CE/CS/SE 3354
Software Engineering

Unified Modeling Language (UML)

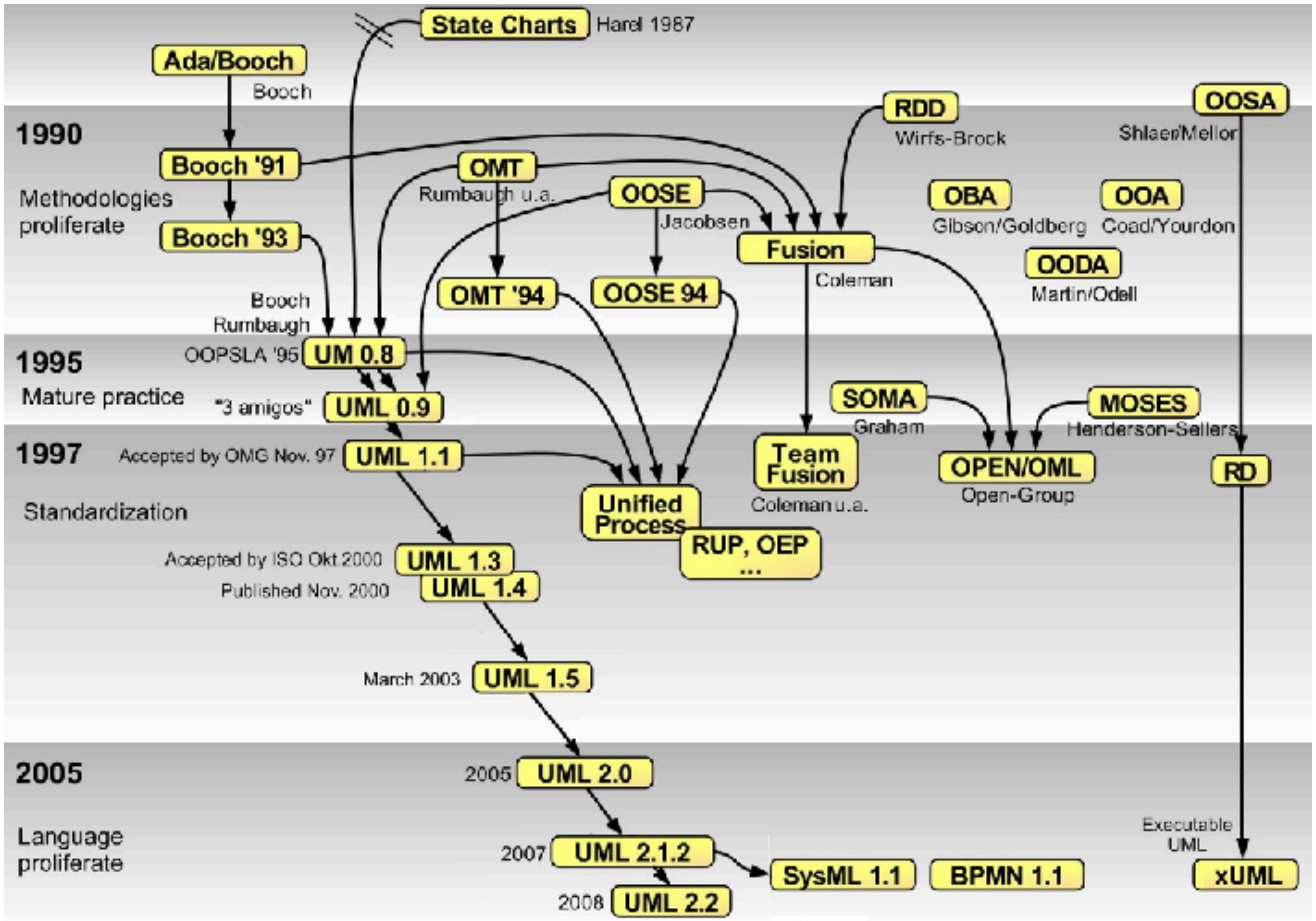
History of UML

- Unified Modeling Language (UML)
- UML became a standard in 1997 after many years of modeling war: 50+ modeling languages
  - Three leading languages

    Booch, OMT, OOSE
  - 1994 Rumbaugh (OMT) joined Booch (in Rational)
  - 1995 Rational bought Objectory

    Jacobson, OOSE -- use cases
  - UML = OMT + Booch + OOSE + …

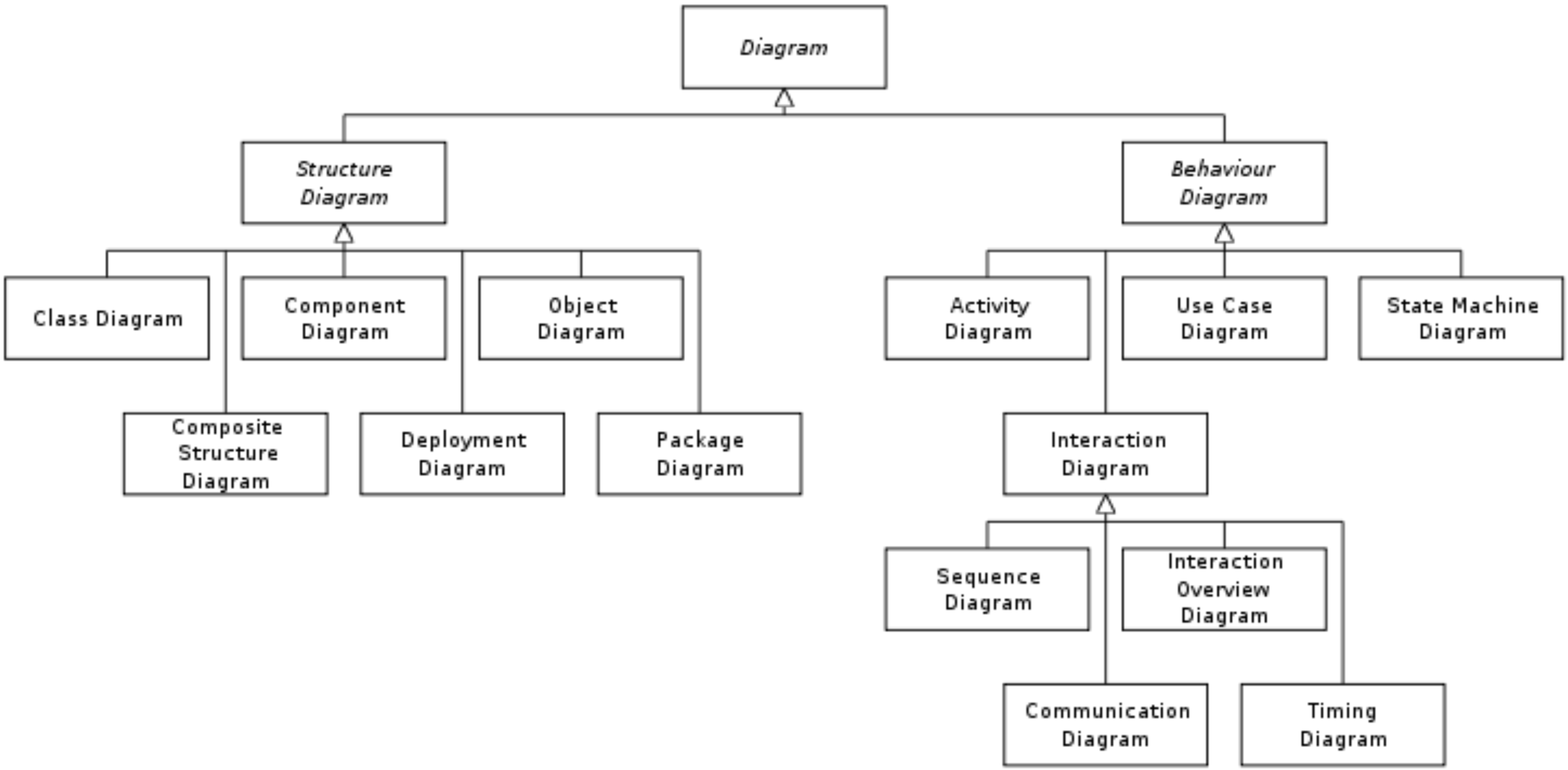
History of Object-Oriented Methods and Notation



UML: Introduction

- UML is a set of modeling notations, which include 13 diagrams
  - Static structure of the system
    - Class diagram
    - Object diagram
    - … …
  - Dynamic behavior of the system
    - Use-case diagram
    - Sequence diagram
    - … …

UML: 13 Diagrams



UML Use Case Diagram

- Used as a graphics notation for requirement engineering
  - System: drawn as a box
  - Actors: outside the system
  - Use cases: inside the system
  - Relations among use cases and actors

Actors

- Actors are external to the system
- An actor specifies a role
  - Users that operate the system directly
  - Other software systems or hardware pieces that interact with the system
- One person or thing may play many roles in relation to the system simultaneously or over time

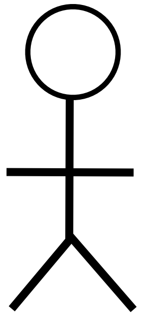
Use Cases

- Use cases are usages of the system
- Use cases capture the functional requirements
  - Use cases provide the high-level descriptions of the system's functionality in terms of interactions
  - Use cases show inputs and outputs between the system and the environment
  - Use cases are from the user's point of view

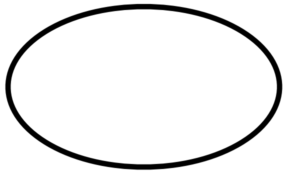
Use Case – An Example

- ATM system
  - Withdraw cash
  - Check account balance
  - Maintain usage statistics
  - …

Legends



Actor: an entity in the environment that initiates and interacts with the system



Use case: usage of system, a set of sequences of actions

Association: relation between actor and use cases

<<Include>>

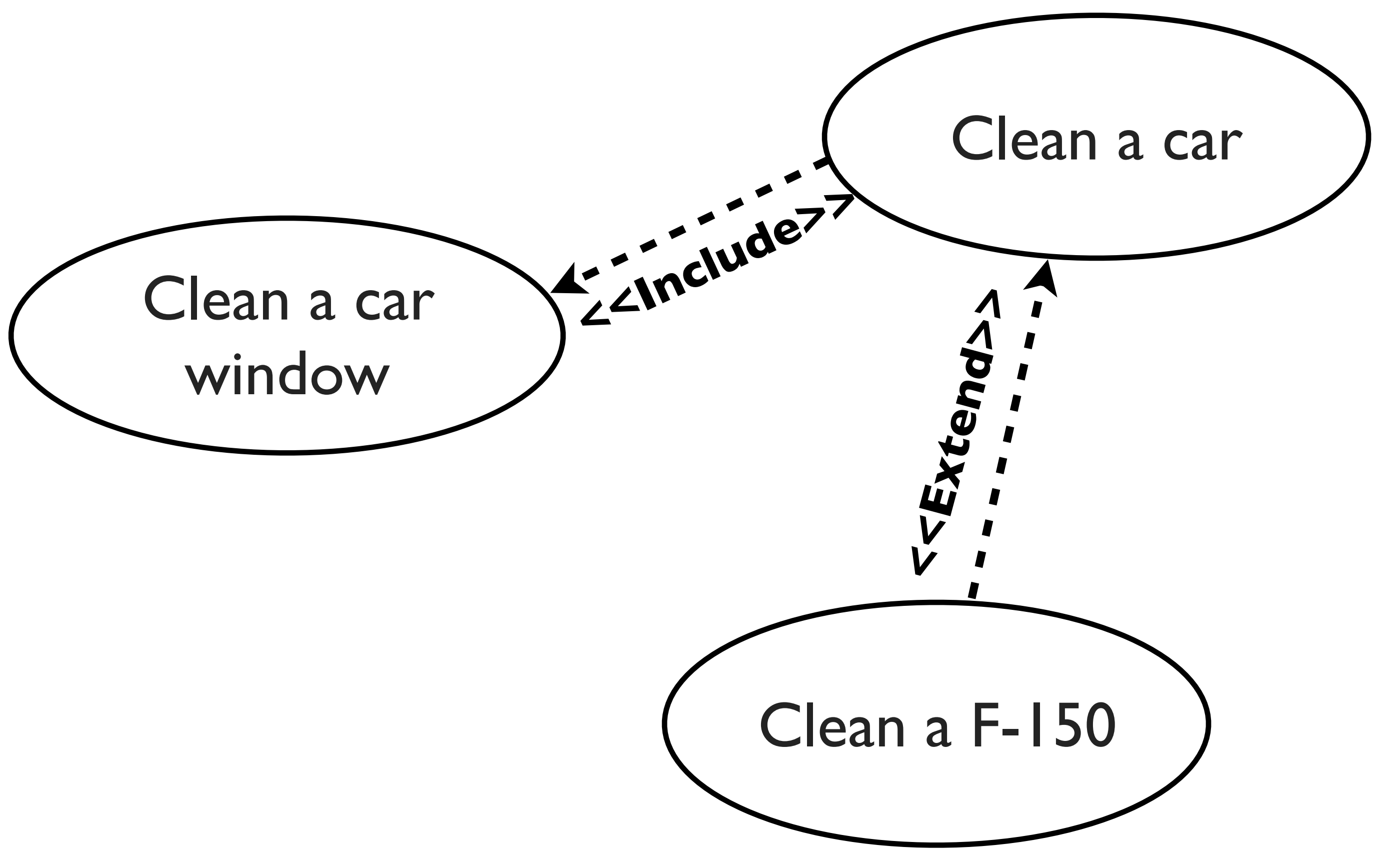
Includes dependency: a base use case includes the sub use case as a component

<<Extend>>

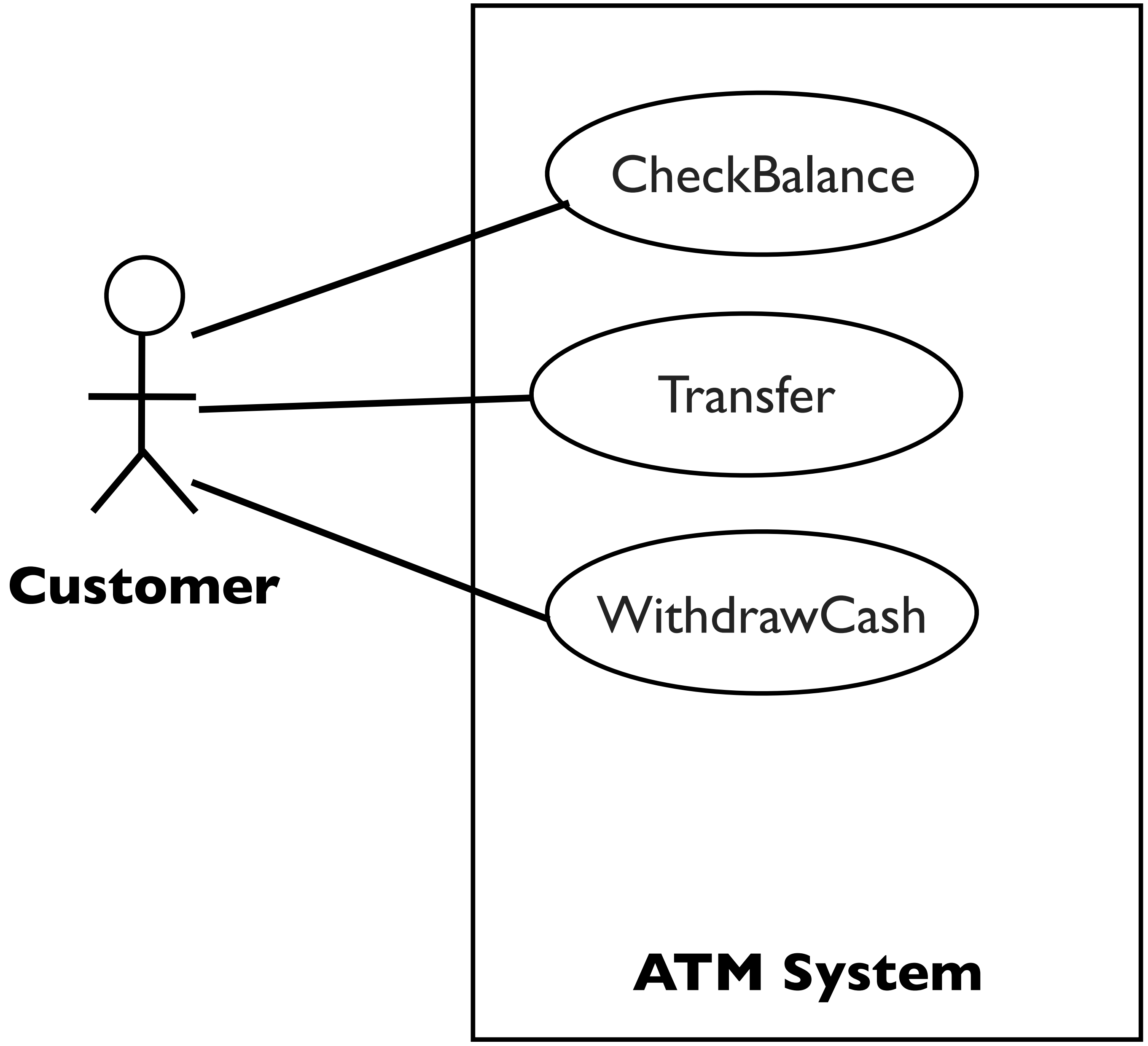
Extends dependency: a subtype of use cases that extend the behavior of the base use case

Generalization: one actor can inherit the role of the other actor

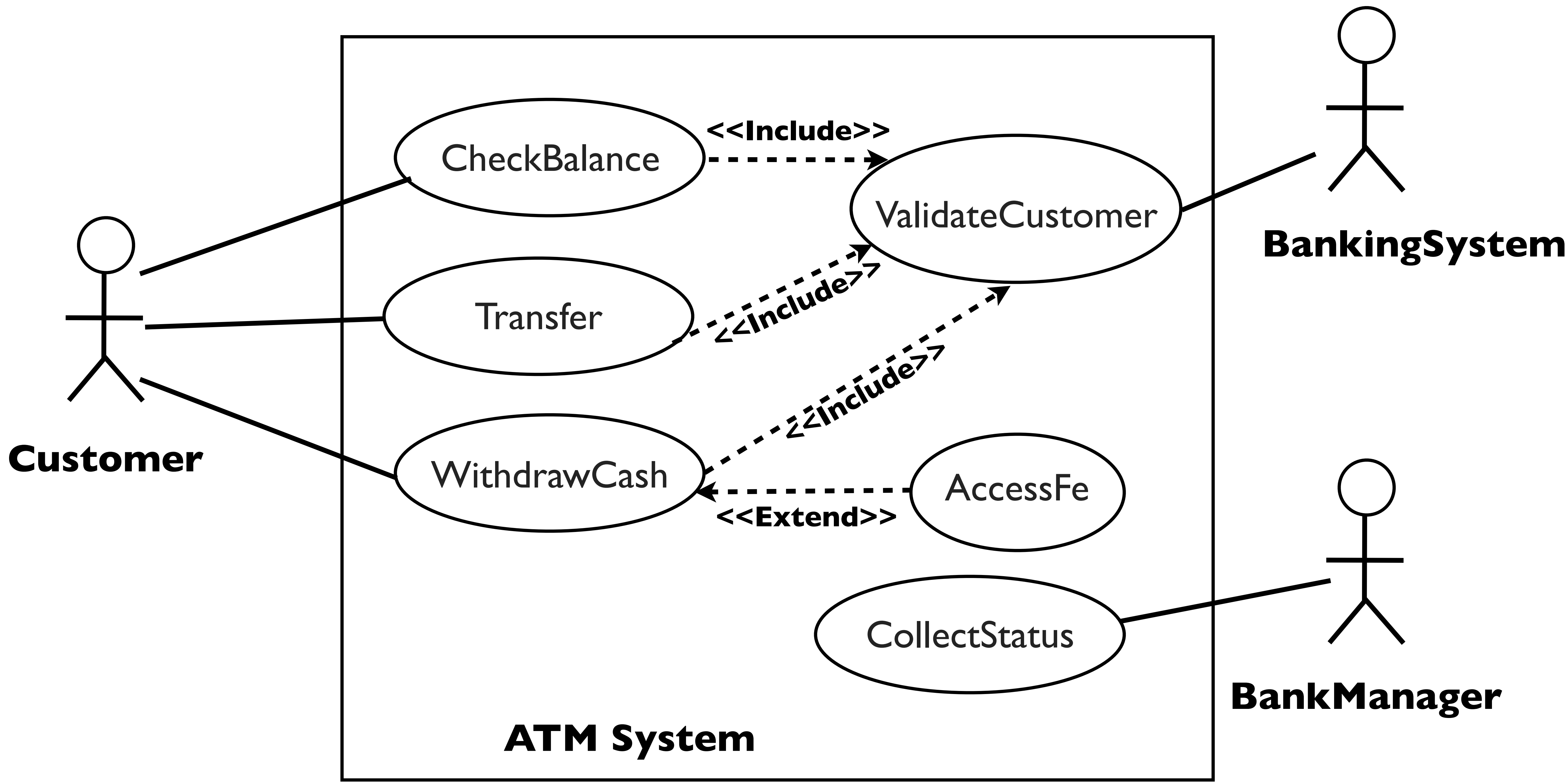
Include vs. Extend



Initial Use Case Diagram for ATM



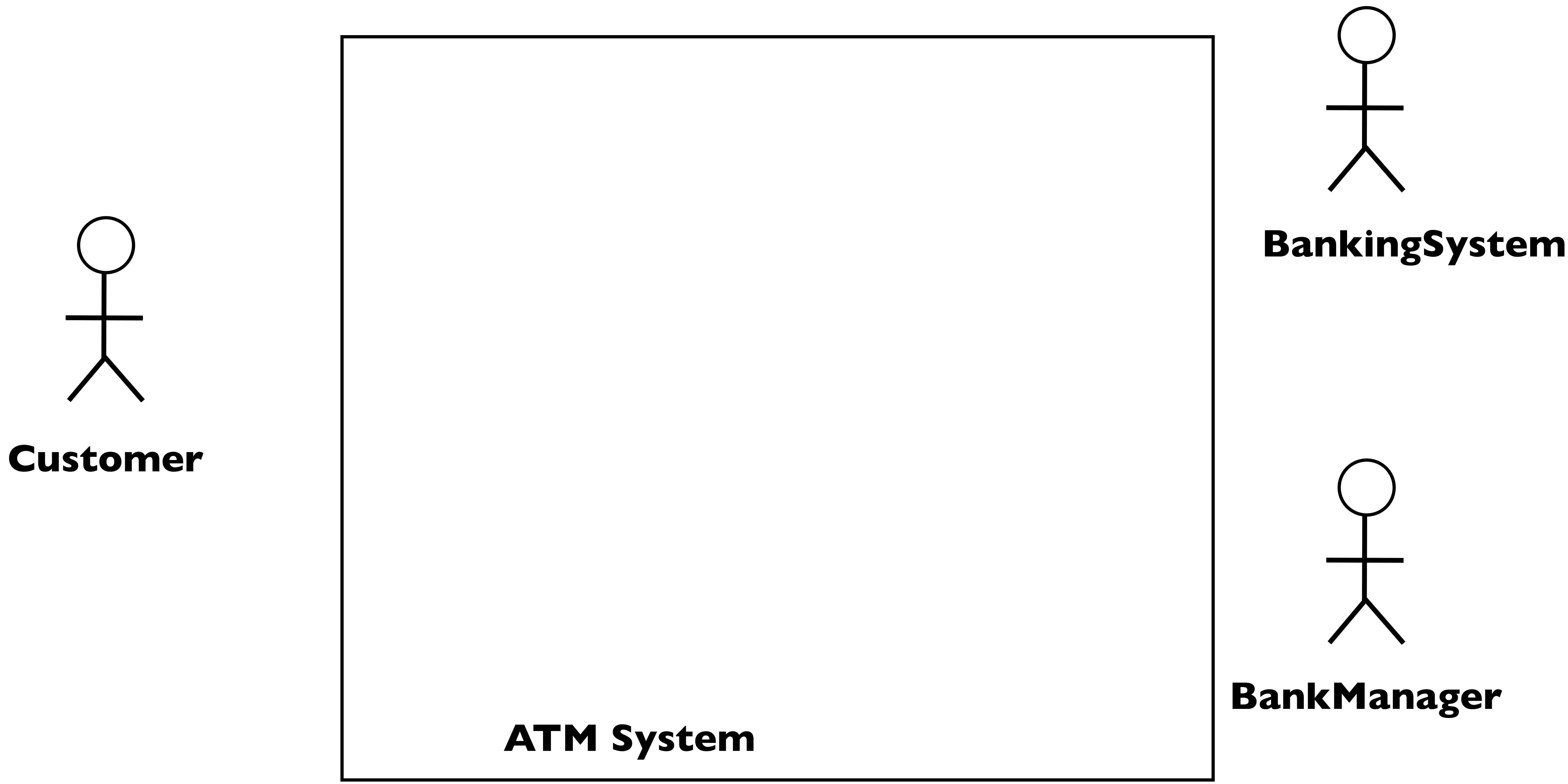
Elaborated Use Case Diagram for ATM



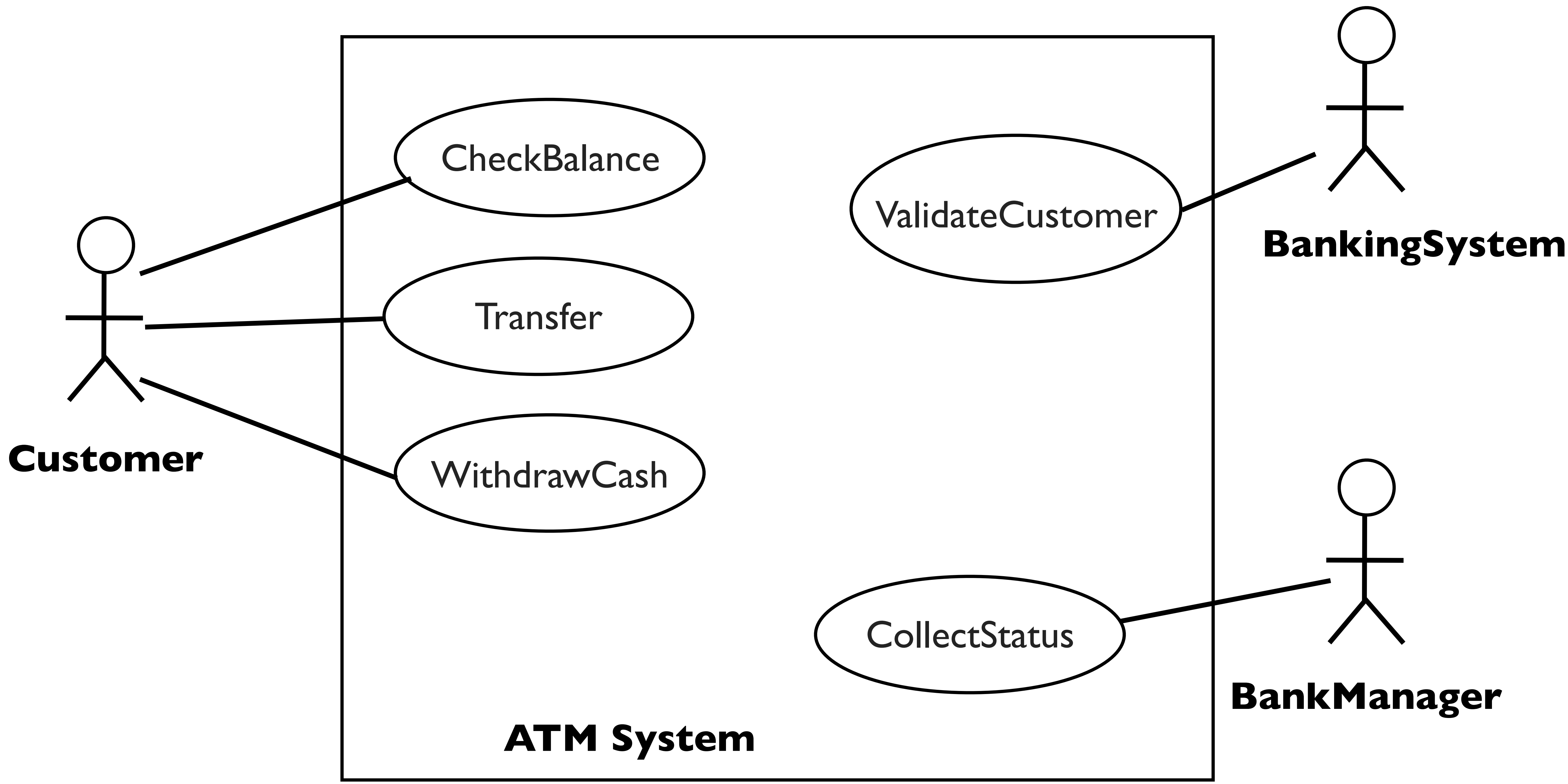
Process for Identifying Use Cases

- Choose your system boundary
- Identify primary actors
- For each actor, find their goals
- Define a use case for each goal
- Decompose complex use cases into sub-use cases
- Organize normal alternatives as extension use cases

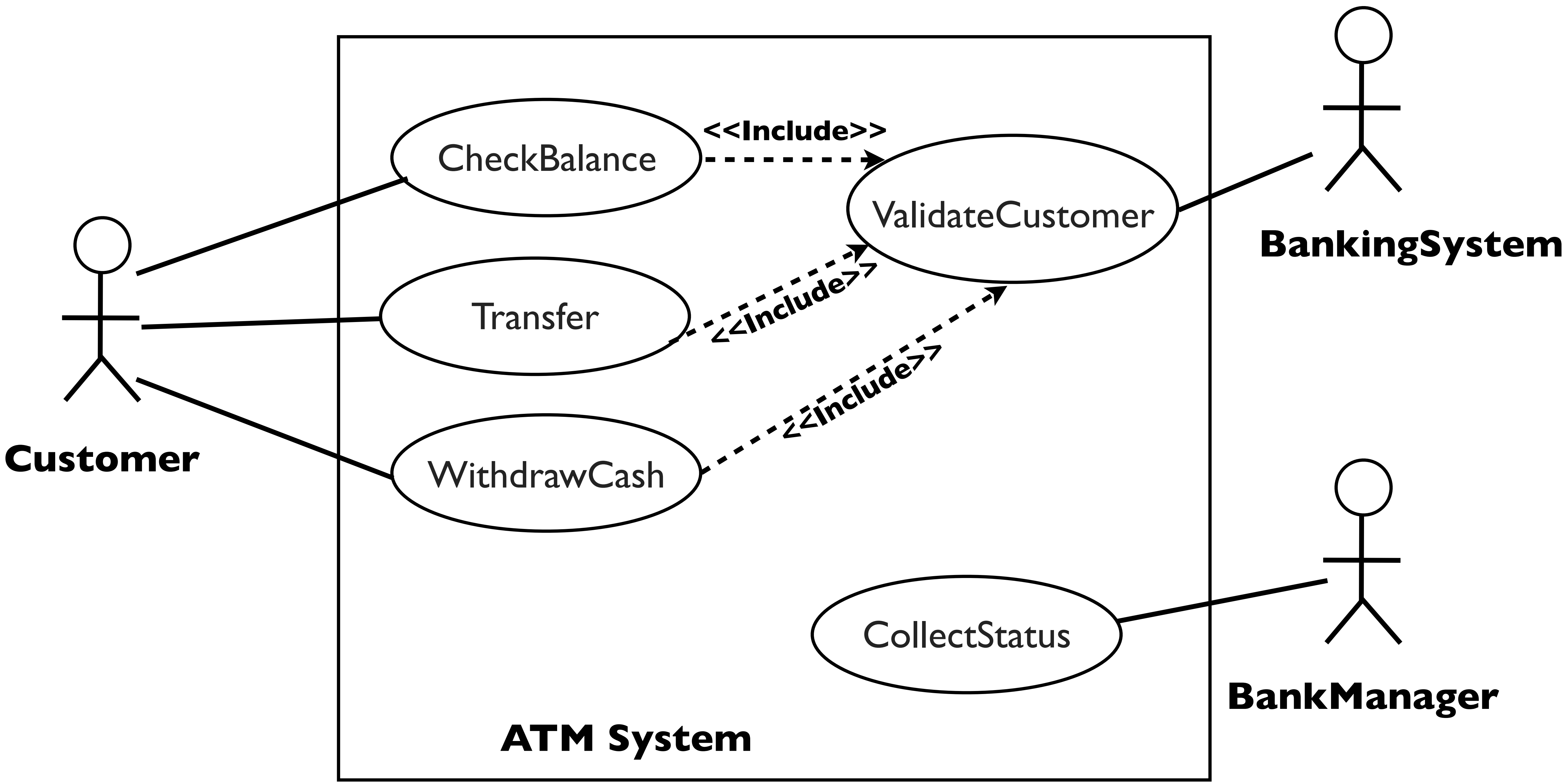
Elaborated Use Case Diagram for ATM



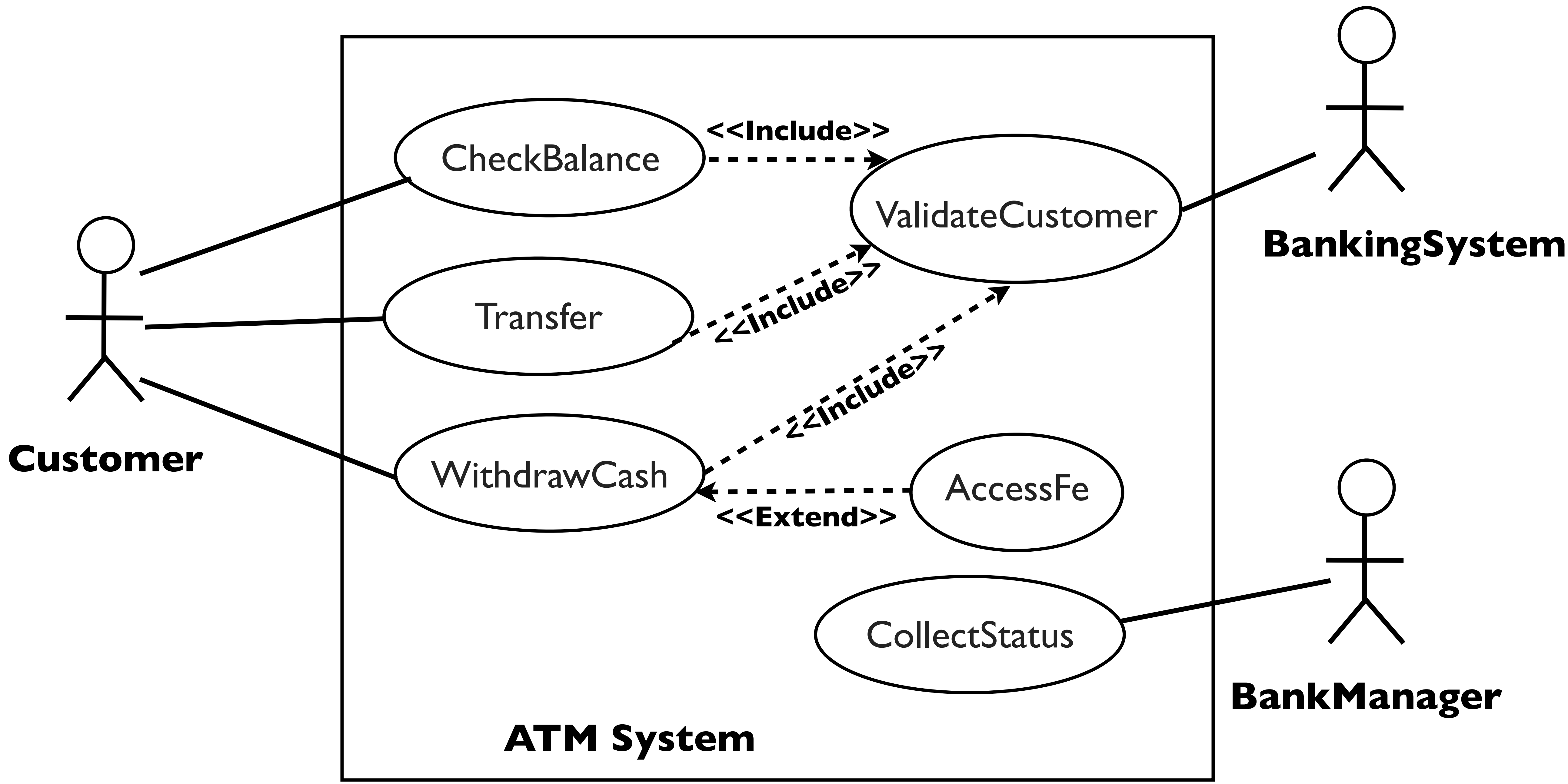
Elaborated Use Case Diagram for ATM



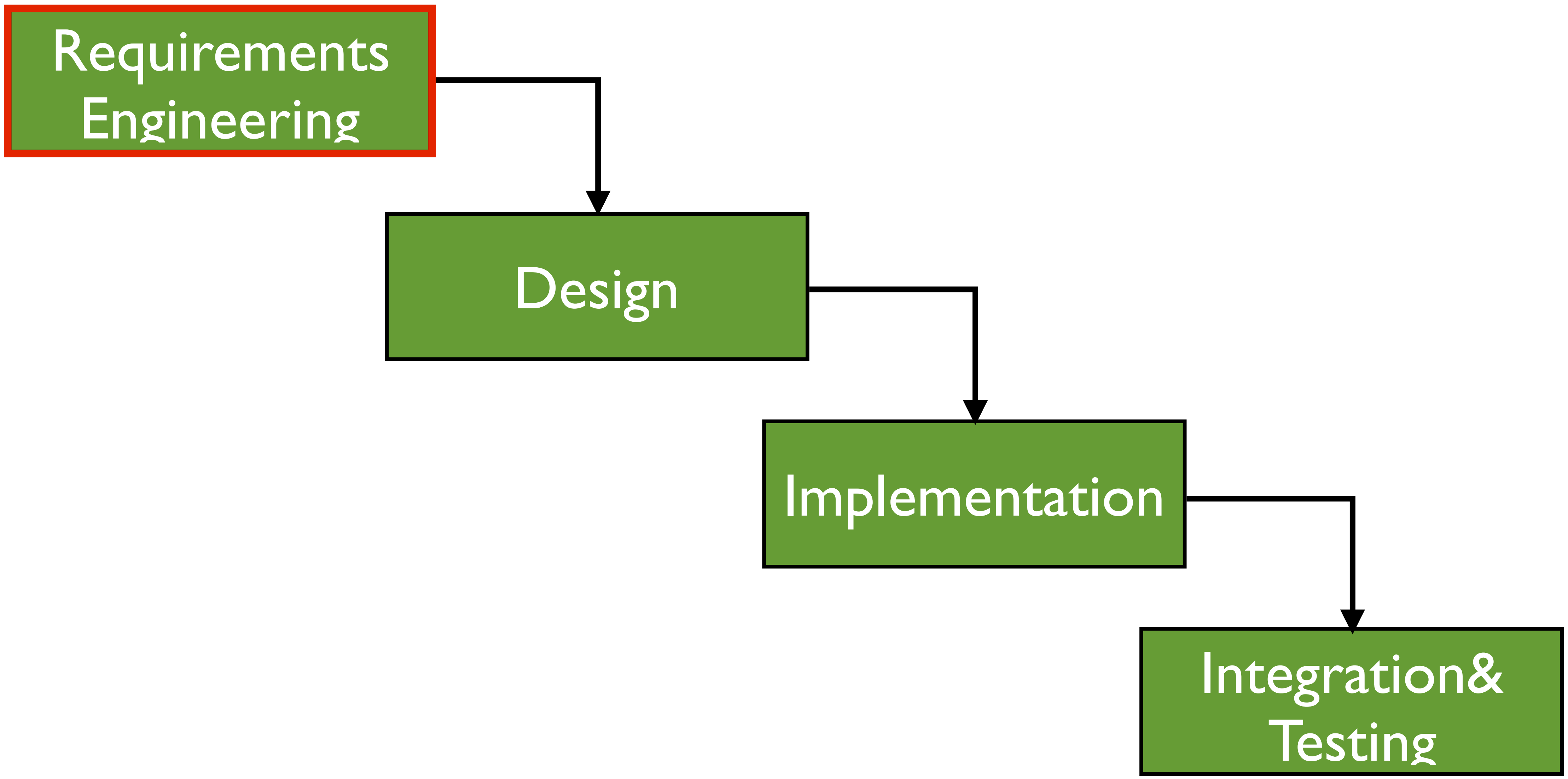
Elaborated Use Case Diagram for ATM



Elaborated Use Case Diagram for ATM



Basic Steps in Software Process Models



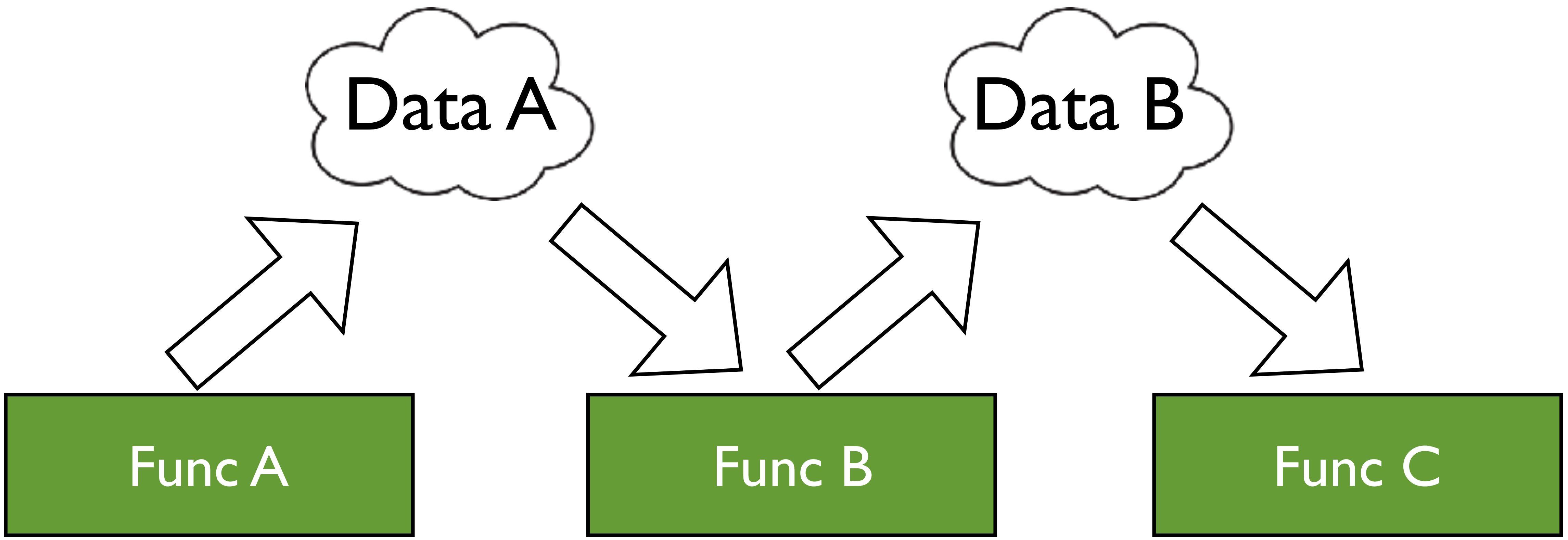
What Is Procedural Approach?

- Traditional programming languages were procedural
  - C, Pascal, BASIC, Ada and COBOL
- Programming in procedural languages involves choosing data holders (appropriate ways to store data), designing algorithms, and translating algorithm into code

What Is Procedural Approach? (Cont'd)

- In procedural programming, data and operations on the data are separated
- This methodology requires sending data to procedure/functions

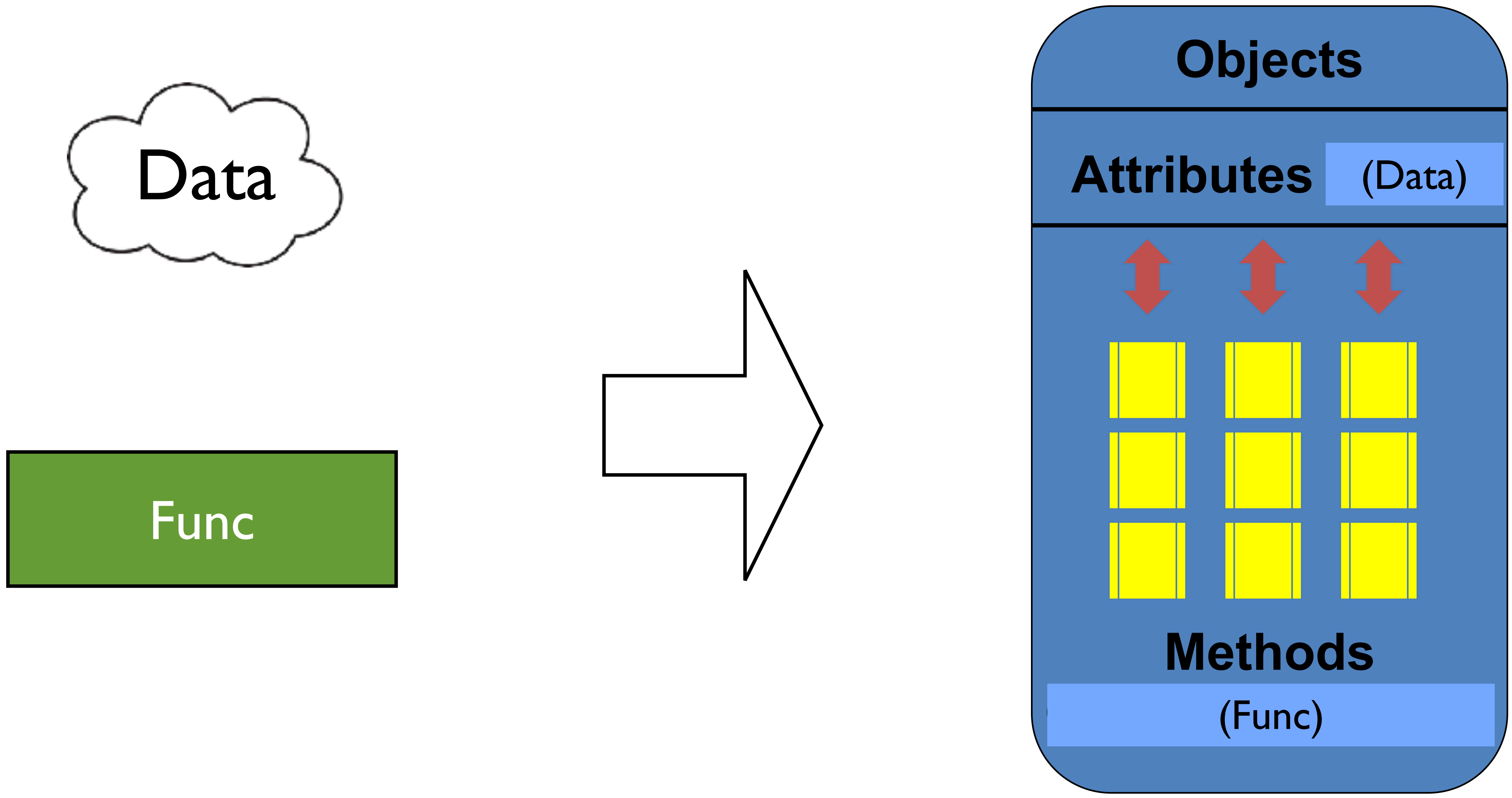
Procedural Design



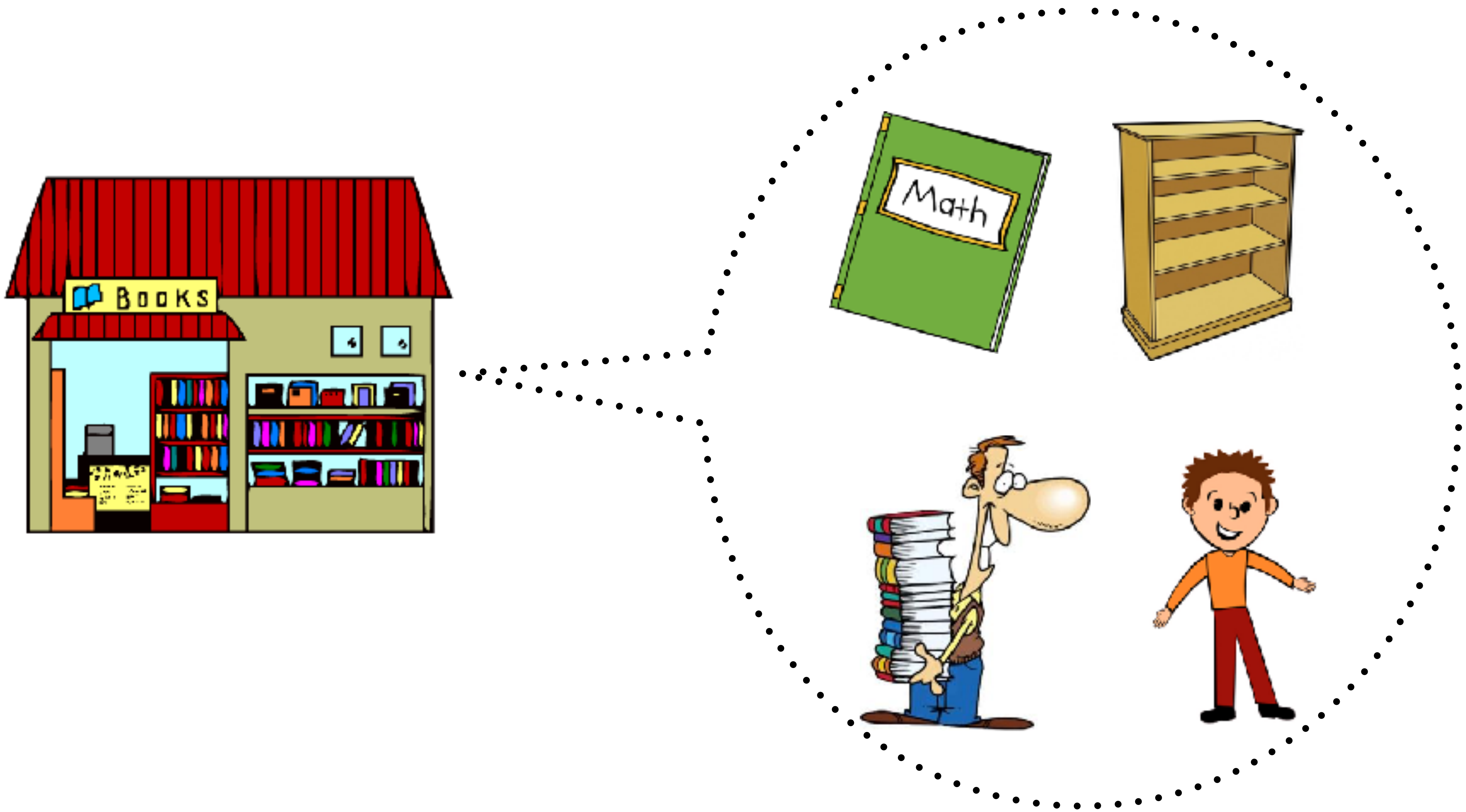
Object-Oriented Approach

- Object-oriented programming is centered on creating objects rather than procedures/functions
- Objects are a melding of data and procedures that manipulate that data
- Data in an object are known as attributes
- Procedures/functions in an object are known as methods/operations

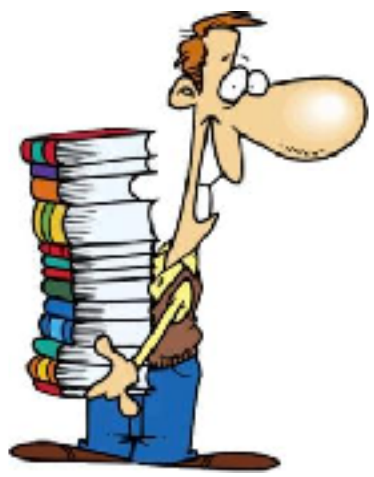
Objects



Object-Oriented Approach: Example



Object-Oriented Approach: Example (Cont'd)

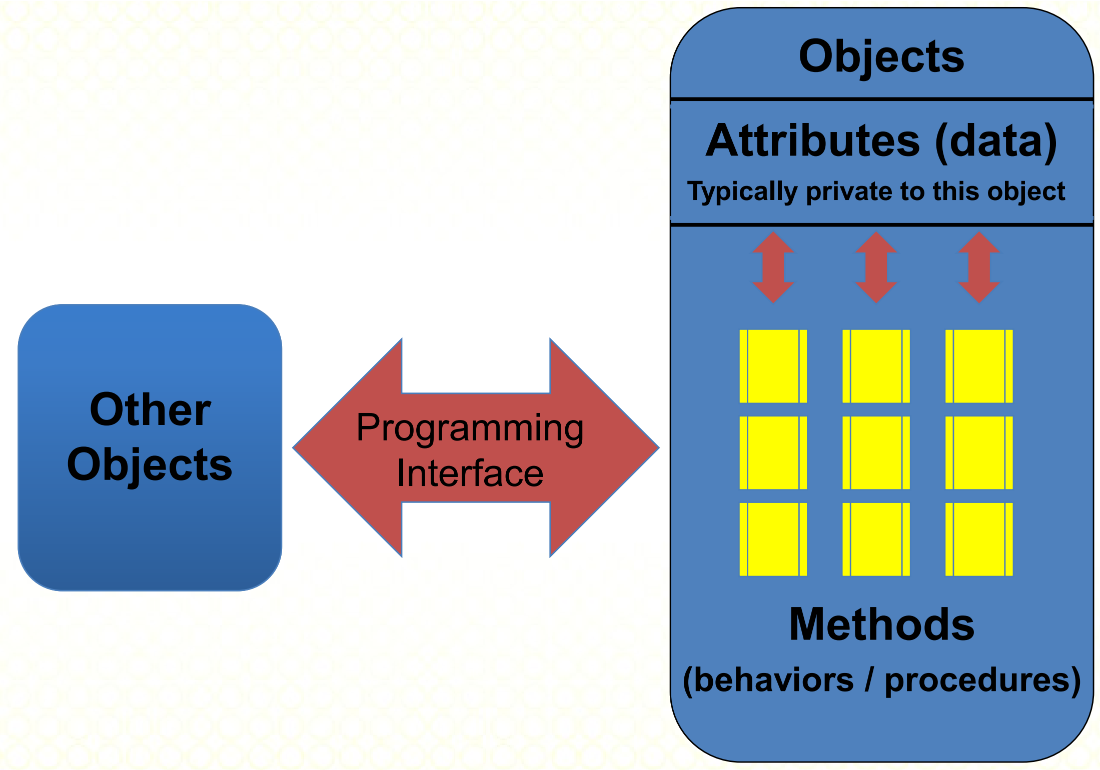


Attributes: id, name, age, ...
Operations: sell books, ...

Attributes: id, size, ...
Operations: load/remove books, ...

Object-Oriented Approach

- What is the object-oriented approach
  - Software is viewed as an set of objects interacting with each other
  - An object ask another object (providing inputs) to get the information it wants (getting output)



Key steps in OOA

- Define the domain model
  - Find the objects, classes
- Define class diagram
  - Find relationships between classes (static)
- Define the interaction diagrams
  - Describe the interaction between the objects (dynamic)

Objects

- Definition
  - Discrete entities with well defined boundaries
  - Data
  - Operations
- Life Cycle
  - Construction (new shelf bought)
  - Running (loading, removing, and moving books)

    Runtime state: value of mutable data in the object
  - Destruction (shelf broken or removed)

Classes

- Too many objects
  - Cannot define one by one in the system
  - Not general enough

    Consider three clerks in the bookstore (John, Mike, Nancy)
- Objects share similar features can be grouped as a Class
  - John, Mike, Nancy --> Sales Clerk
  - Thousands of different books --> Book
- Class is a natural concept in our mind

OOA: Pros

- Code reuse and recycling
- Encapsulation: Objects have the ability to hide certain parts of themselves from programmers
- Design benefits: OO Programs force designers to go through an extensive planning
- Post-implementation benefits: Good design facilitates software maintenance and debugging

OOA: Cons

- Steep learning curve
- Larger program size
- Slower programs
- Not suitable for all types of programs

OOA: In Practice

- Most widely used programming paradigm
- Language supports
  - Smalltalk
  - Eiffel
  - Java
  - JavaScript
  - C#
  - C++
  - PHP
  - Objective-C

UML Class Diagram

- A diagram to describe classes and relations
  - Core part in UML and OOA
  - Used as a general design document
  - Maps to code directly in OO programming languages
  - Modeling the system in a more visualized way

UML Class Diagram Syntax

- Elements of class diagram:
  - Class represented as a box containing three compartments

    Name

    Attributes

    Operations
  - Relation represented as a line between two classes

    Association

    Generalization

    Aggregation and composition

Class

- Classes are named, usually, by short singular nouns
- Names start with capitalized letter
- Legend: A box with three compartments for names, attributes, and operations respectively

Class

- Attributes
  - Visibility (+: public, -: private)
  - Name (lowercase start)
  - Type
- Operations
  - Visibility (+: public, -: private)
  - Name (lowercase start)
  - Parameters (in/out, name, type)
  - Return Type

Class Diagram – Class

Shelf
-id : string -size : int -aisle : int -row : int
+loadbook(in book : Book) : bool +removebook(in book : Book) : bool +countbook() : int

Identifying Class

- Classes are entities from the problem domain
  - Actors that interact with system

    e.g., Sales Clerk
  - Concrete objects with some information

    e.g., Books, shelves
  - Abstract objects

    e.g., transactions, orders, etc.
  - Structured Outputs

    e.g., forms, reports
  - Helper Classes

    e.g., utility, logger, order manager, etc.

Identifying Class

- Classes are usually derived from the use cases for the scenarios currently under development
- Brainstorm about all the entities that are relevant to the system
- Noun Phrases
  - Go through the use cases and find all the noun phrases
  - Watch out for ambiguities and redundant concepts
  - Subtypes of a class may also be a class

    e.g., Member is a subtype of Customer

Identifying Class

- Not too many
  - Poor performance
  - Complexity
  - Maintenance efforts
- Not too few
  - Class too large, poor performance

    Have a class BookStoreSystem and do everything
  - Uneasy to reuse

    Class Publisher : may be used in both book information, and order

    If no such class, may have to implement twice

Class Relationships

- Generalization
- Aggregation & Composition
- Association

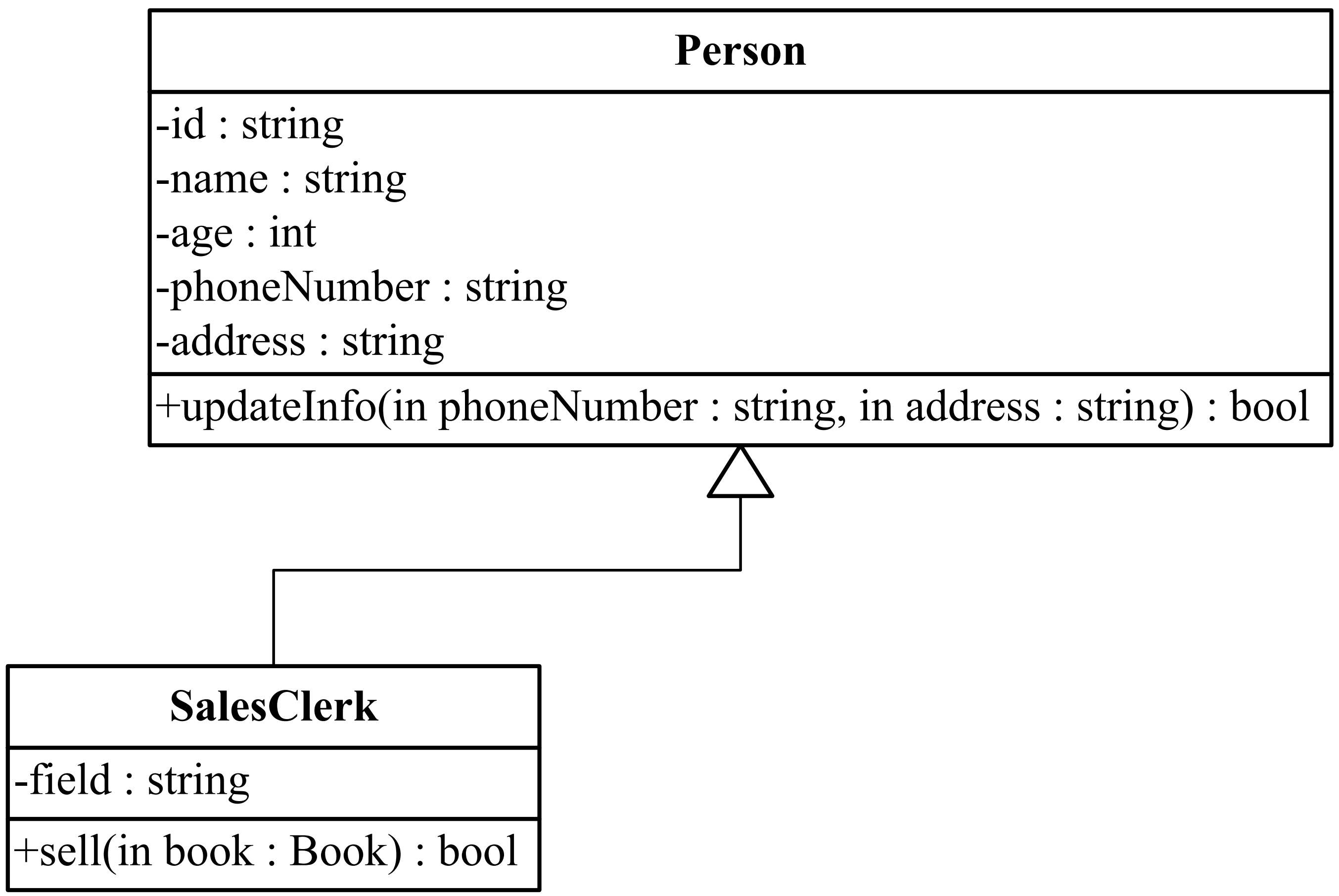
Generalization

- Indicates an “is-a” relationship
- All instances of the subclass are instances of the super class
- A subclass inherits all attributes, operations and associations of the parent : enabling reuse
- Example:
  - Member “is a” customer
  - Fruit “is a” kind of food

Generalization: syntax

- Arrow pointing (hollow triangle shape) at the super class end of the line
- The common attributes and operations are placed in the super class;
- Subclasses extend the attributes, operations, and relations as they need them

Generalization: example



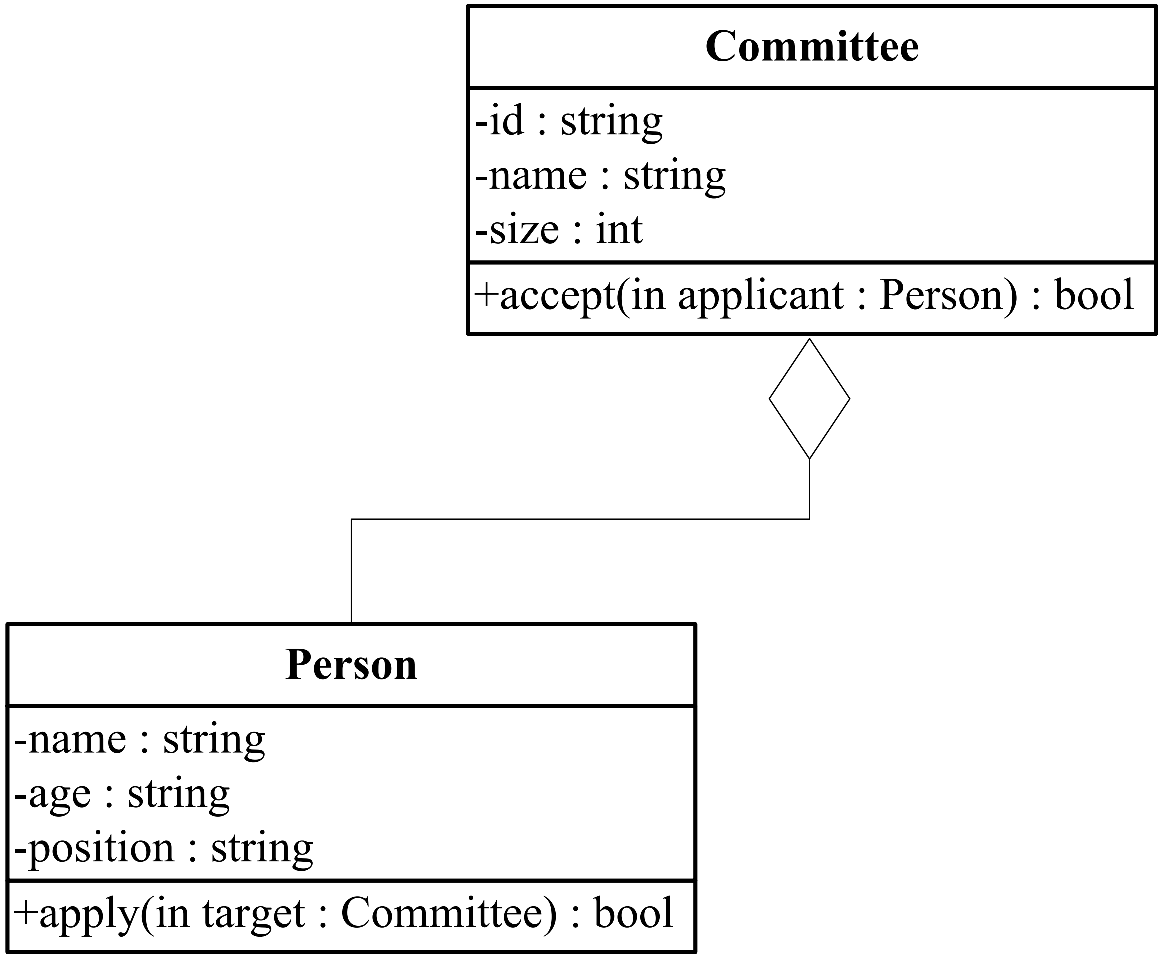
Aggregation

- Indicates a loose "has-a" relationship
- The compound class is made up of its component classes
- Represents the "whole-part" relationship, in which one object consists of the associated objects
- Syntax: hollow diamond at the compound class end of the association
- Example:
  - Committee "has a" person

Aggregation Semantics

- Whole can exist independently of the parts
- Part can exist independently of the whole
- It is possible to have shared ownership of the parts by several wholes

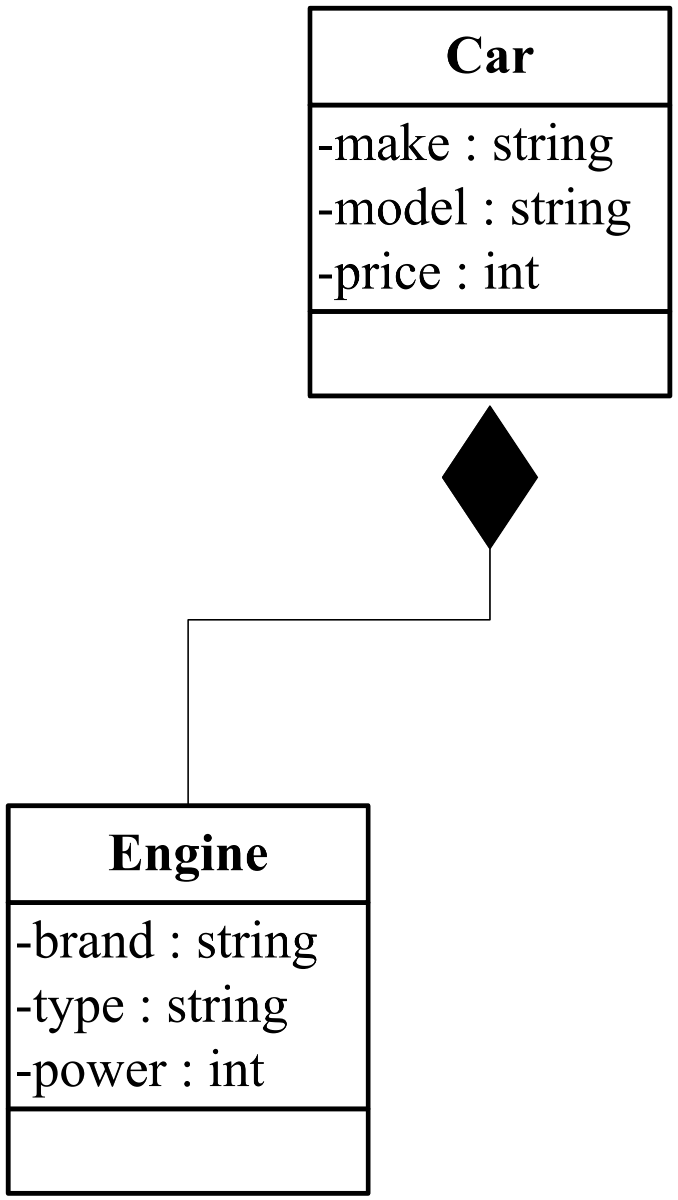
Aggregation Example



Composition

- Composition also describe "has a" relationship
- Component classes are physically part of the compound class
- The component class dies if the compound class dies
- Syntax: filled diamond at the compound class end of the association
- Example:
  - Car : Engine

Composition: Example



Aggregation vs. Composition

- The lifecycle of components is controlled by the compounds in Composition but not Aggregation
- A component usually can be shared by different compounds in Aggregation but not Composition
  - Aggregation means "use"
  - Composition means "owns"

A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation). When the Text Editor is closed, the Buffer is destroyed but the File itself is not destroyed.

```
public class Composition{
    Component c;
    public Composition (){
        this.c = new Component();
    }
}
public class Aggregation{
    Component c;
    public Aggregation (Component
comp){
        this.c = comp;
    }
}
```

Composition vs. Aggregation

- Examples:
  - University: Department
  - Class: Student

Association

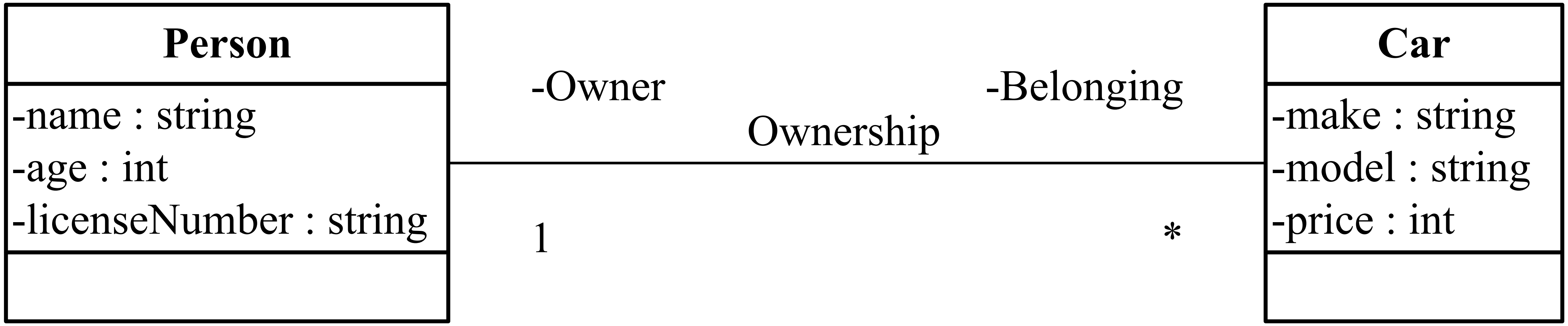
- An association is a relationship between classes
- An association is a name, usually short verb
  - Some people name every association
  - Others name associations only when such names will improve understanding

    e.g., avoid names like "is related to", and "has"
- An association represents different types of relationships
  - e.g., student take course, book on the shelf, etc.

Association Syntax

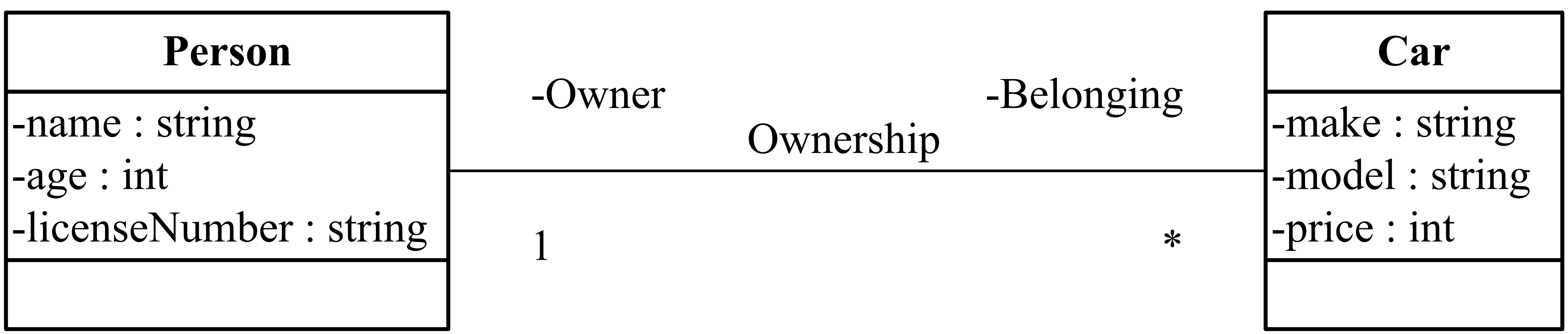
- An association may have
  - An association name
  - Multiplicity
  - Role names

Association Example



Multiplicity

- Multiplicities give lower and upper bounds on the number of instances of the local class that can be linked to one instance of the remote class
- Multiplicities indicate the number of instances at runtime (i.e., objects)



Multiplicity

- Syntax: 1, *, etc. at the association end
- Examples:
  - * (zero or more)

    Person : Car
  - 1 .. * (one or more)

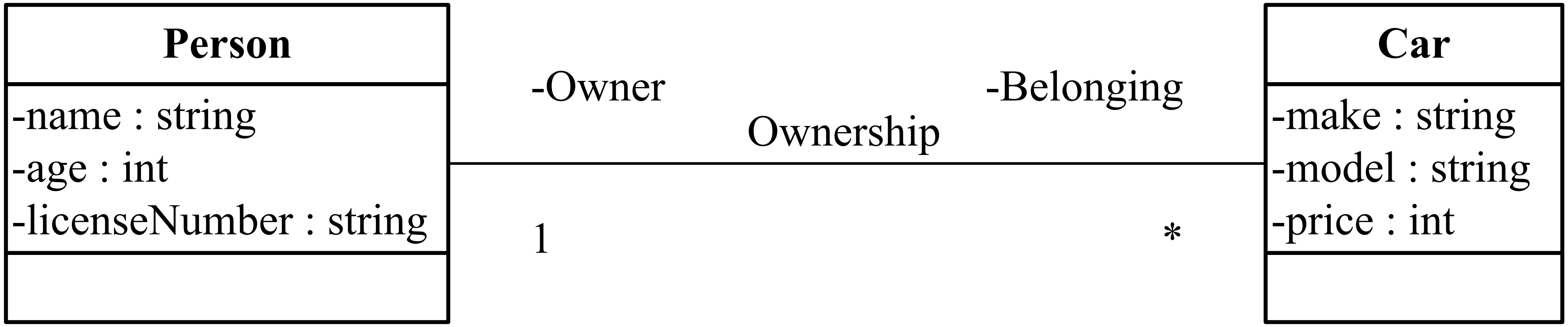
    Person : Address
  - 5 .. 40 (5 to 40)

    Students : Course
  - 10 (exactly 10)

    Referee: Basketball Player
  - If no multiplicity is specified, the default is 1

Role Name

- Is a part of association
- Describes how the object at the association end is viewed by the associated object
- Is useful for specifying the context of the class
- Syntax: name at the association end



Review of Class Diagram

- Class is a group of objects with same features within the context of the system
- Class diagram describes classes and their relations
- Identify classes
  - Actors
  - Concrete / Abstract objects
  - Structured Outputs
  - Helper for utils
  - Subtype

Review of Class Diagram (Cont'd)

- Class
  - Name, Attributes, Operators
- Relations
  - Generalization
  - Aggregation and Composition

    Aggregation vs. Composition
  - Association

    Name, multiplicity, role name

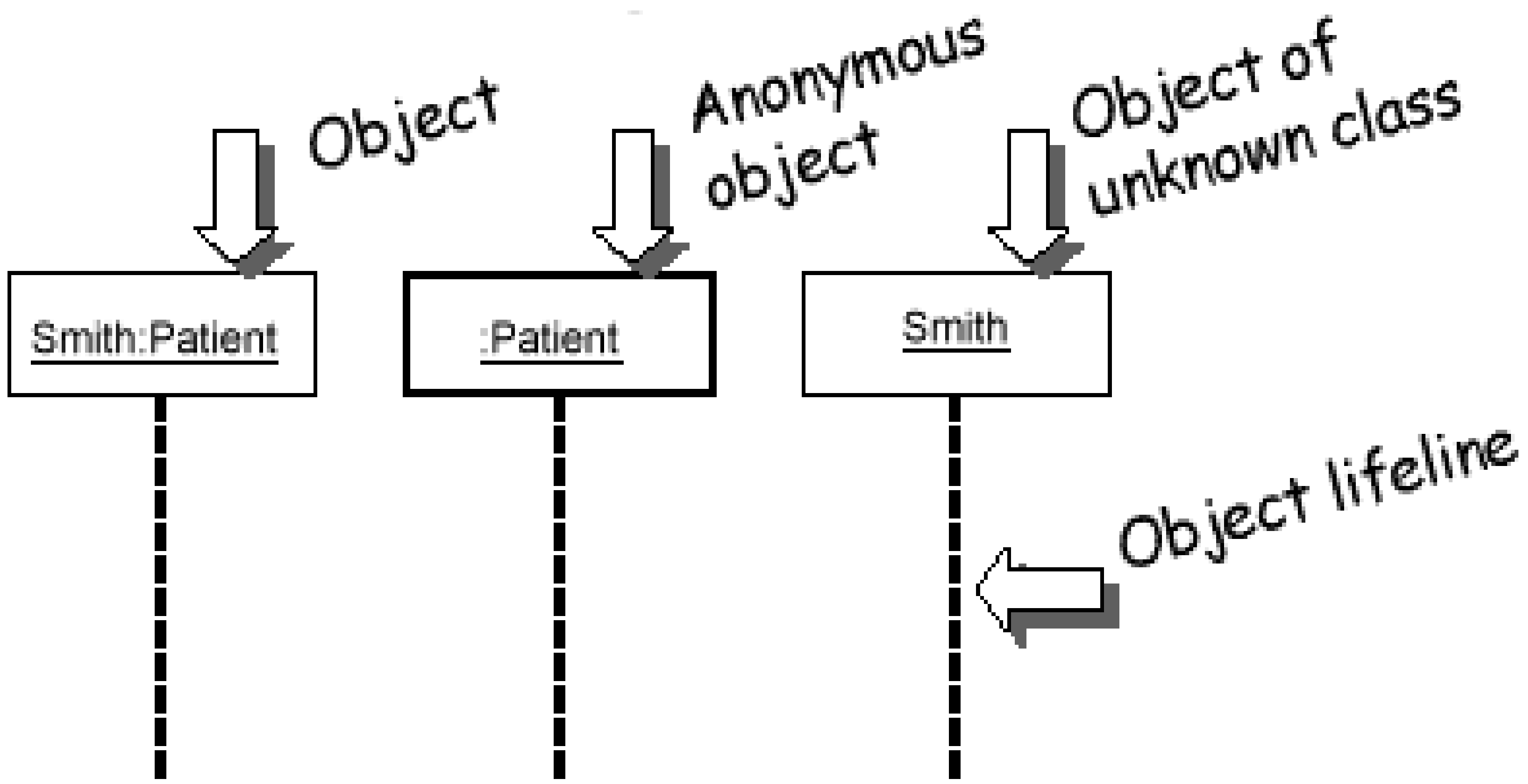
Sequence Diagram

- Class Diagram describe the static structure of a software
- Need to know how objects will interact with each other
- Sequence Diagram describes how objects talk with each other dynamically
- Sequence diagram emphasizes the time-ordered sequence of messages sent and received

Object and Lifeline

- Column is an instance of the class
  - Naming: [instance]:[class]
  - "instance" and "class" are optional
- Vertical dashed line is lifeline of the instance

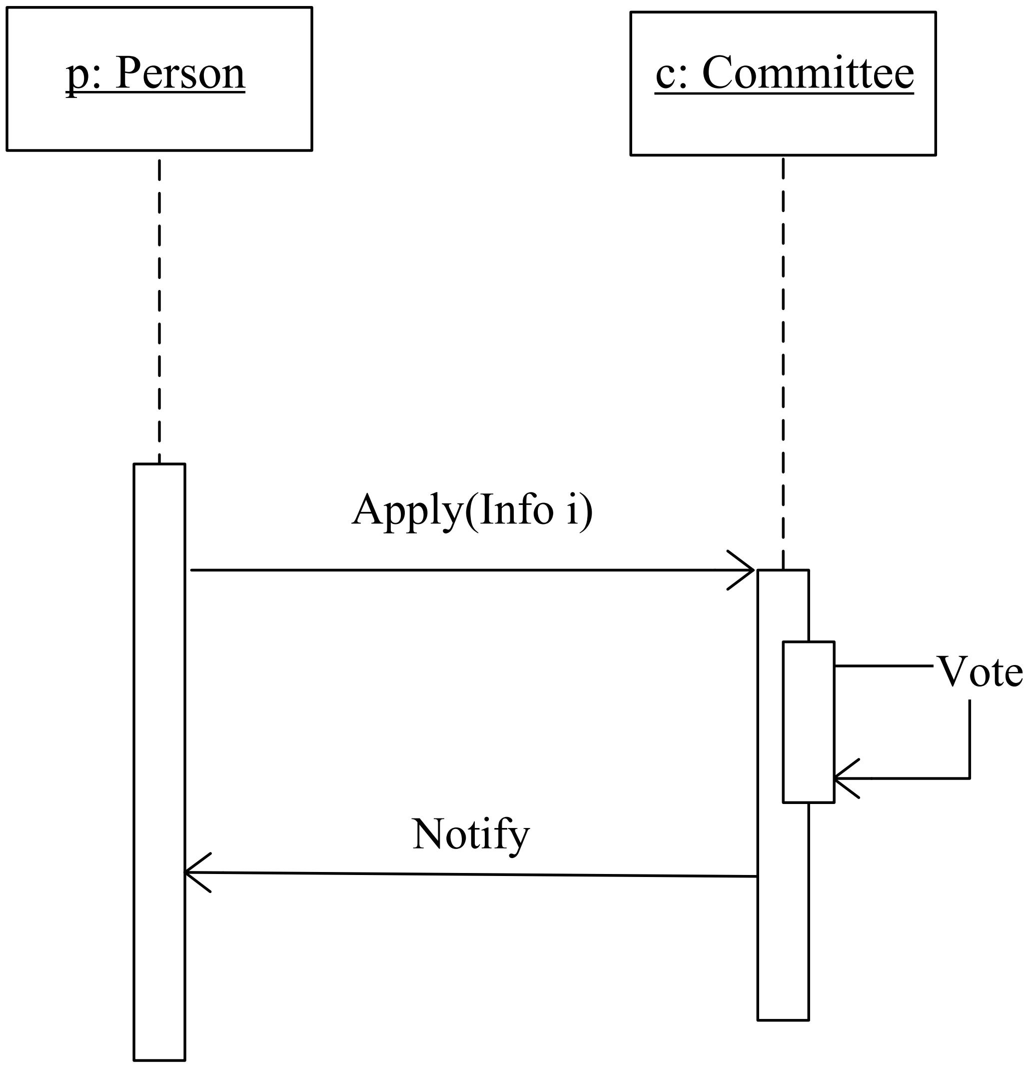
Object and Lifeline - Example



Message

- Horizontal arrow expresses messages conveyed by source instance to target instance
- Messages may carry parameters: msg (par1, …)
- Looping arrow shows self-delegation: a lifeline sends a message to itself
- Rectangle on life line is the focus of control (or execution), i.e., the duration of the execution of a method in response to a message

Message - Example

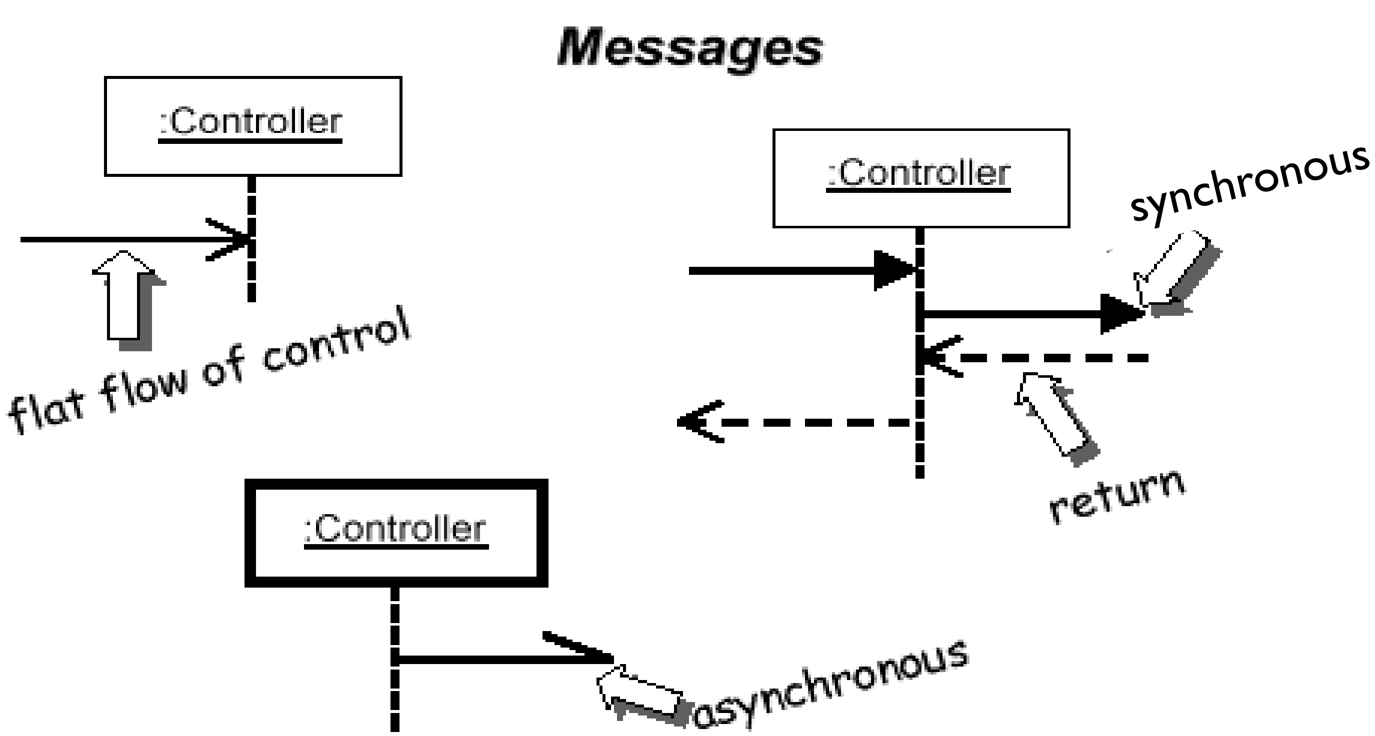


Why use objects instead of classes?

- Class is a static concept
- Only objects have life cycles
- Objects of same class may interact

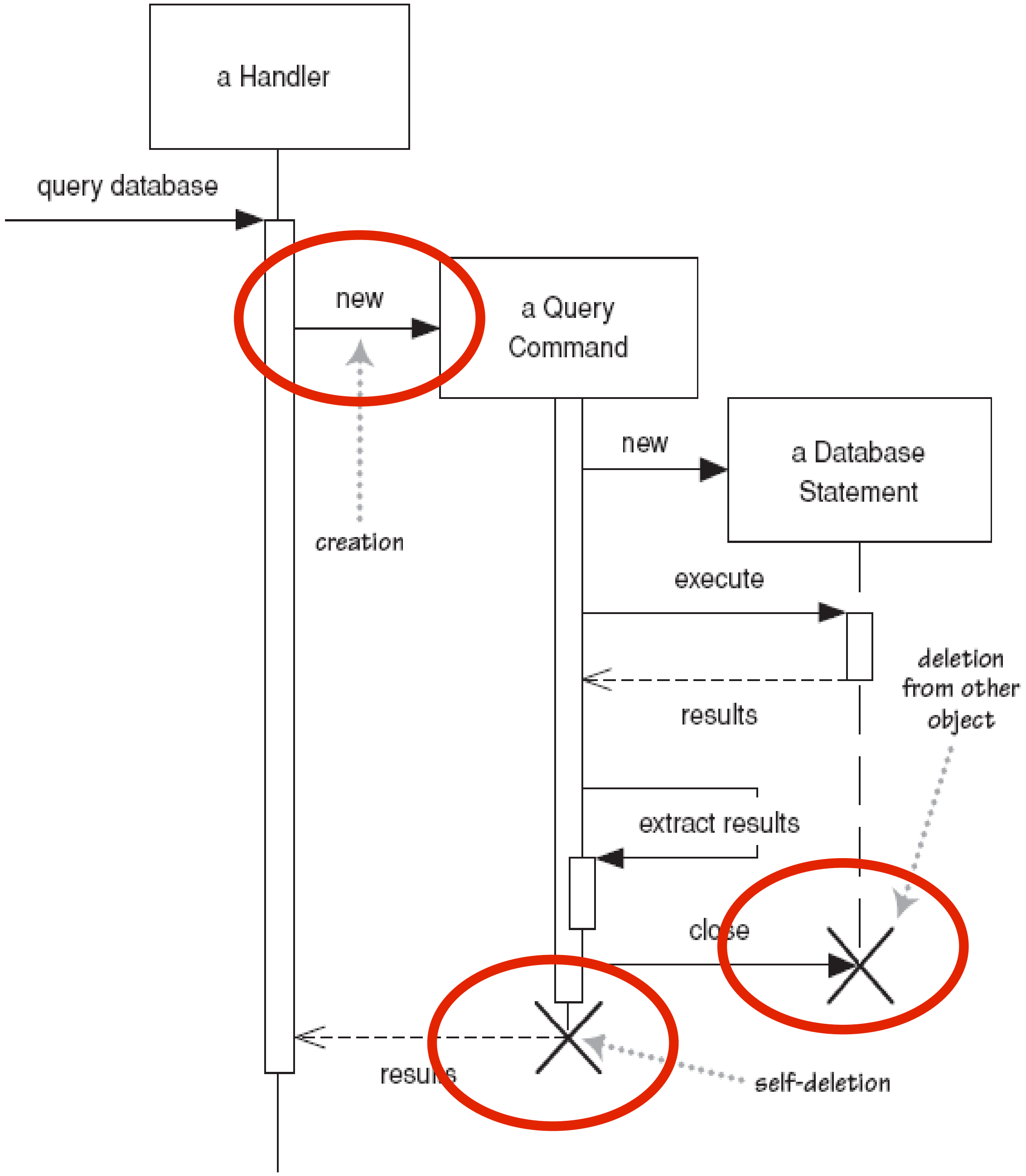
Different Message Types

- Types of messages
  - Different arrowheads for normal / concurrent (asynchronous) methods
  - Dashed arrow back indicates return (can be optional)



Lifetime of Objects

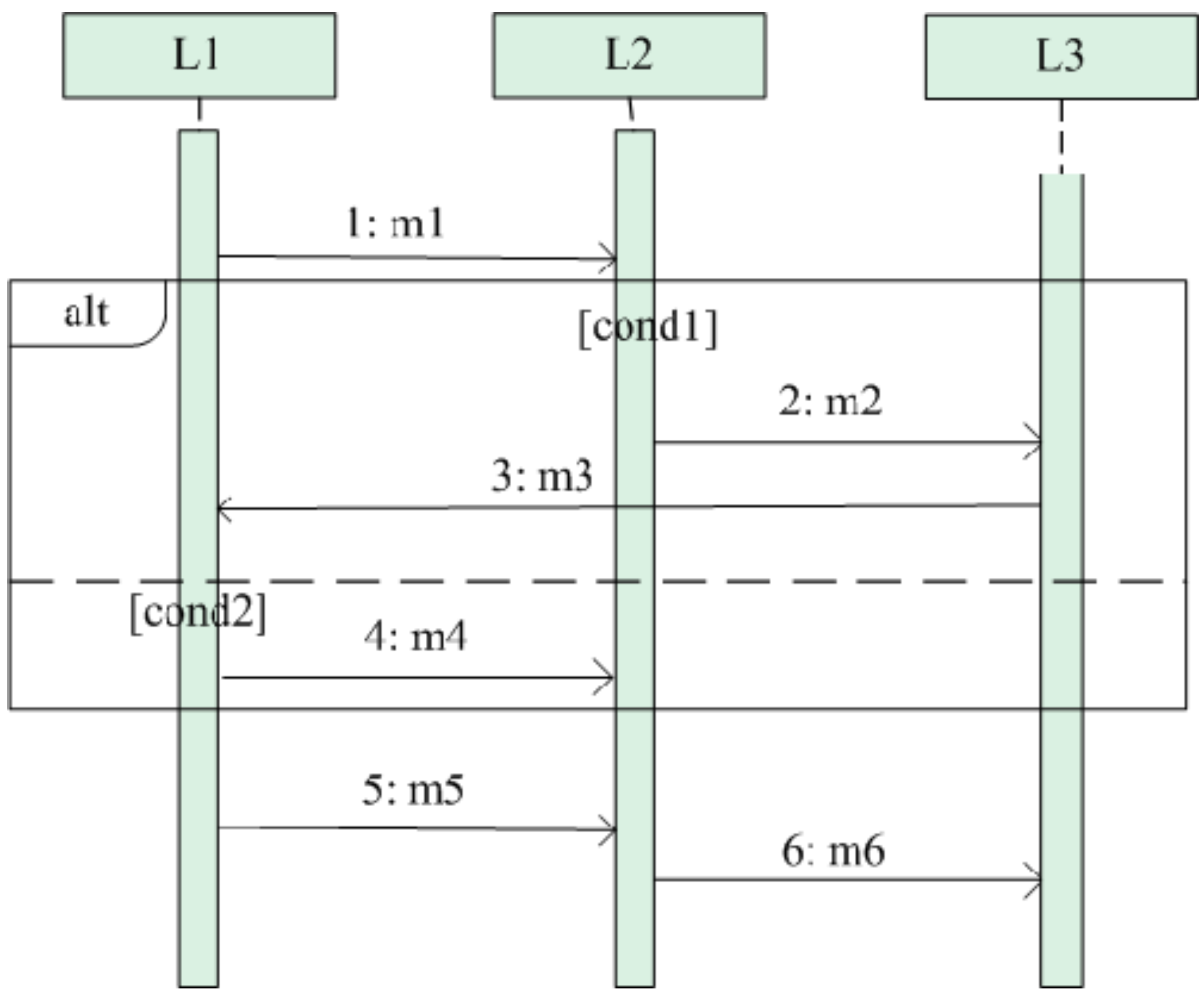
- Creation: arrow with 'new' written above it
  - An object created after the start of the scenario appears lower than the others
- Deletion: an X at bottom of object's lifeline
  - Java doesn't explicitly delete objects; they fall out of scope and are garbage- collected



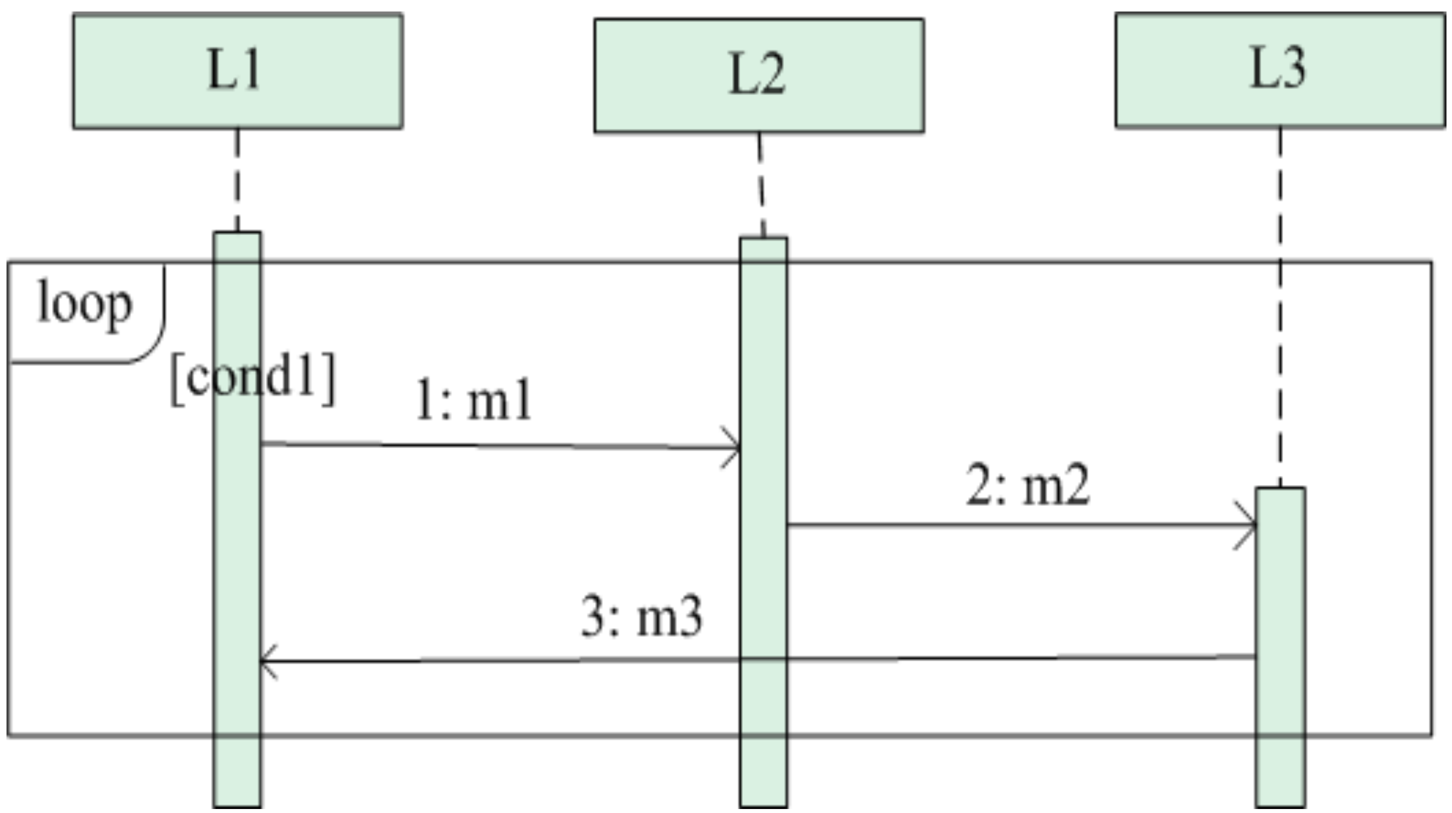
Sequence Diagram – Advanced Features

- Use combined fragments, which consists of a region of a sequence diagram, to represent
  - Branching: operator "alt"
  - Loop: operator "loop"
  - ...

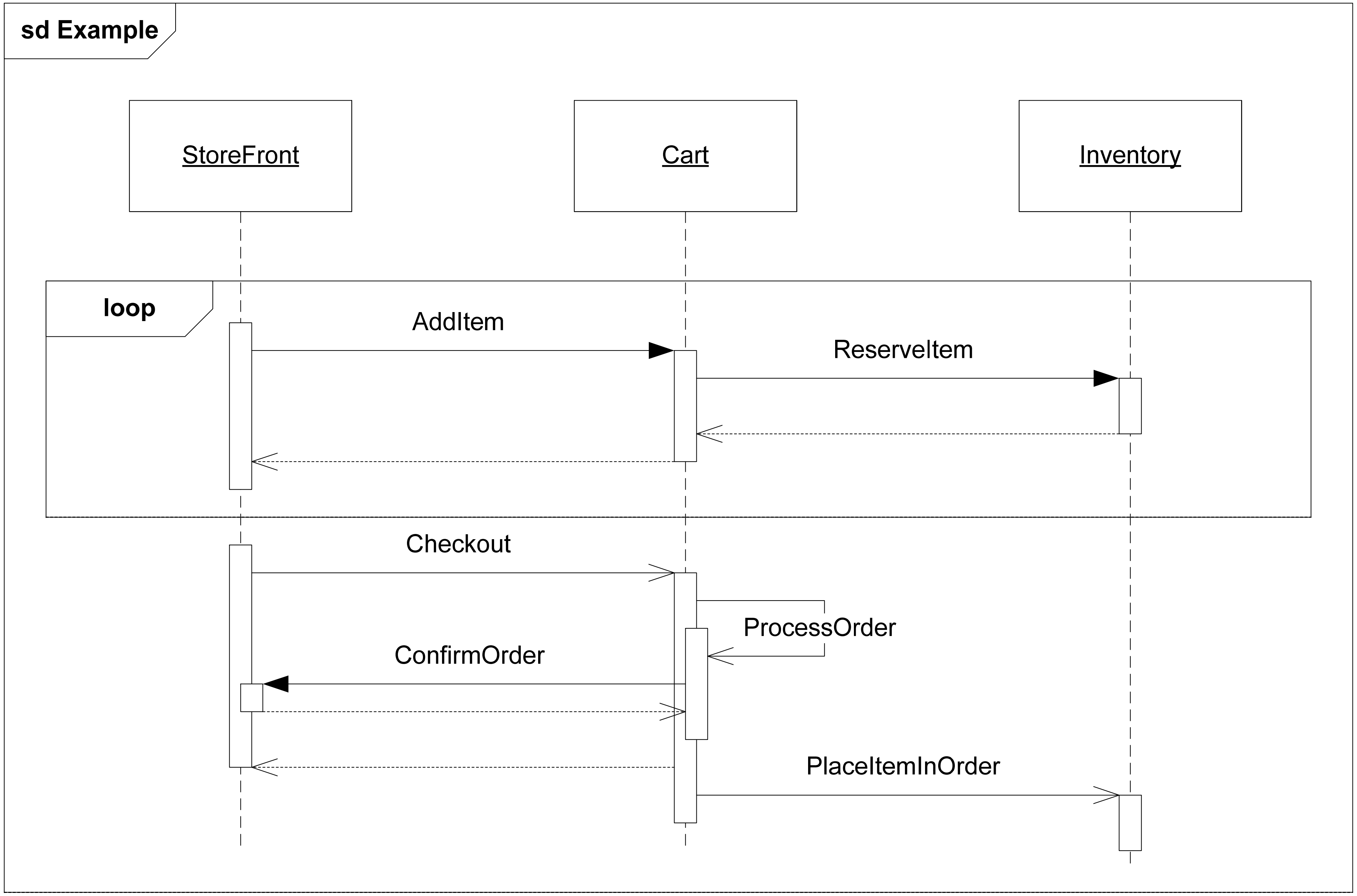
Alternative



Loop



Sequence Diagram - Last Example



Apply UML

- UML as sketch
  - informal and incomplete diagrams (often hand drawn on whiteboards) created to explore difficult parts of the problem or solution space
  - emphasized in agile modeling
- UML as blueprint
  - relatively detailed design diagrams used for reverse engineering or code generation
- UML as programming language
  - complete executable specification of a software system in UML

"Obvious" Design Rules?

- Emphasis in this class on class and sequence diagrams.
  - need both static and dynamic views of the design
- There should be (at least one) sequence diagram for each use case.
- Every class defined should occur in a dynamic (sequence) diagram – it should DO something.
- Every communicating object in a dynamic (sequence) diagram should have been defined in the class diagram.

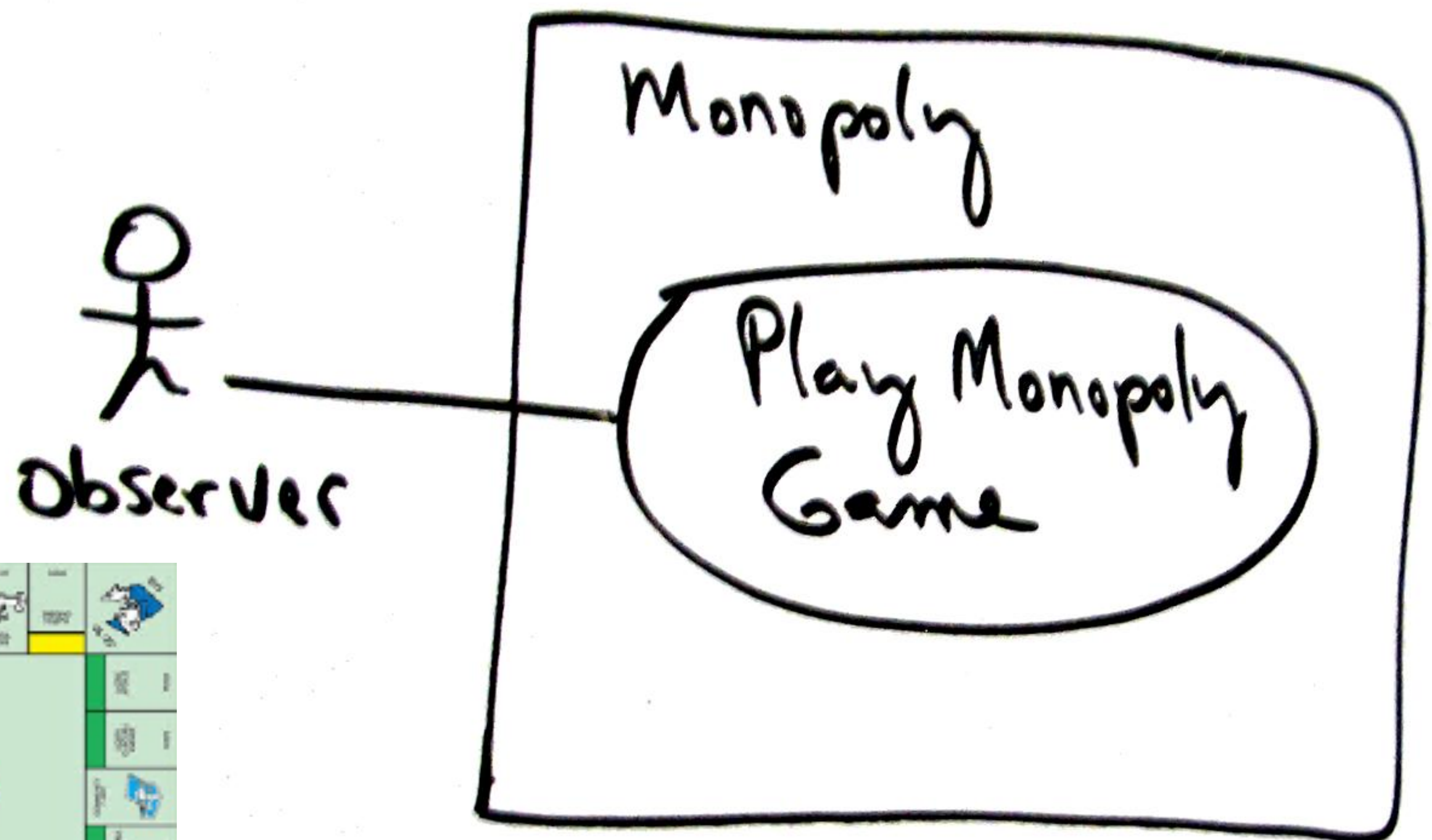
Object Oriented Analysis (OOA)

- Create a model of the system's functional requirements.
  - need both static and dynamic views of the design
- In OOA, organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects.

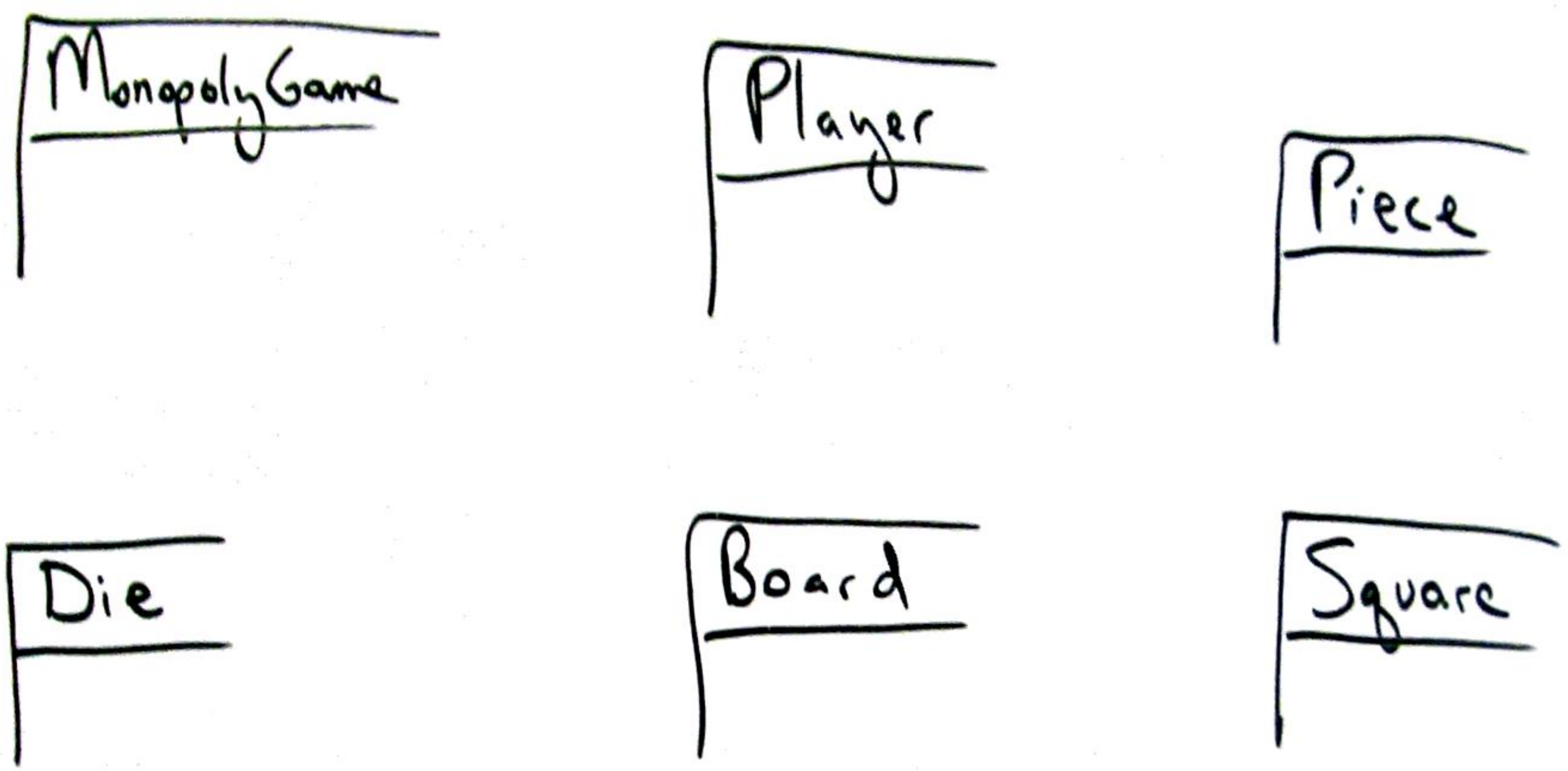
Object Oriented Analysis (OOA)

- The primary OOA tasks
  - find the objects
  - organize the objects
  - describe how the objects interact
  - define the behavior of the objects
  - define the internals of the objects

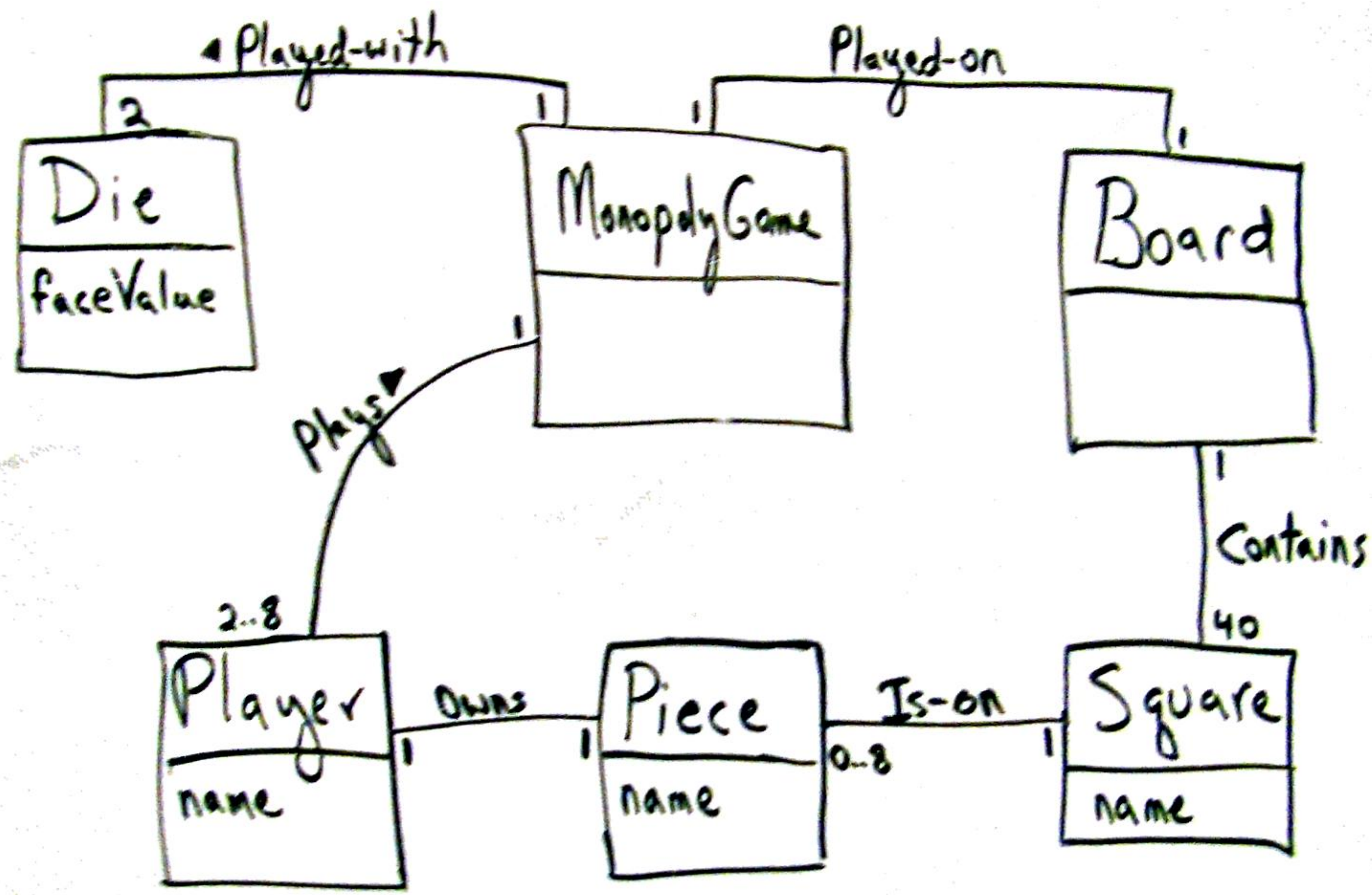
Monopoly Case Study (Larman)



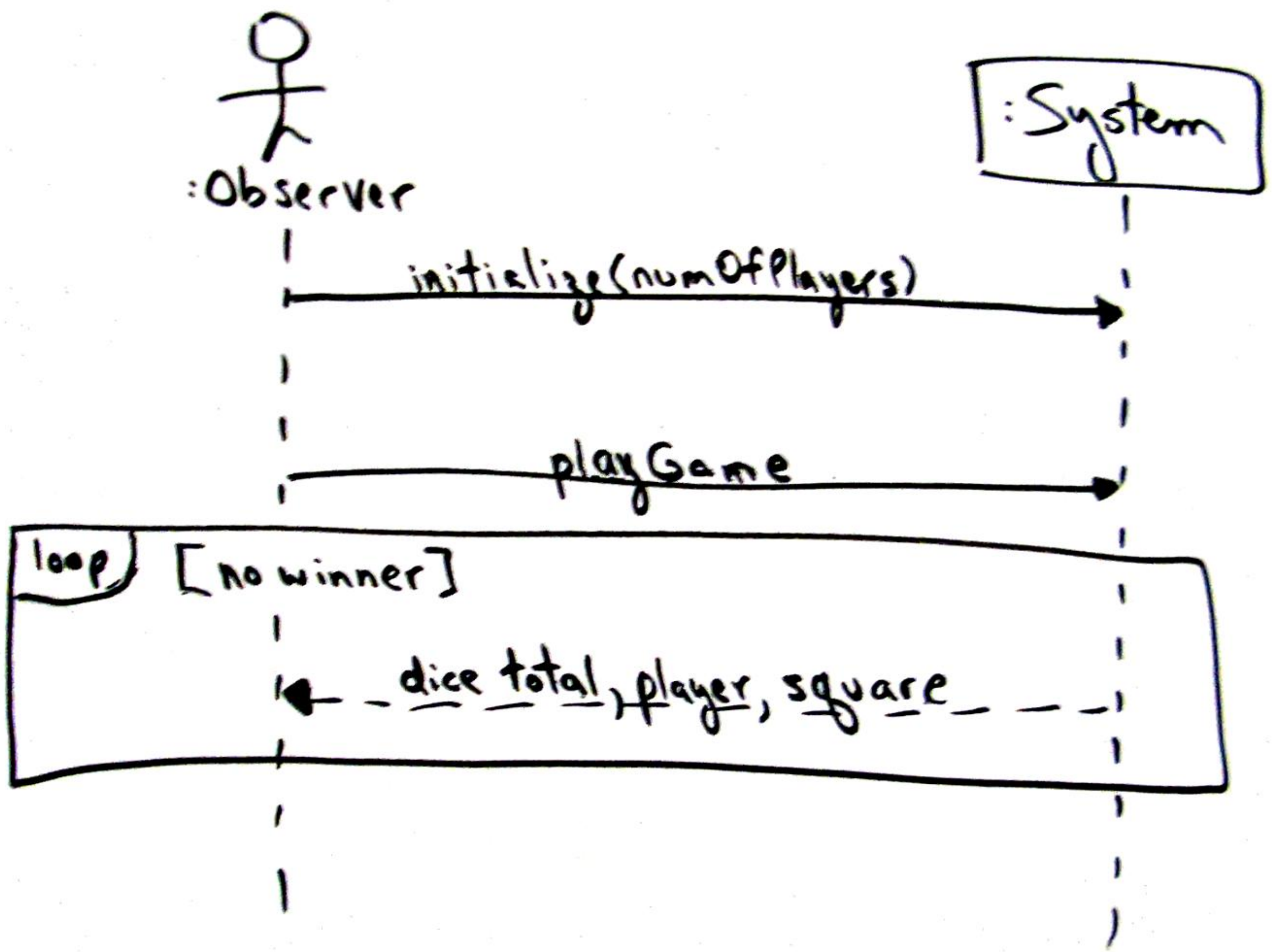
Monopoly Case Study



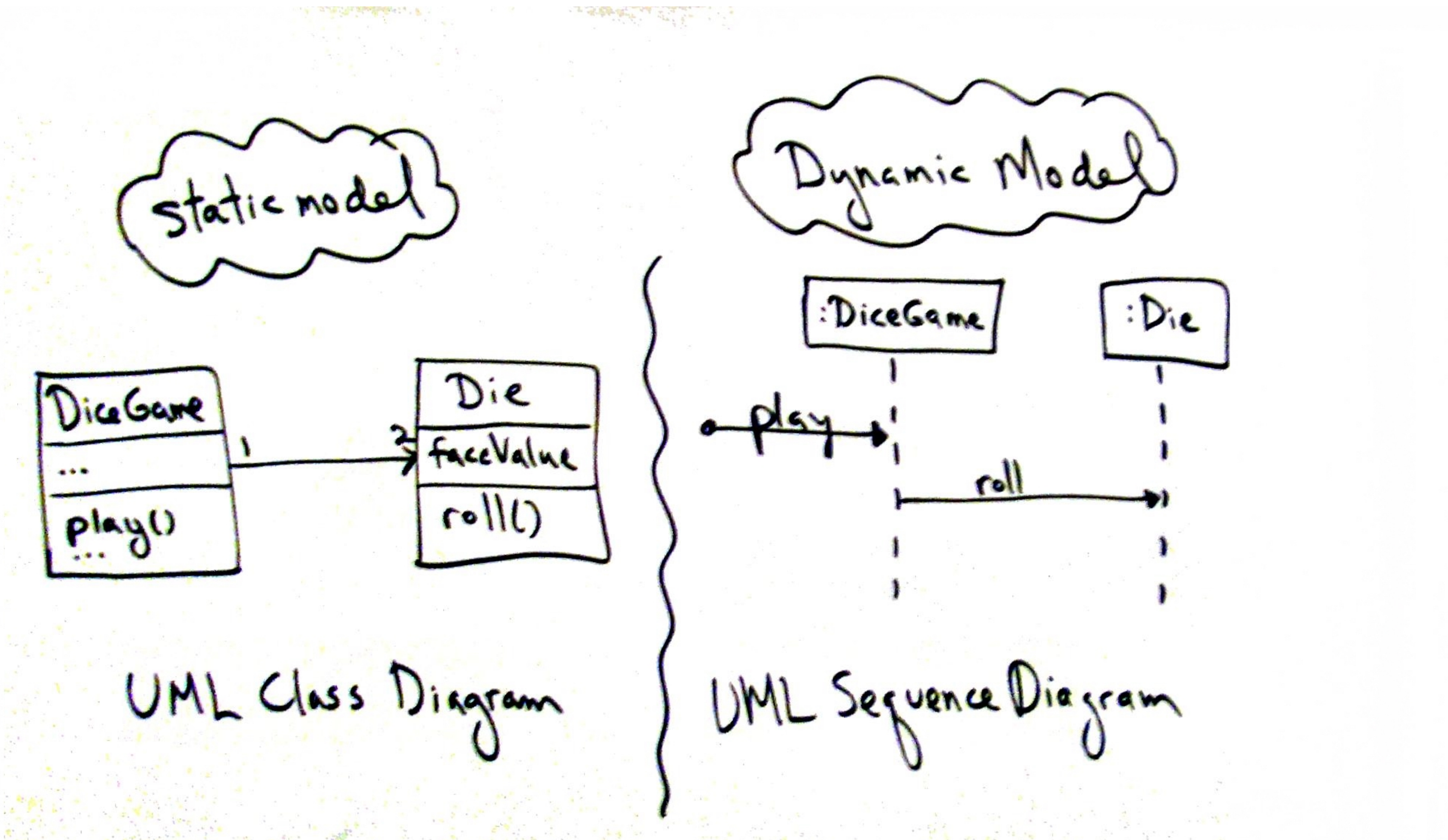
Monopoly Case Study



Monopoly Case Study



Monopoly Case Study



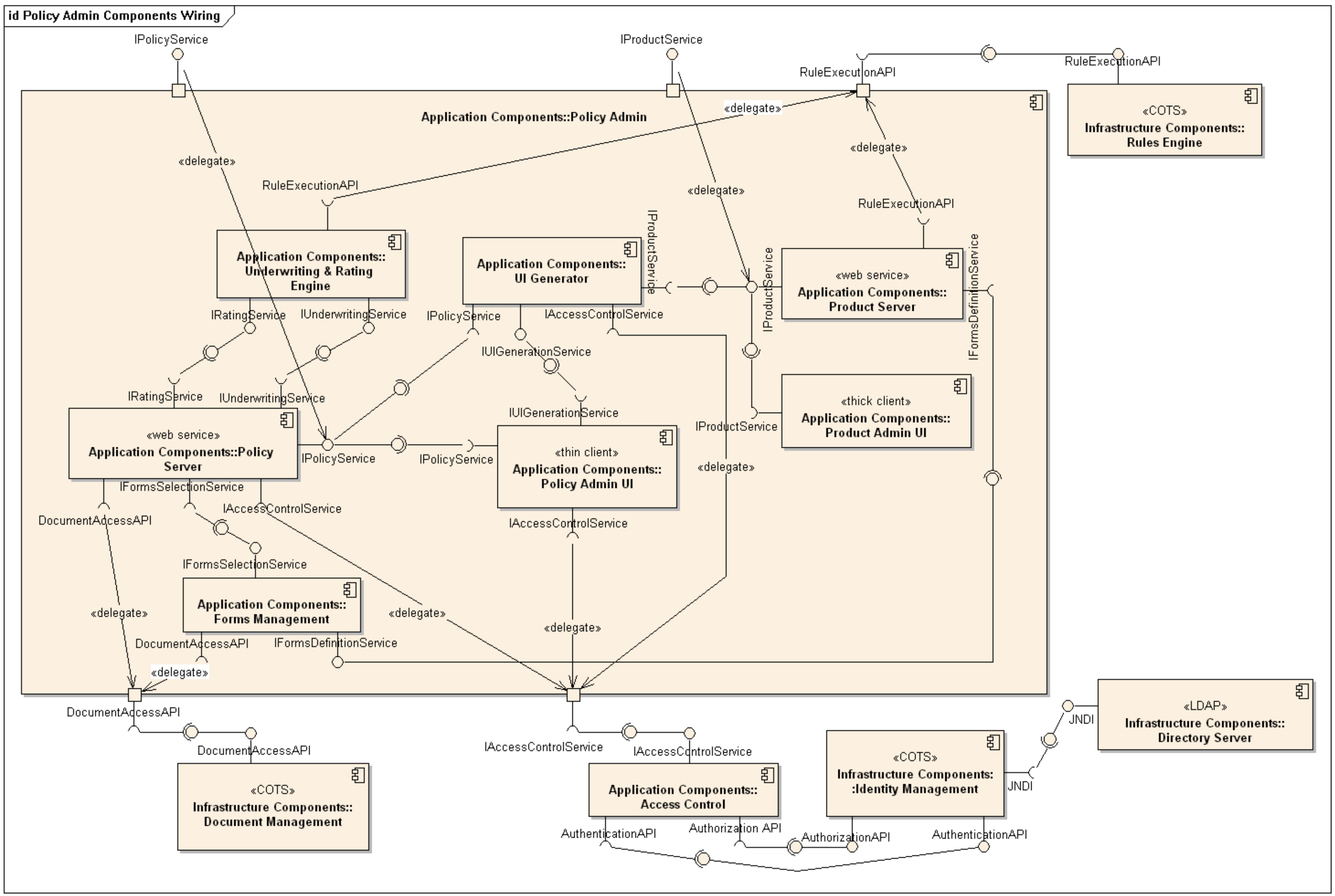
A Good Design Practice

- Recommendation: move back and forth between the static and dynamic views of the design.
- For example, work on the class diagram some, then work on the sequence diagram, back to the class diagram, return to the sequence diagram, etc.
- Note that you will typically have a sequence diagram for each use case.
    - you may have multiple class and sequence diagrams at different levels of abstraction…

Component Diagram

- Describes how a software system is split up into components and shows the dependencies among these components.
- Components are wired together by using an assembly connector to connect the required interface of one component with the provided interface of another component.

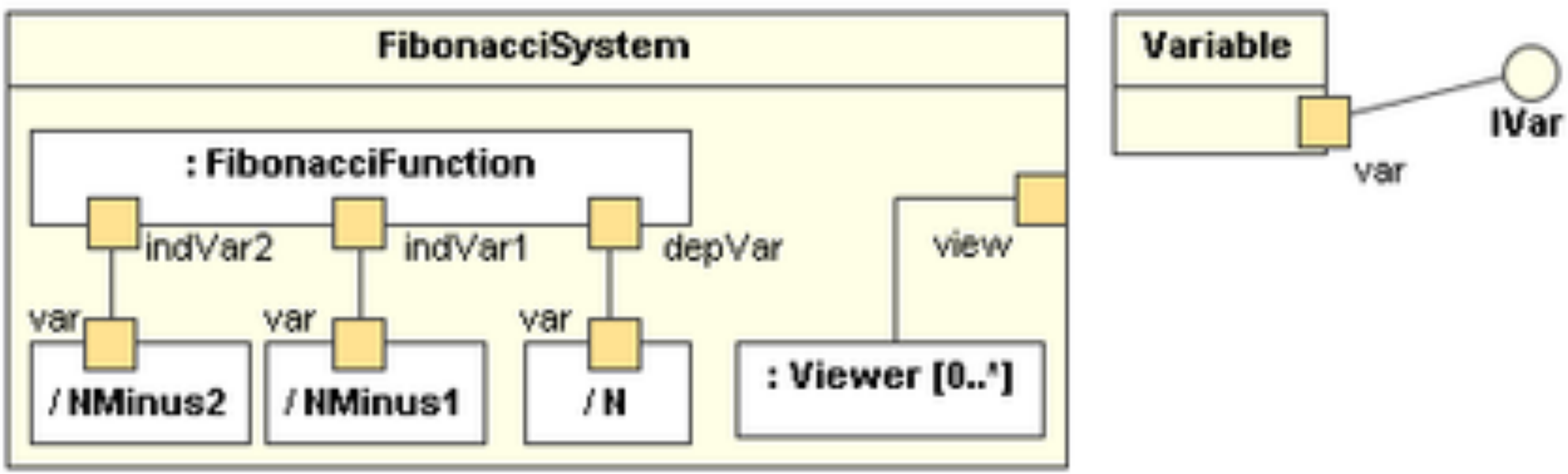
Component Diagram



Composite Structure Diagram

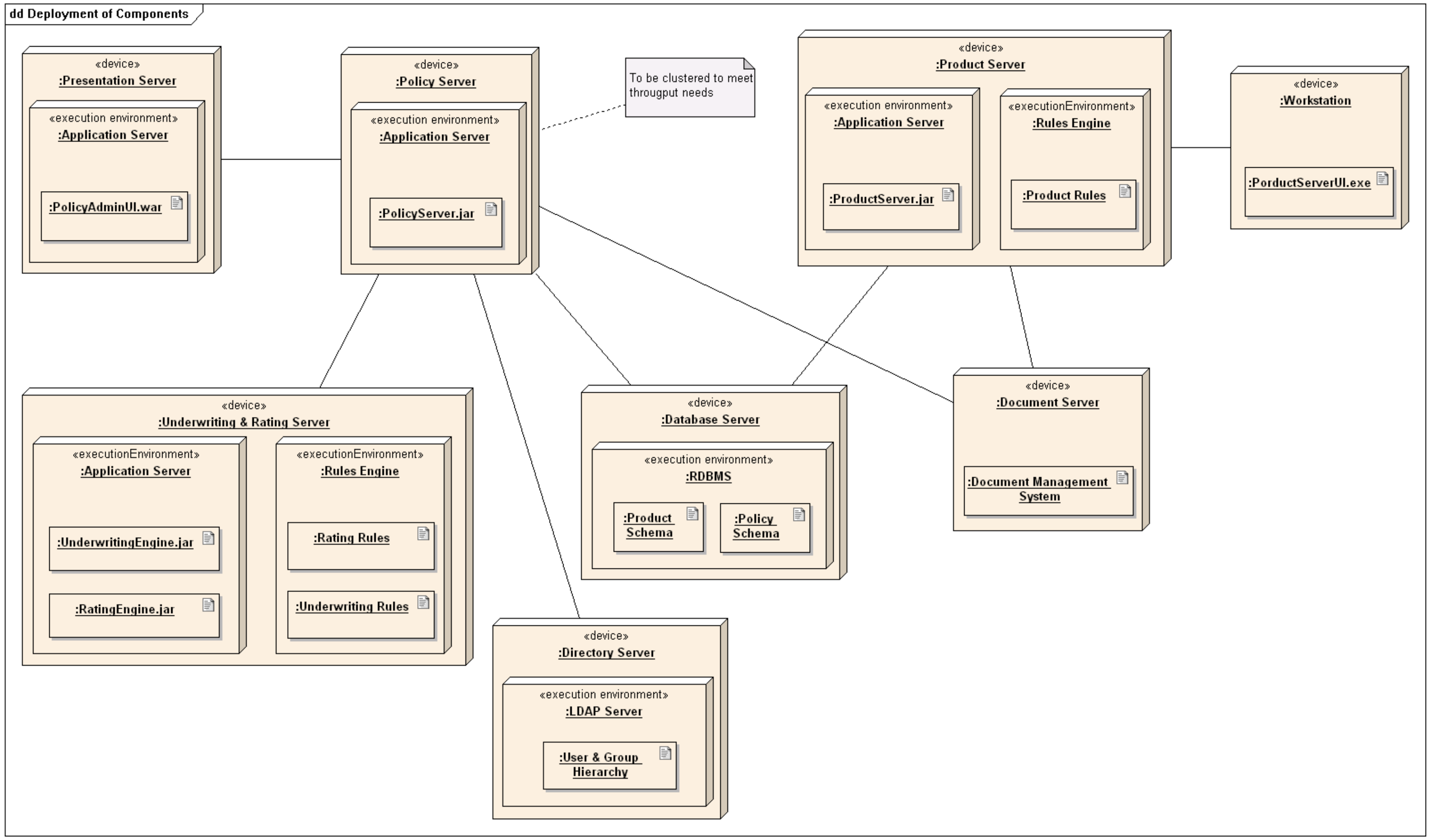
- Describes the internal structure of a class and the collaborations that this structure makes possible.
- Can include internal parts
  - ports through which the parts interact with each other or through which instances of the class interact with the parts and with the outside world,
  - connectors between parts or ports

Composite Structure Diagram



Deployment Diagram

- Describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.

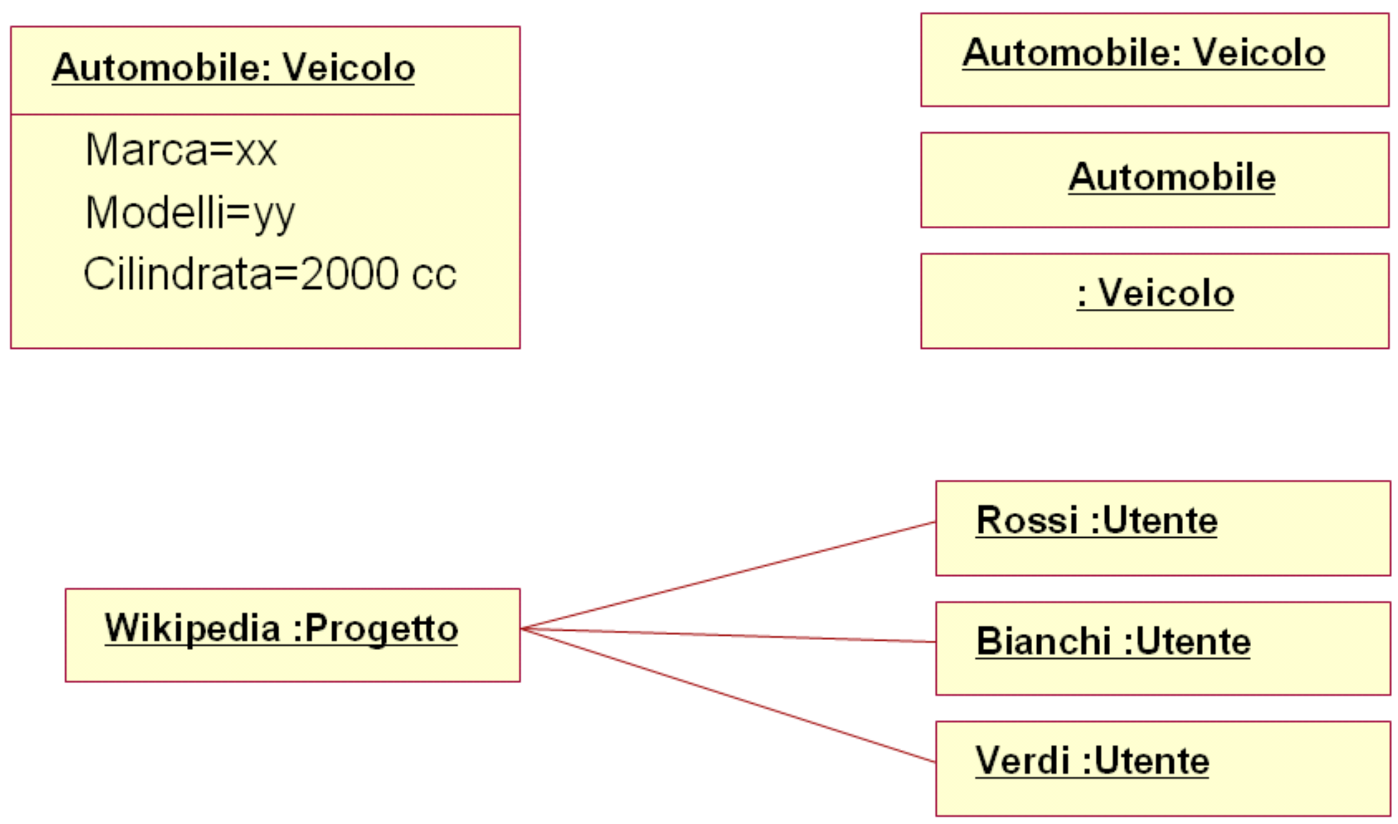


Object Diagram

- Shows a complete or partial view of the structure of an example modeled system at a specific time.
- “An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, namely to show examples of data structure.”

Object Diagram

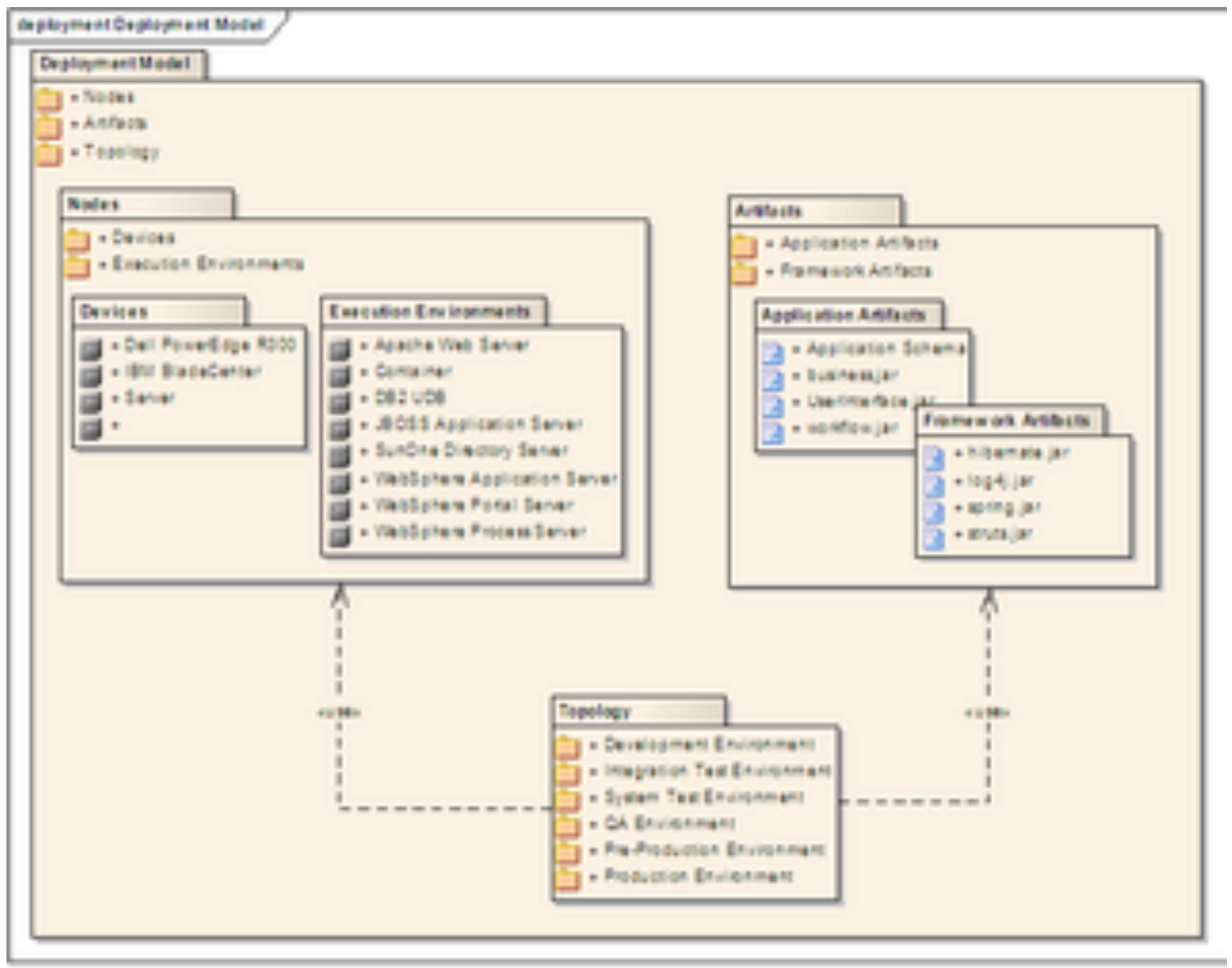
Object Diagram



Package Diagram

- Describes how a system is split up into logical groupings by showing the dependencies among these groupings
  - package: a general purpose mechanism for organizing model elements and diagrams into groups
  - class: usually describe logical structure of system
  - interface: a specification of behavior
  - object: an instance of class
  - table: a stereotyped class

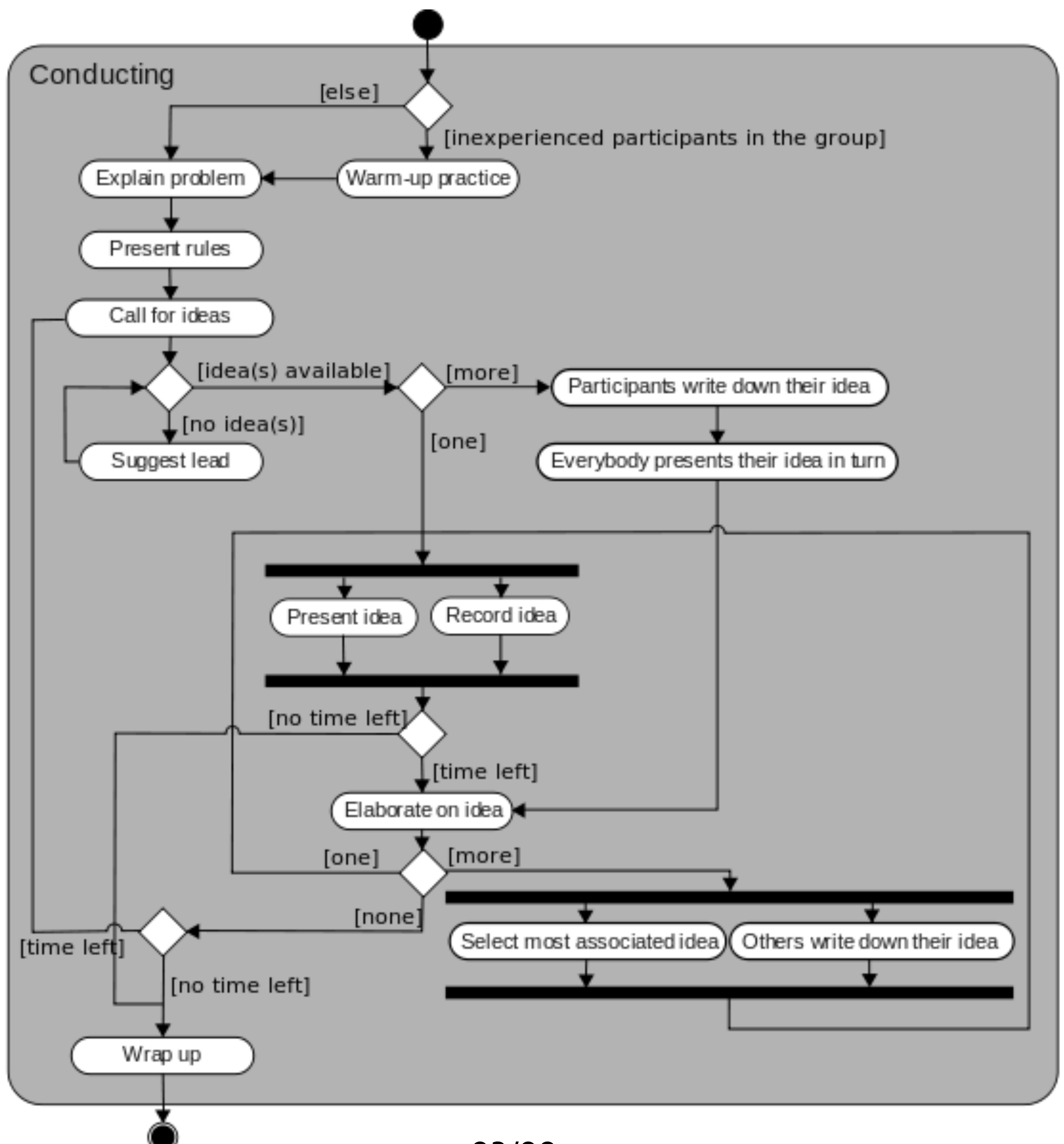
Package Diagram



Activity Diagram

- Describes the business and operational step-by-step workflows of components in a system
  - shows the overall flow of control
  - Rounded rectangles represent actions
  - Diamonds represent decisions
  - Bars represent the start (split) or end (join) of concurrent activities
  - A black circle represents the start (initial state) of the workflow
  - An encircled black circle represents the end (final state)

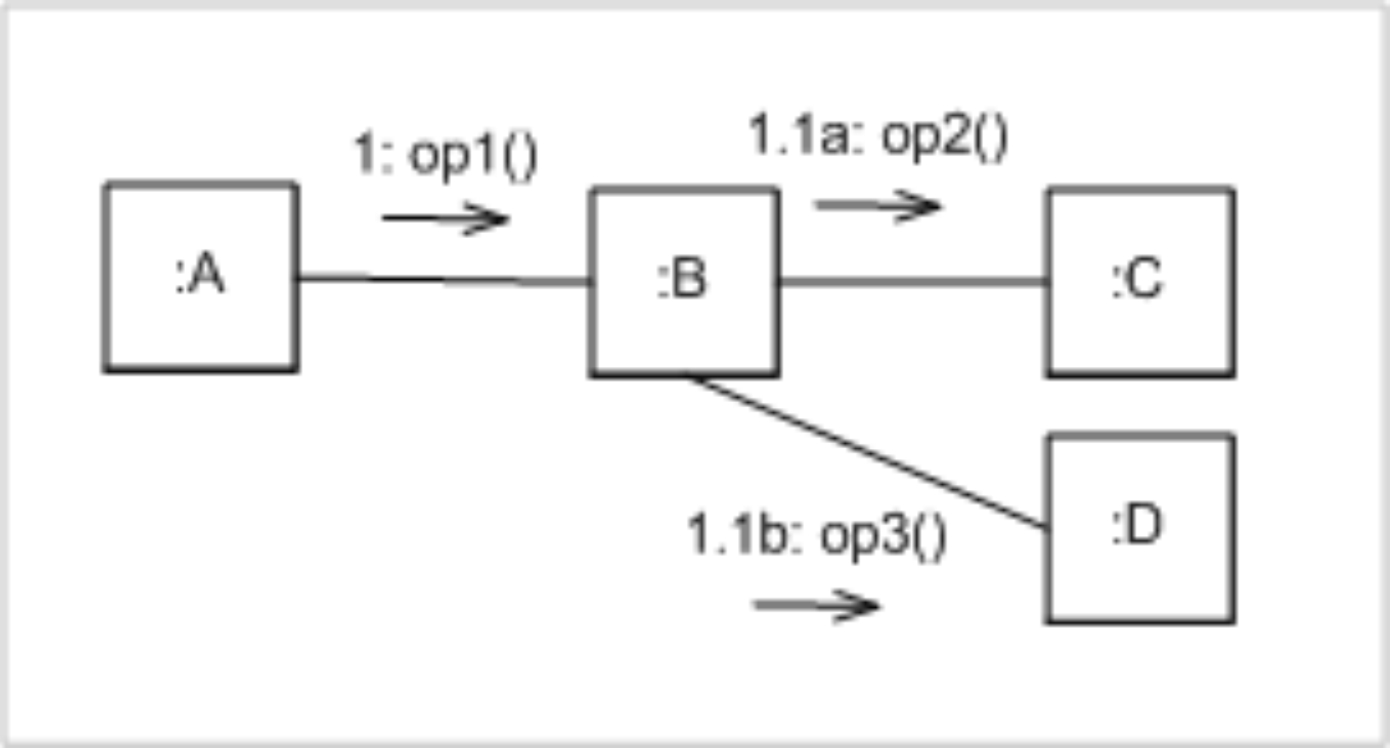
Activity Diagram



Communication Diagram

- Shows the interactions between objects or parts in terms of sequenced messages.
- Messages are labeled with a chronological number and placed near the link the message is sent over. Reading a communication diagram involves starting at message 1.0, and following the messages from object to object.

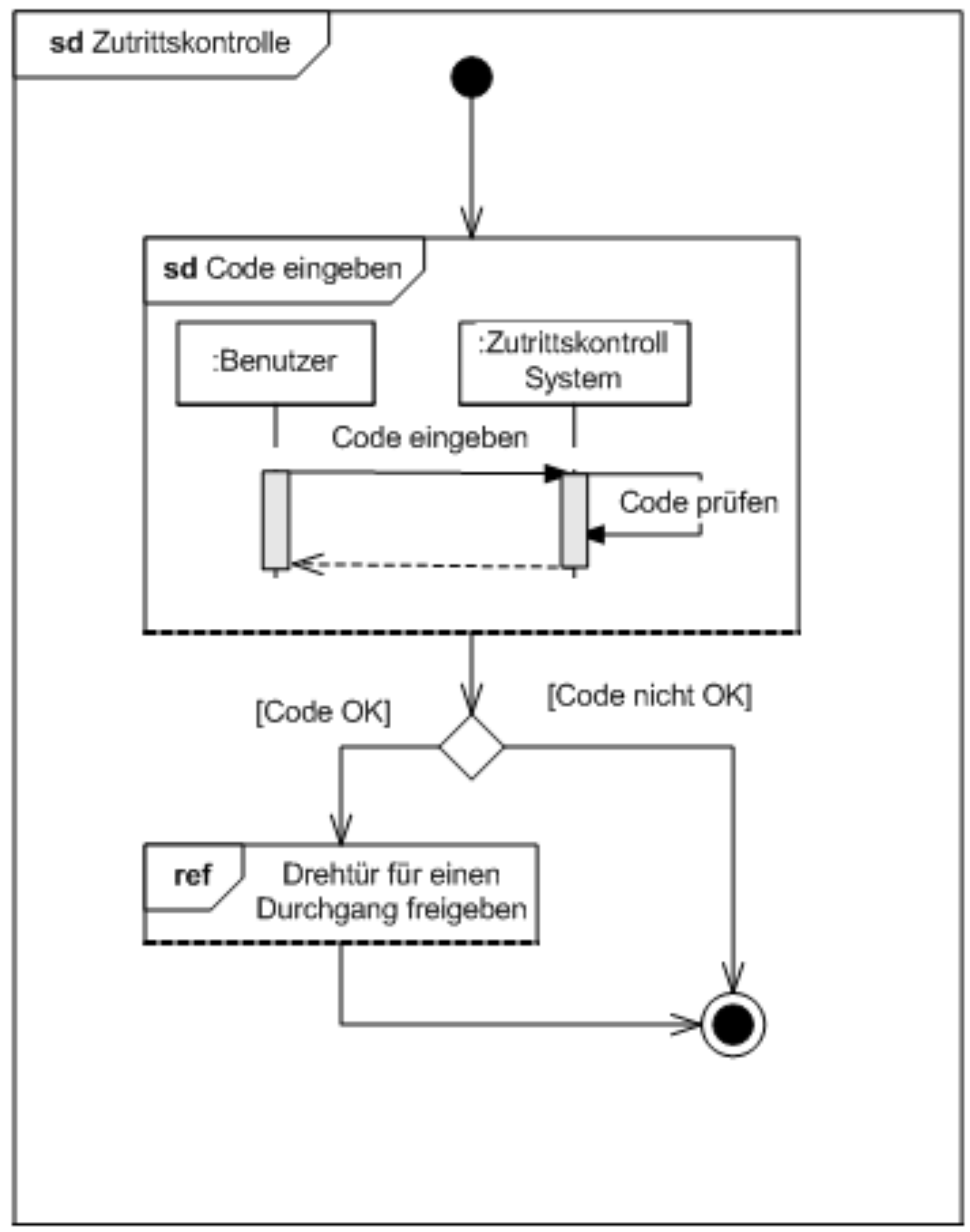
Communication Diagram



Interaction Overview Diagram

- Provides an overview in which the nodes represent communication diagrams.
- Individual activity is pictured as a frame, which can contain interaction or sequence diagrams.
- Constructed with building blocks of other diagrams
  - sequence
  - communication
  - interaction overview
  - timing diagram

Interaction Overview Diagram



Timing Diagram

- A specific type of interaction diagram where the focus is on timing constraints.
- Axes are reversed so that the time is increased from left to right and the lifelines are shown in separate compartments arranged vertically.