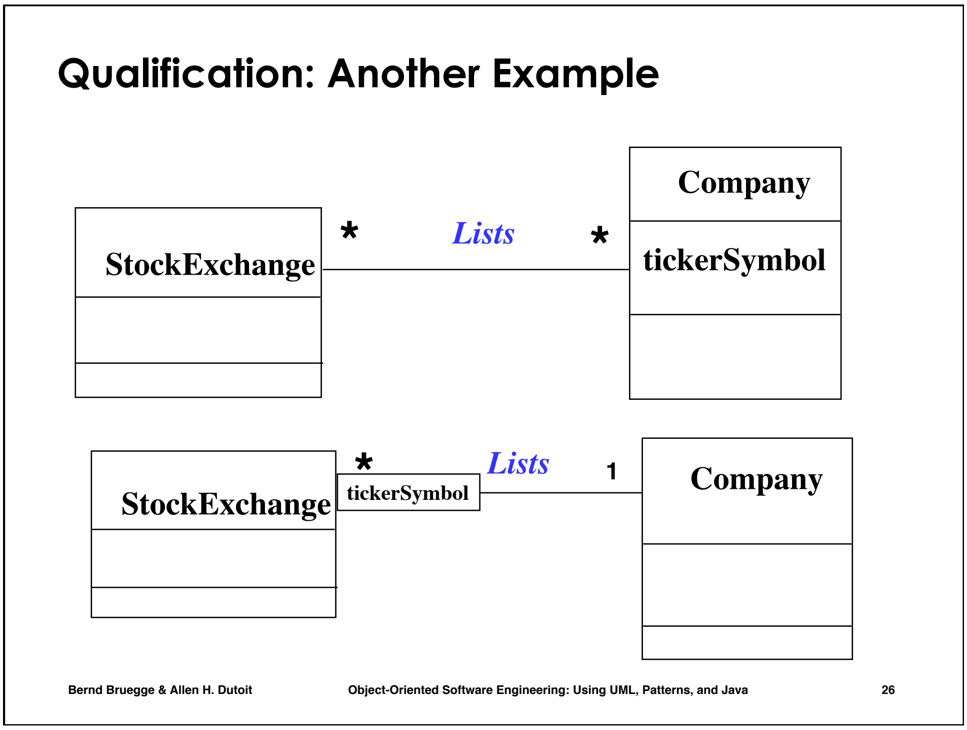
Without qualification

Directory 1 — * File | filename

With qualification

Directory | filename 1 — 0..1 File

- Qualifiers can be used to reduce the multiplicity of an association

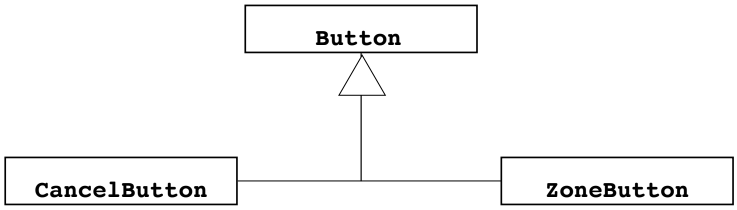


Qualification: Another Example

StockExchange * *Lists* * Company | tickerSymbol

StockExchange | tickerSymbol * *Lists* 1 Company

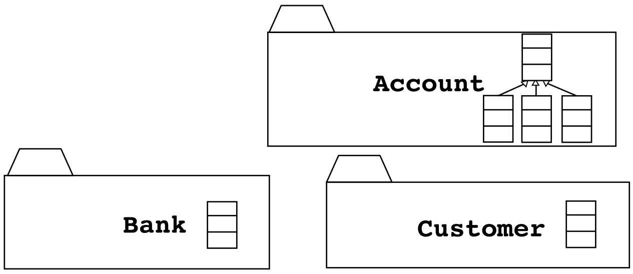
Inheritance



- *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.**

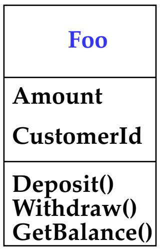
Packages

- 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

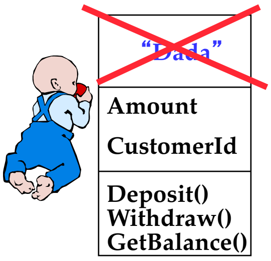
CustomerId

Deposit()
Withdraw()
GetBalance()

Class Identification: Name of Class, Attributes and Methods

Is Foo the right name?

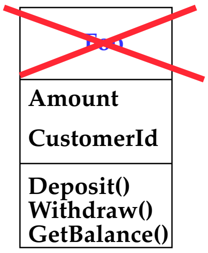
Object Modeling in Practice: Brainstorming


~~“Dada”~~

Amount

CustomerId

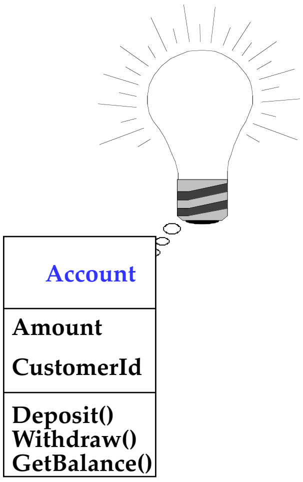
Deposit()
Withdraw()
GetBalance()


~~Foo~~

Amount

CustomerId

Deposit()
Withdraw()
GetBalance()


Account

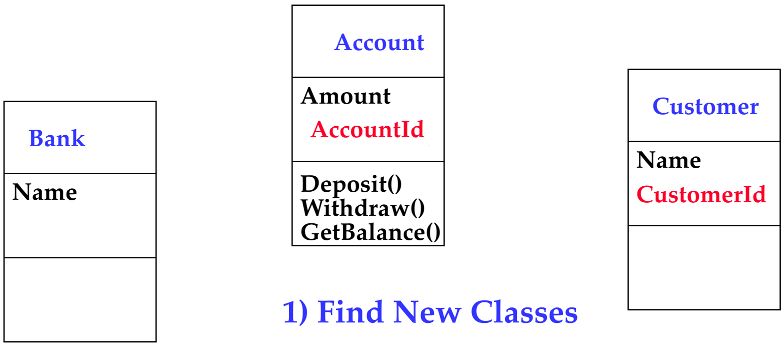
Amount

CustomerId

Deposit()
Withdraw()
GetBalance()

Is Foo the right name?

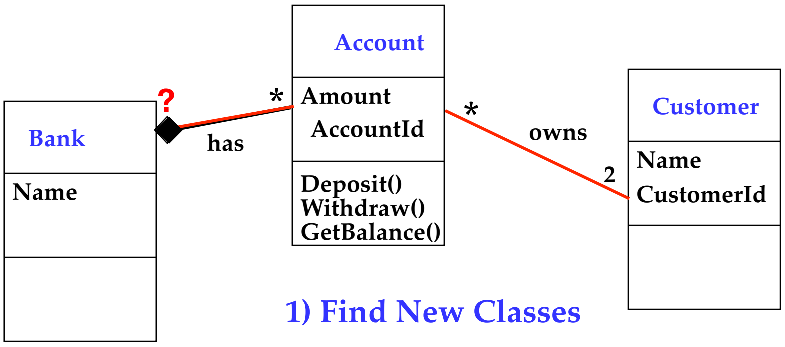
Object Modeling in Practice: More classes



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

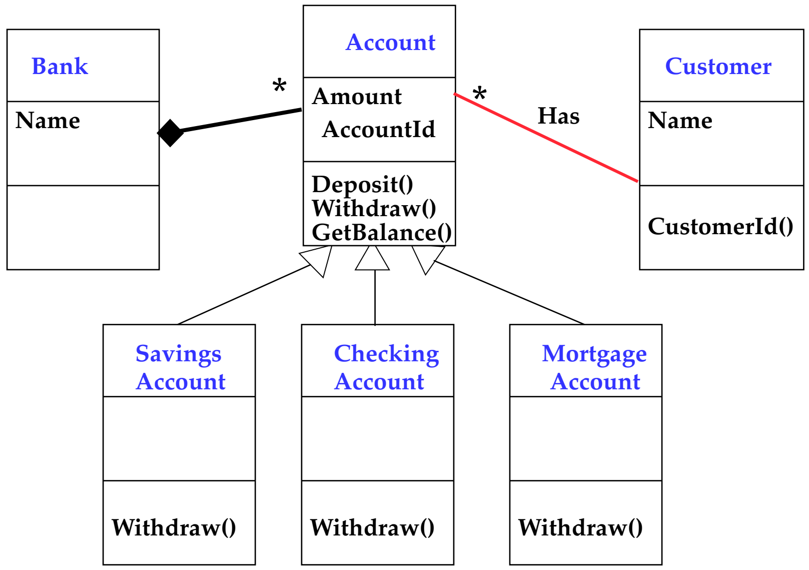
3) Find Associations between Classes

4) Label the generic assocations

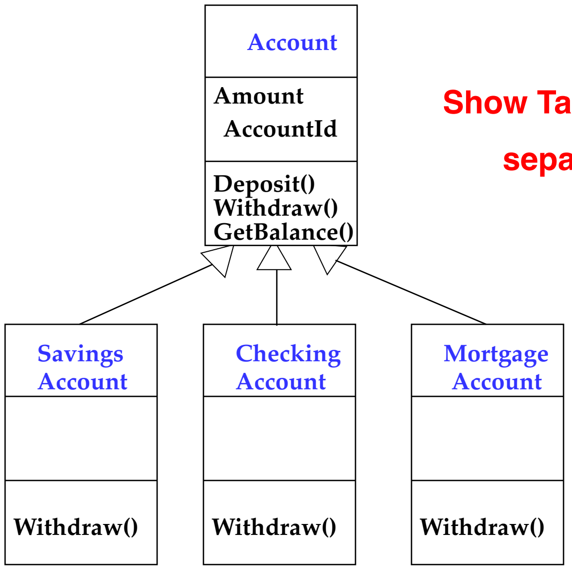
5) Determine the multiplicity of the assocations

6) Review associations

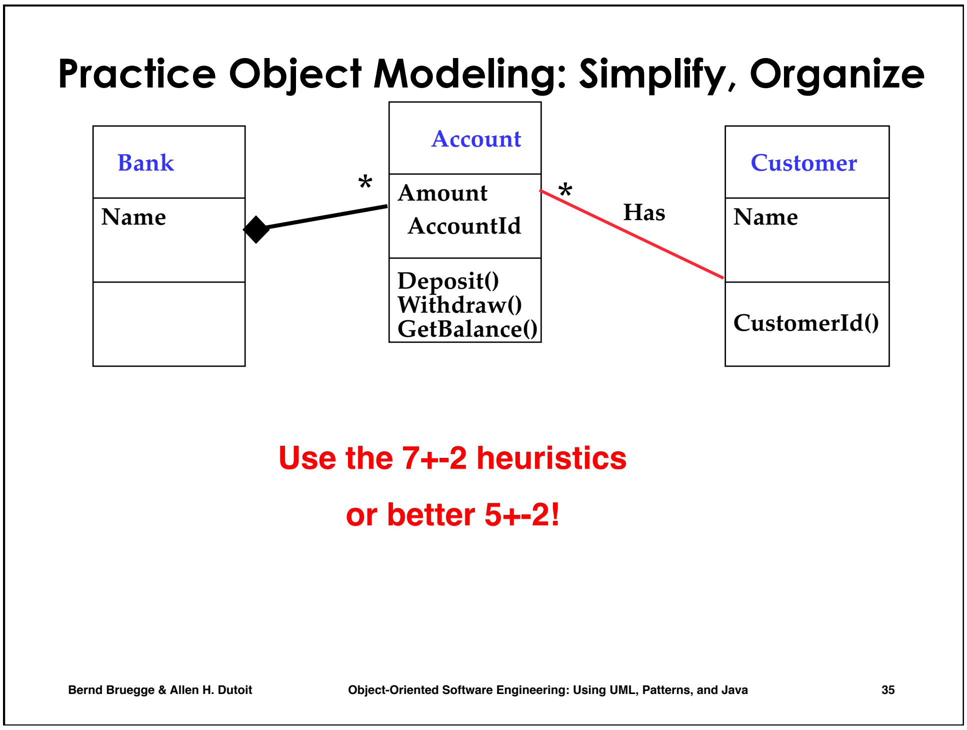
Practice Object Modeling: Find Taxonomies



Practice Object Modeling: Simplify, Organize



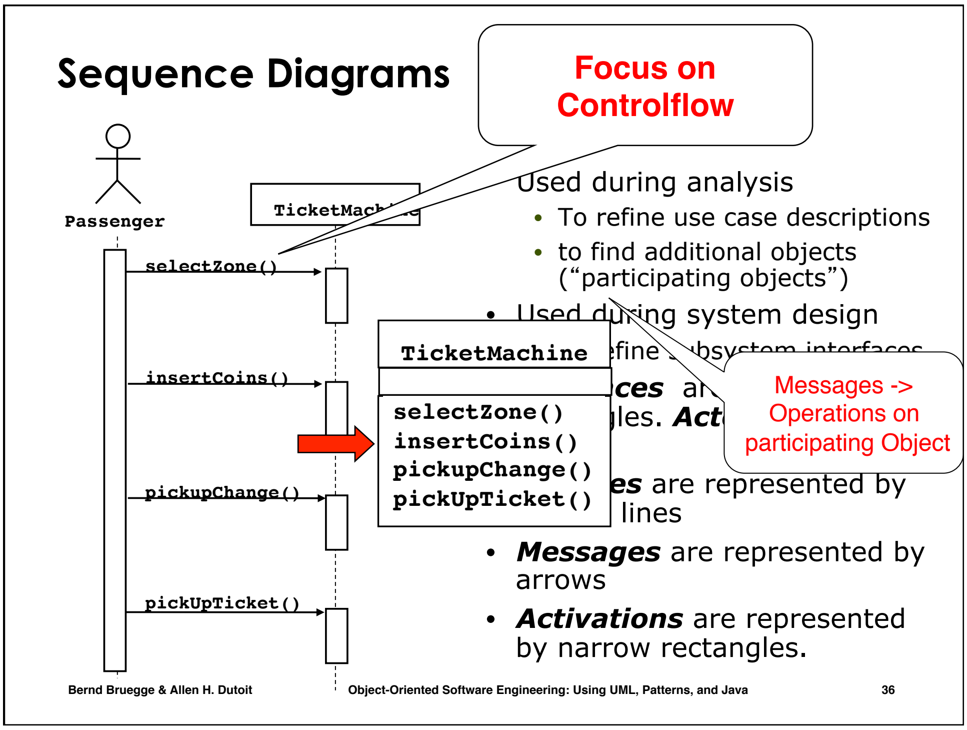
Show Taxonomies separately



Practice Object Modeling: Simplify, Organize

Use the 7+-2 heuristics

or better 5+-2!



Sequence Diagrams

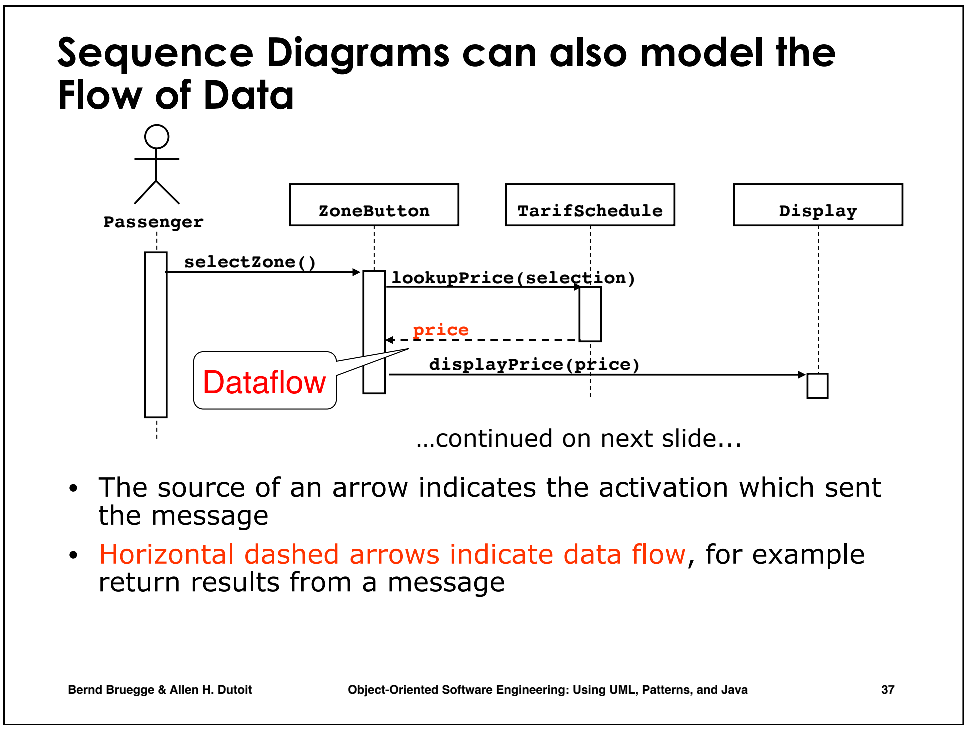
Focus on Controlflow

Used during analysis

- To refine use case descriptions
- to find additional objects ("participating objects")

- Used during system design
  - efine subsystem interfaces
- *ces* ar
  gles. *Act*
  es are represented by lines
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles.

Messages -> Operations on participating Object



Sequence Diagrams can also model the Flow of Data

Passenger

ZoneButton

TarifSchedule

Display

selectZone()

lookupPrice(selection)

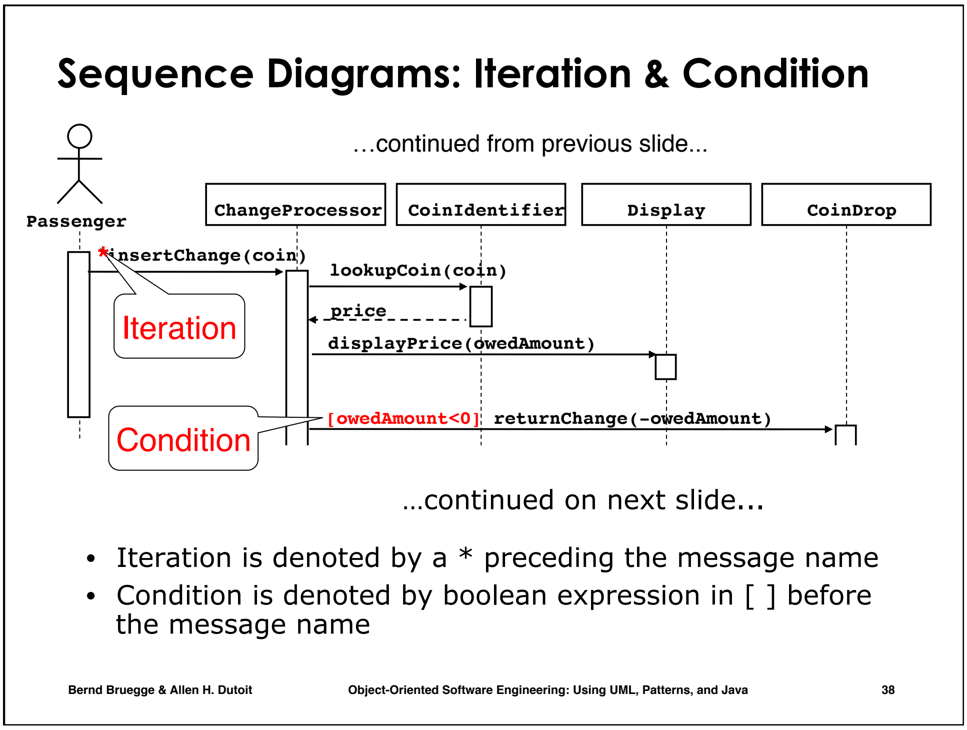
price

displayPrice(price)

Dataflow

...continued on next slide...

- The source of an arrow indicates the activation which sent the message
- Horizontal dashed arrows indicate data flow, for example return results from a message



Sequence Diagrams: Iteration & Condition

…continued from previous slide...

Passenger

ChangeProcessor

CoinIdentifier

Display

CoinDrop

*insertChange(coin)

lookupCoin(coin)

price

displayPrice(owedAmount)

[owedAmount<0] returnChange(-owedAmount)

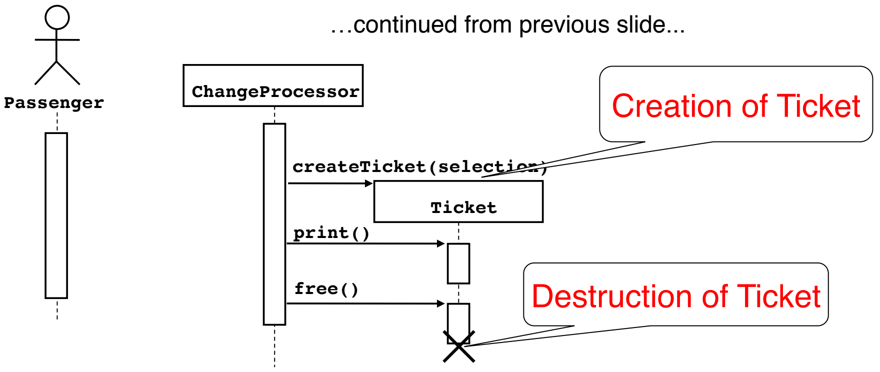
Iteration

Condition

...continued on next slide...

- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction



…continued from previous slide...

Creation of Ticket

Destruction of Ticket

- Creation is denoted by a message arrow pointing to the object
- Destruction is denoted by an X mark at the end of the destruction activation
    - In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence Diagram Properties

- UML sequence diagram represent *behavior in terms of interactions*
- Useful to identify or find missing objects
- Time consuming to build, but worth the investment
- Complement the class diagrams (which represent structure).

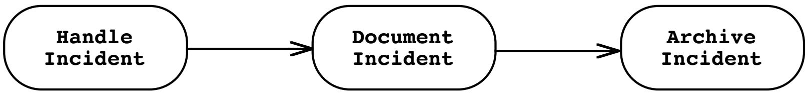
Outline of this Class

- A more detailed view on

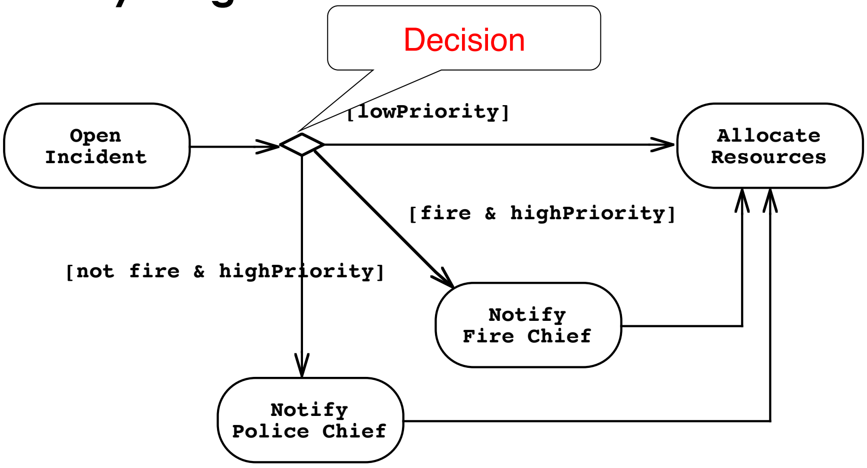
  ✓Use case diagrams
  ✓Class diagrams
  ✓Sequence diagrams
  ➢Activity diagrams

Activity Diagrams

- An activity diagram is a special case of a state chart diagram
- The states are activities (“functions”)
- An activity diagram is useful to depict the workflow in a system



Activity Diagrams allow to model Decisions


Decision

Open Incident

[lowPriority]

Allocate Resources

[fire & highPriority]

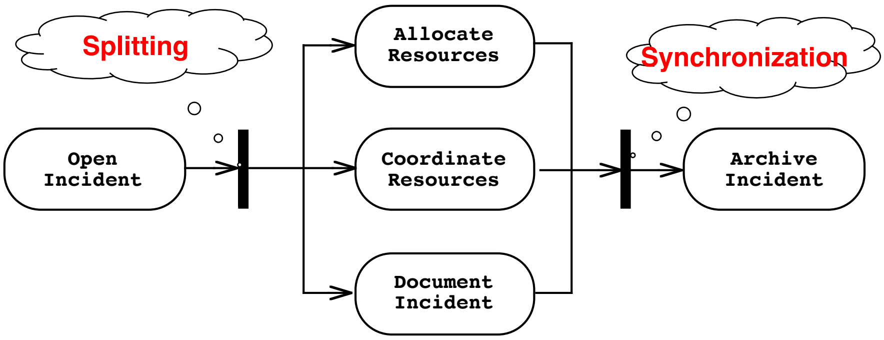
[not fire & highPriority]

Notify Fire Chief

Notify Police Chief

Activity Diagrams can model Concurrency

- Synchronization of multiple activities
- Splitting the flow of control into multiple threads


Splitting

Synchronization

Open Incident

Allocate Resources

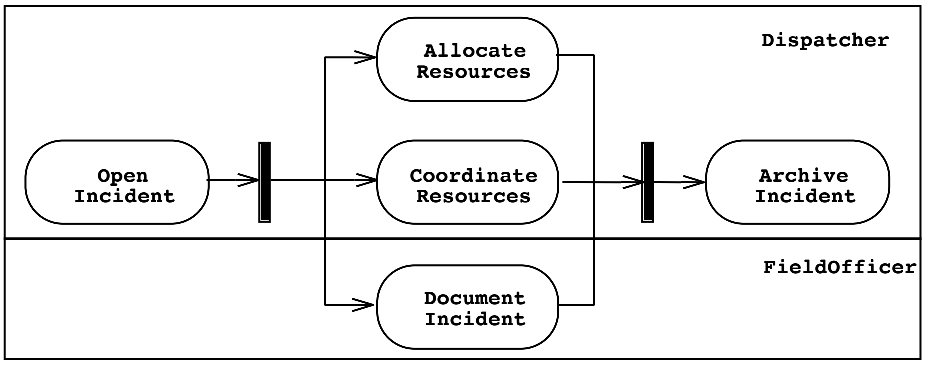
Coordinate Resources

Document Incident

Archive Incident

Activity Diagrams: Grouping of Activities

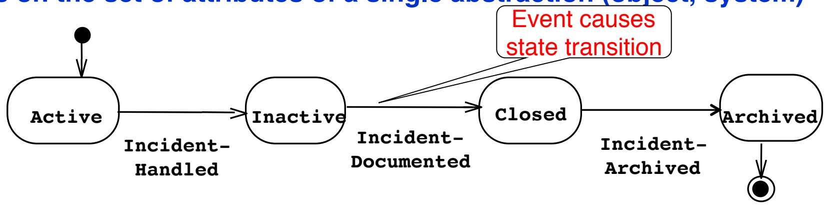
- Activities may be grouped into swimlanes to denote the object or subsystem that implements the activities.



Activity Diagram vs. Statechart Diagram

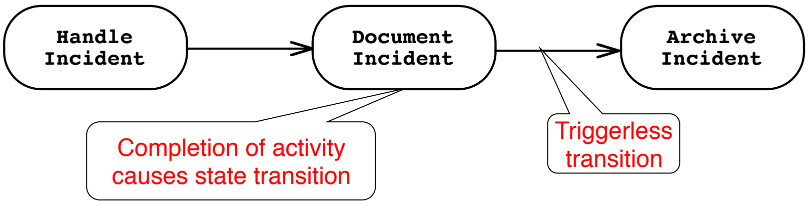
Statechart Diagram for Incident

Focus on the set of attributes of a single abstraction (object, system)


Event causes state transition

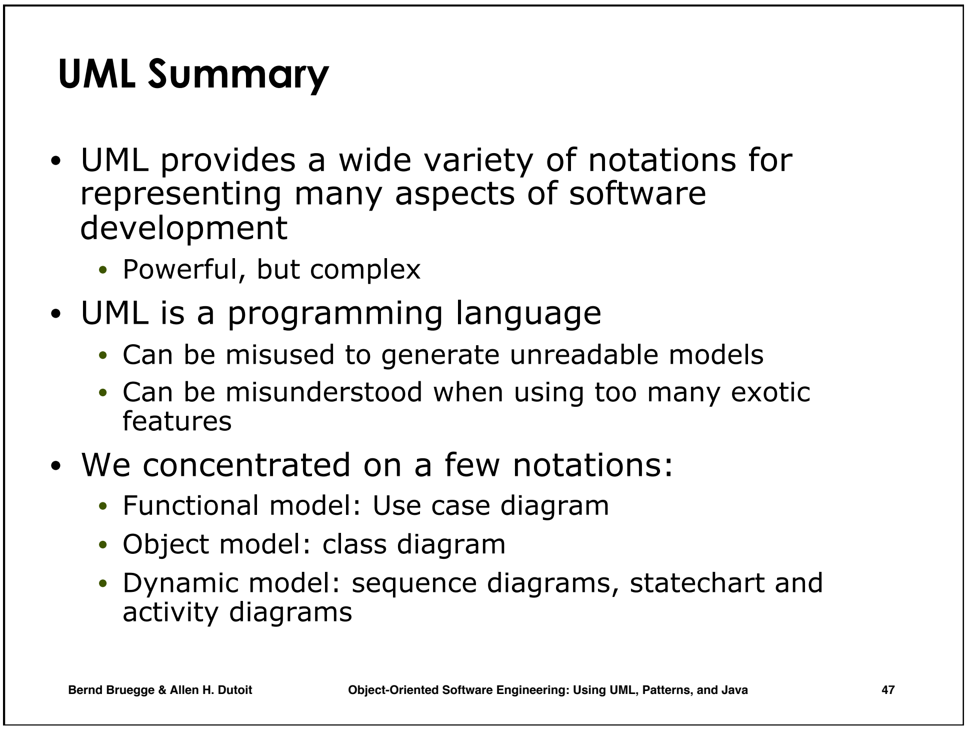
Activity Diagram for Incident

(Focus on dataflow in a system)



Completion of activity causes state transition

Triggerless transition



UML Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex
- UML is a programming language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
- We concentrated on a few notations:
  - Functional model: Use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, statechart and activity diagrams