Software Architecture and Engineering

Analysis

Peter Müller

Chair of Programming Methodology

The slides in this section are partly based on the lecture "Software Engineering I" by Prof. Bernd Brügge, TU München

Spring Semester 2012

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

3. Analysis

3.1 Modeling

3.2 Object Modeling

3.3 From Use Cases to Objects

3.4 Dynamic Modeling

3.5 Examples

3.6 Analysis Model Validation

Requirements Engineering: Overview

Client

Users

Requirements Elicitation

Requirements specification

Analysis

Analysis Model

Designers

Design

Participation

Used for communication

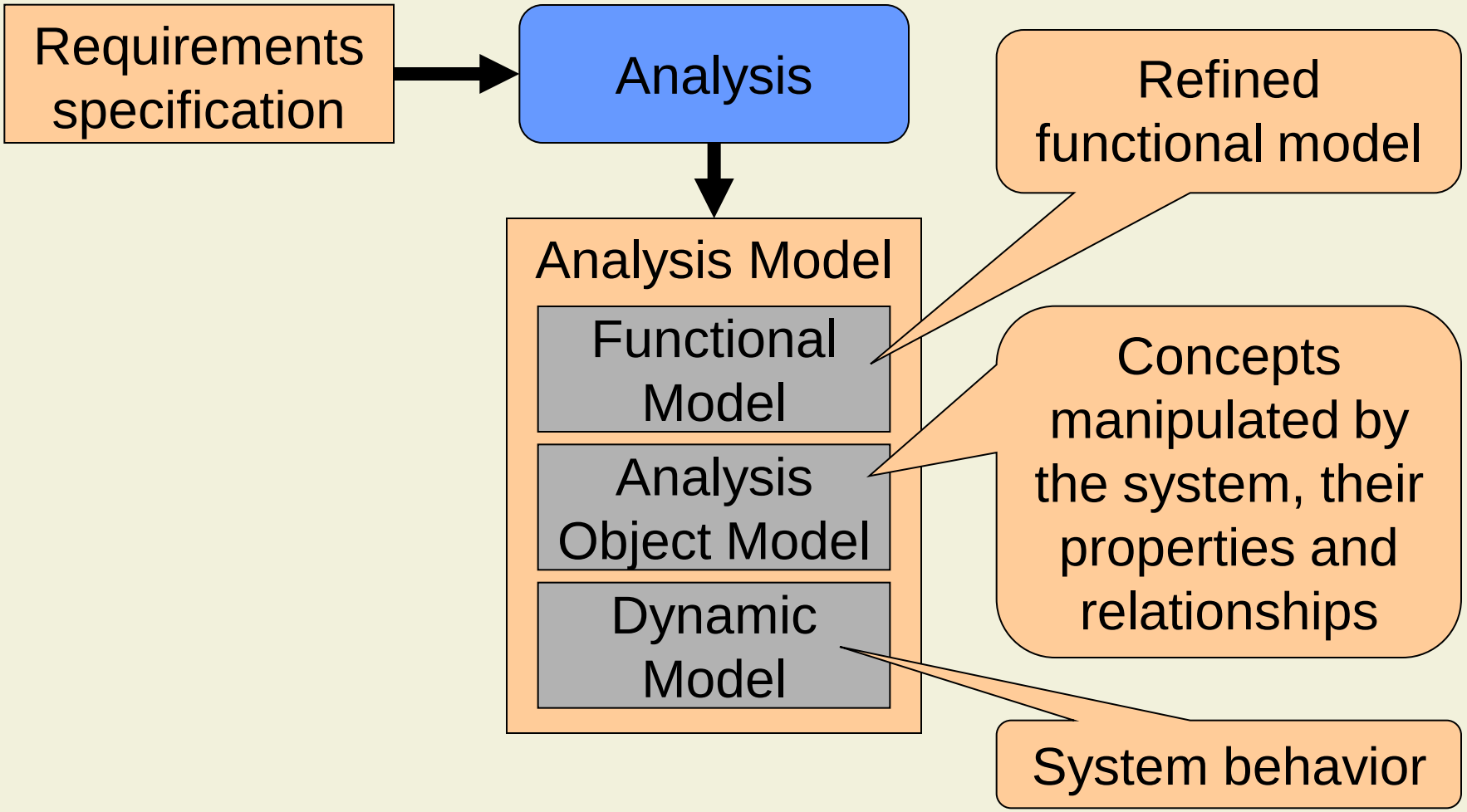
Requirements Elicitation vs. Analysis

- Requirements specification and analysis model **represent the same information**

- **Requirements Elicitation**
  - Definition of the system in terms **understood by the customer**
  - Requirements specification uses **natural language**
  - Communication with **clients and users**

- **Analysis**
  - Technical specification of the system in terms **understood by the developer**
  - The analysis model uses a **formal or semi-formal notation**
  - Communication among **developers**

Analysis Model

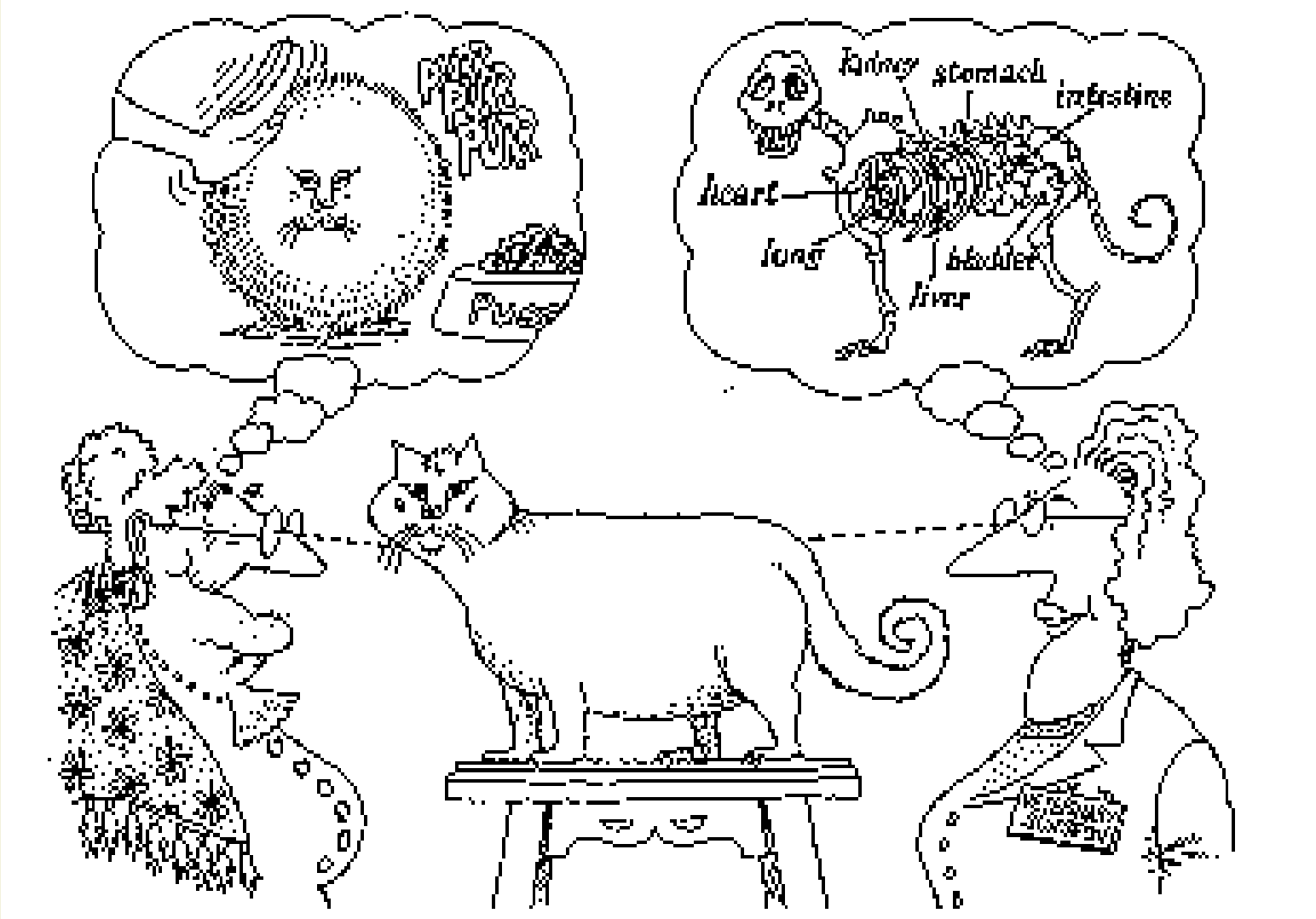


3. Analysis

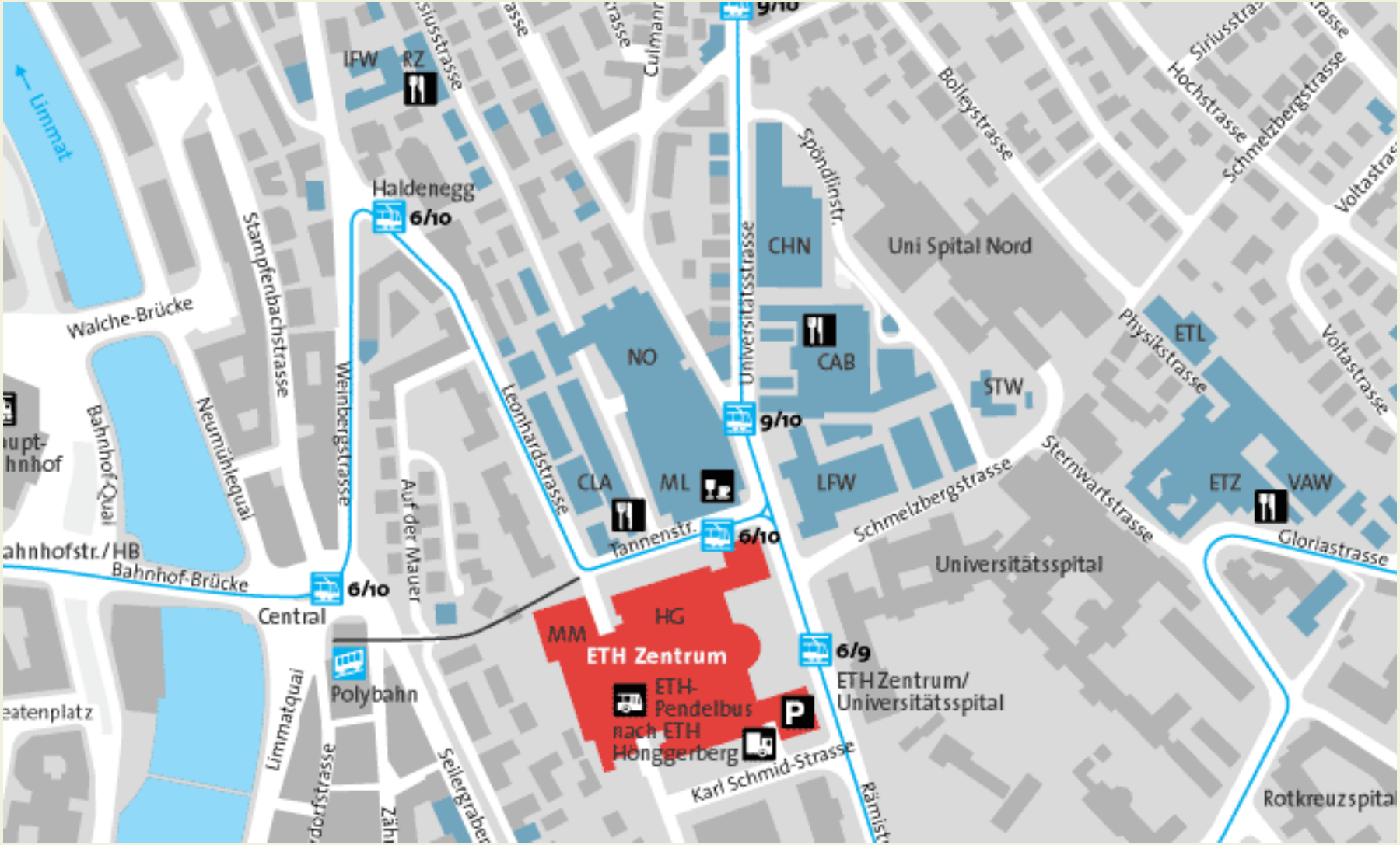
What is Modeling?

- Building an **abstraction of reality**
  - Abstractions from things, people, and processes
  - Relationships between these abstractions
- Abstractions are **simplifications**
  - They ignore irrelevant details
  - They represent only the relevant details
  - What is relevant or irrelevant depends on the purpose of the model
- Draw complicated conclusions in the reality with simple steps in the model

Example 1: Cat

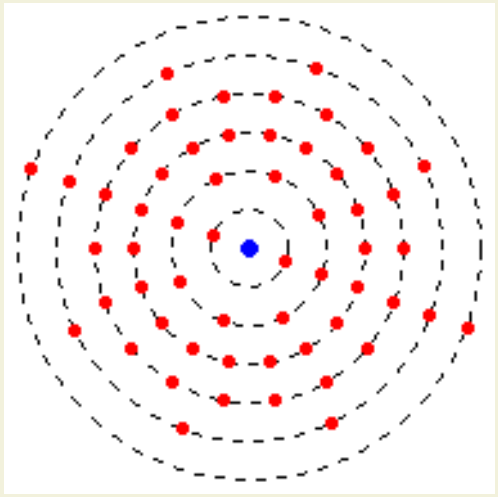


Example 2: Street Map

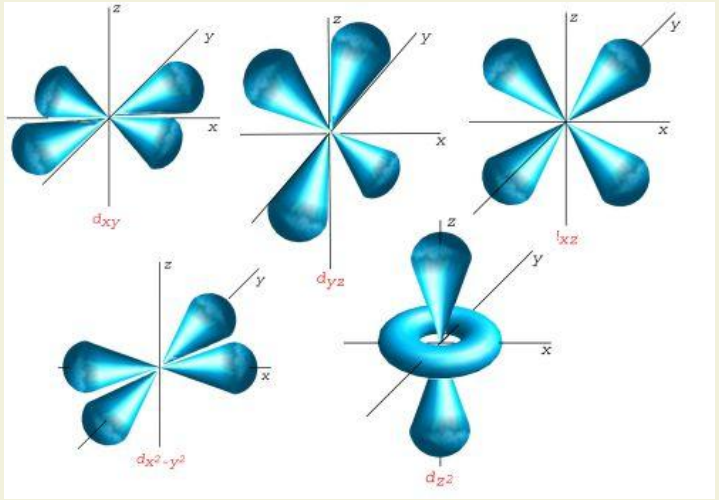


Example 3: Atom Models in Physics

- Bohr model
  - Nucleus surrounded by electrons in orbit
  - Explains, e.g., spectra



- Quantum physics
  - Position of electrons described by probability distribution
  - Takes into account Heisenberg's uncertainty principle

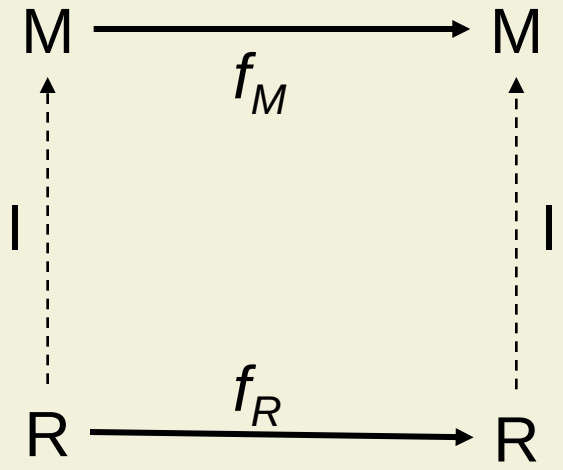


Why Model Software?

- Software is getting increasingly more complex
  - Windows Vista: ~50 millions source lines of code
  - Linux kernel: ~10 millions source lines of code
  - Fedora 9 Linux: ~200 millions source lines of code
  - A single programmer **cannot manage** this amount of code in its entirety
- Code is **not easily understandable** by developers who did not write it
- We need **simpler representations** for complex systems
- Modeling is a means for **dealing with complexity**

What is a Good Model?

- Intuitively: A model is good if relationships, which are valid in reality R, are also valid in model M
- Definition Interpretation I: R $\rightarrow$ M



I: Mapping of real things in reality R to abstractions in model M

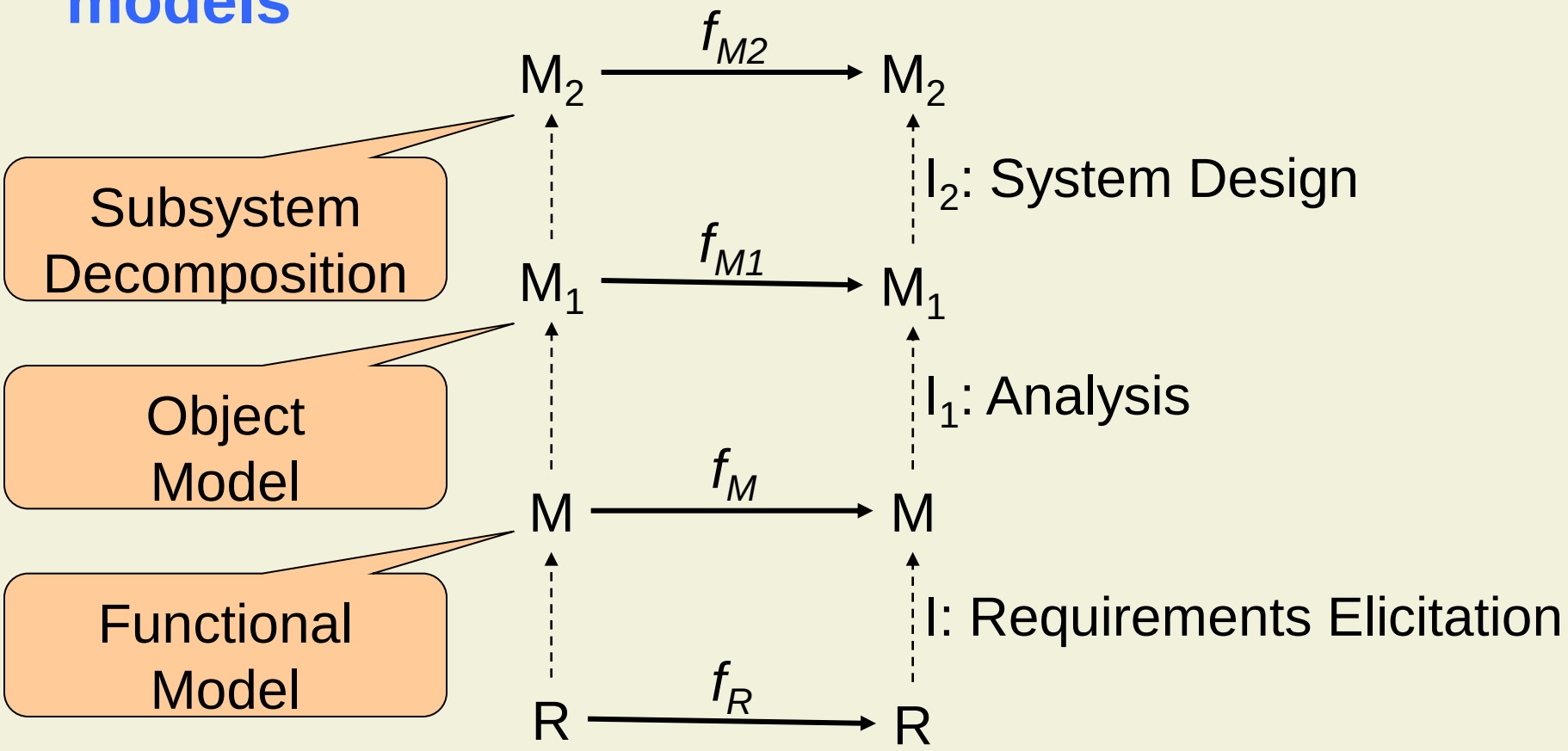
f_M: Relationship between abstractions in M

f_R: Relationship between real things in R

- In a good model this diagram is commutative

Models of Models of Models …

- Software development is **transformation of models**



Modeling the Real World

Problem domain

Abstraction

Model view of problem

Modeling Method

Representation of model

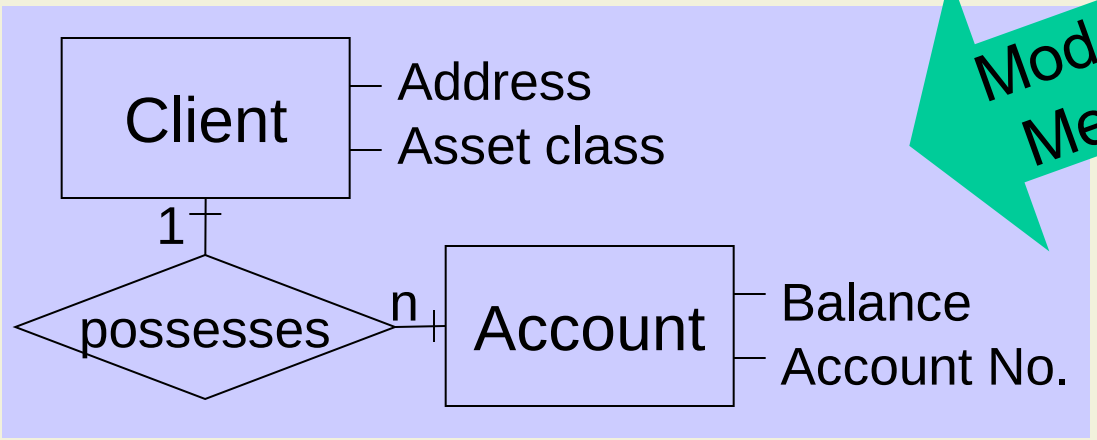
Modeling Example: Data Modeling

Bank client

Abstraction

Tuple of
- Address
- Asset class
- At least one account

Modeling Method



ER-Diagram

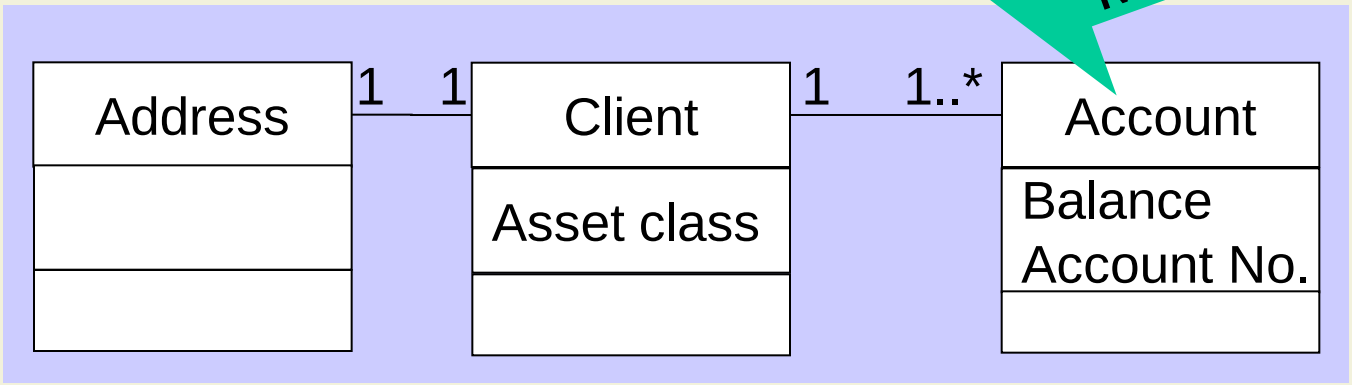
Modeling Example: Object Modeling

Bank client

Abstraction

Object with
- Data
- Operations

Modeling Method



UML Class Diagram

3. Analysis

The Unified Modeling Language UML

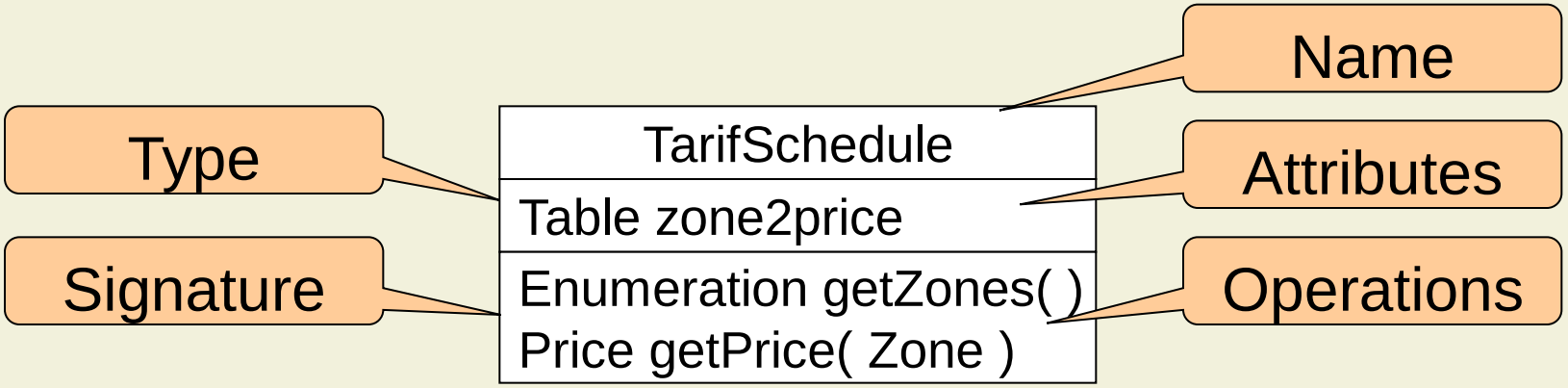
- UML is a modeling language
  - Using **text** and **graphical notation**
  - For documenting specification, **analysis**, **design**, and **implementation**

- Importance
  - Recommended OMG (Object Management Group) standard notation
  - **De facto standard** in industrial software development

UML Notations

- Use case diagrams – requirements of a system
- Class diagrams – structure of a system
- Interaction diagrams – message passing
  - Sequence diagrams
  - Collaboration diagrams
- State and activity diagrams – actions of an object
- Implementation diagrams
  - Component model – dependencies between code
  - Deployment model – structure of the runtime system
- Object constraint language (OCL)

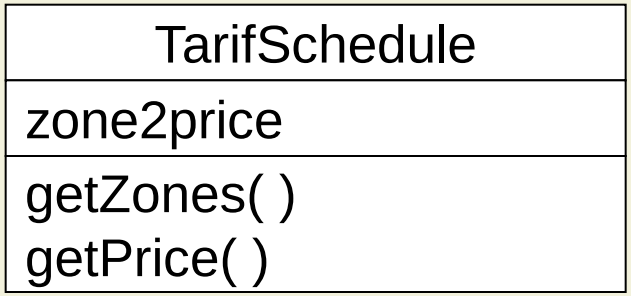
Classes

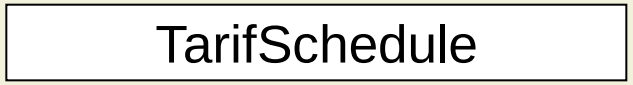


- A class encapsulates **state** (attributes) and **behavior** (operations)
  - Each attribute has a type
  - Each operation has a signature
- The class name is the only mandatory information

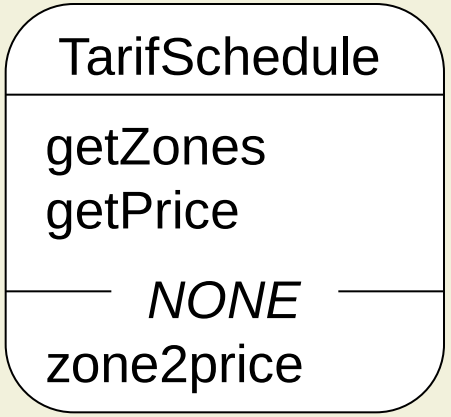
More on Classes

- Valid UML class diagrams

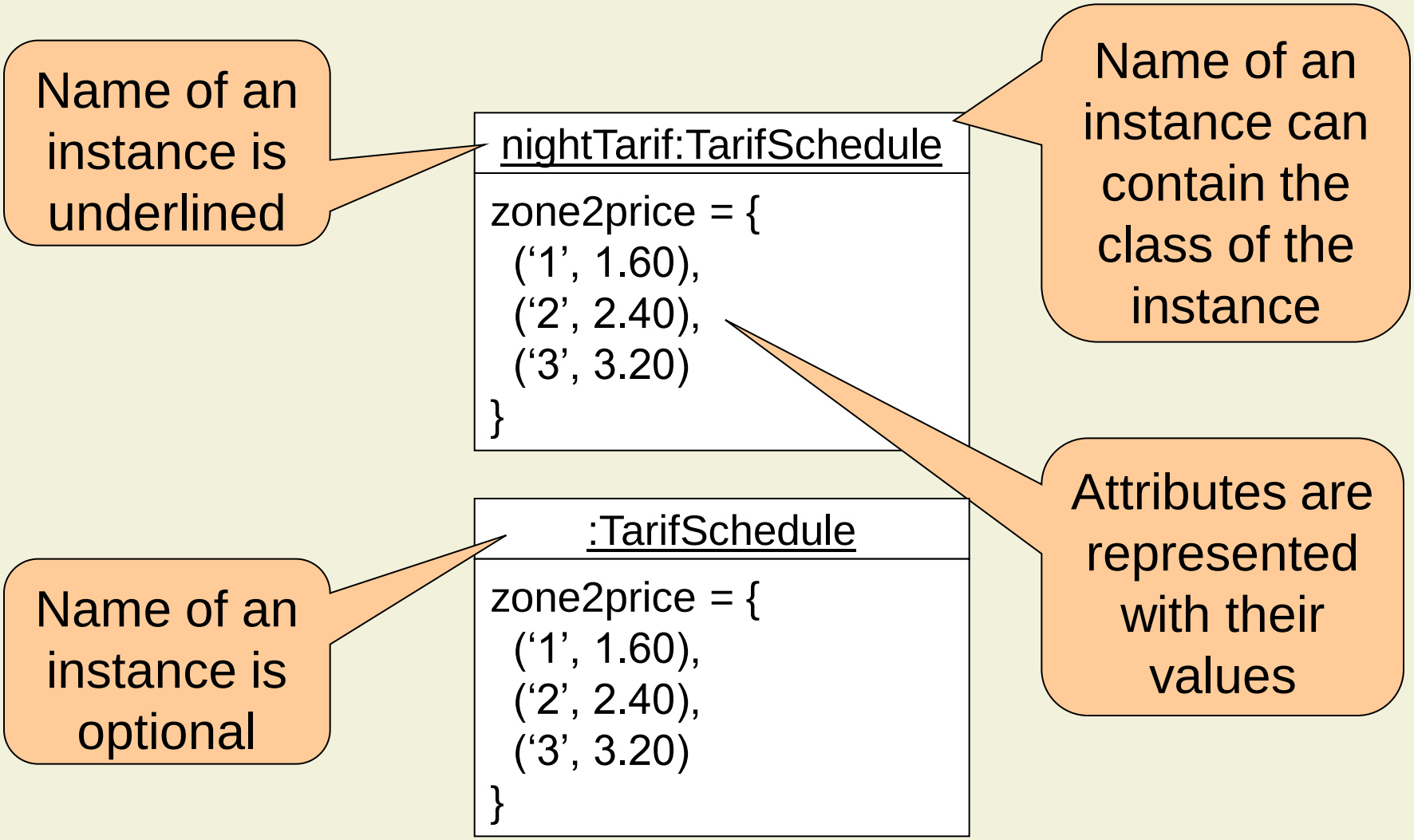




- Corresponding BON diagram
  - No distinction between attributes and operations (uniform access principle)

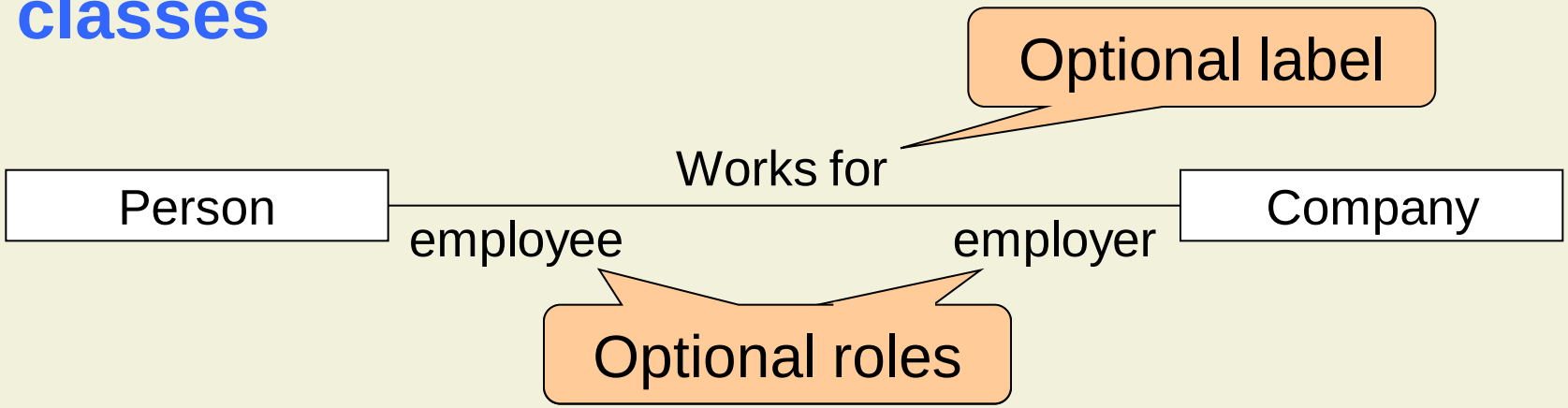


Instances (Objects)



Associations

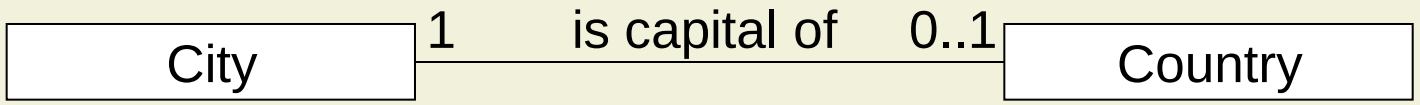
- A link represents a connection between two objects
  - Ability of an object to **send a message** to another object
  - Object A has an **attribute** whose value is B
  - Object A **creates** object B
  - Object A **receives a message** with object B as argument
- Associations denote **relationships between classes**



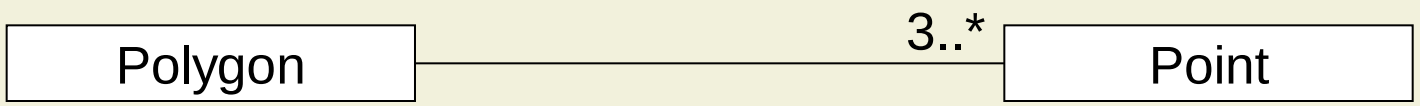
Multiplicity of Associations

- The multiplicity of an association end denotes how many objects the source object can reference
  - Exact number: 1, 2, etc. (1 is the default)
  - Arbitrary number: * (zero or more)
  - Range: 1..3, 1..*

- 1-to-(at most) 1 association

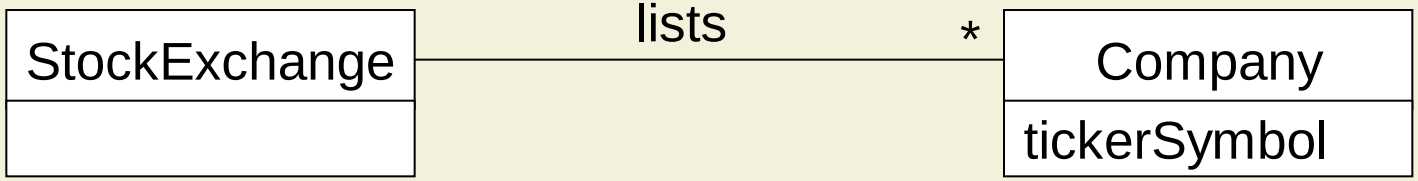


- 1-to-many association

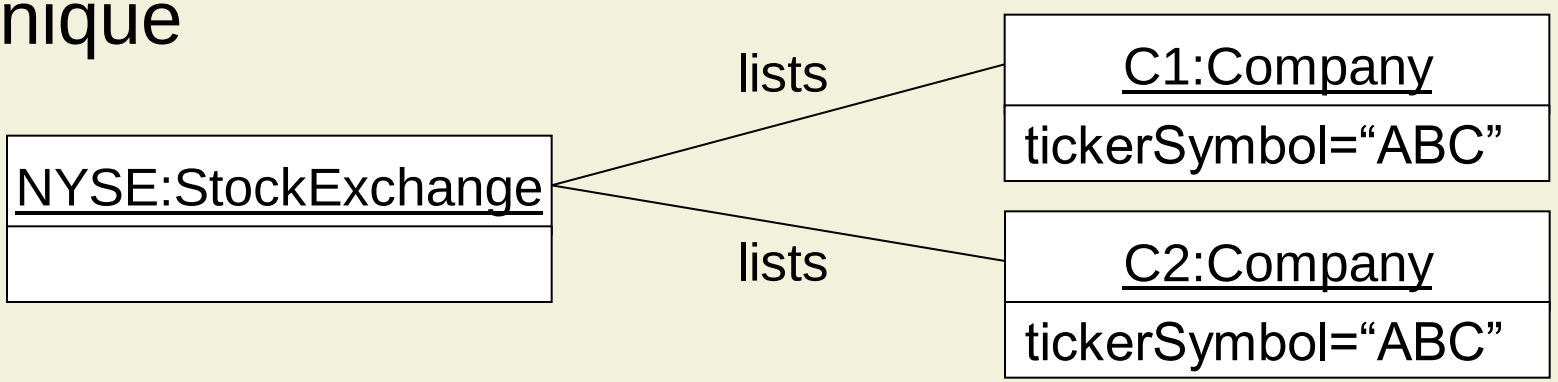


Association: Example

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

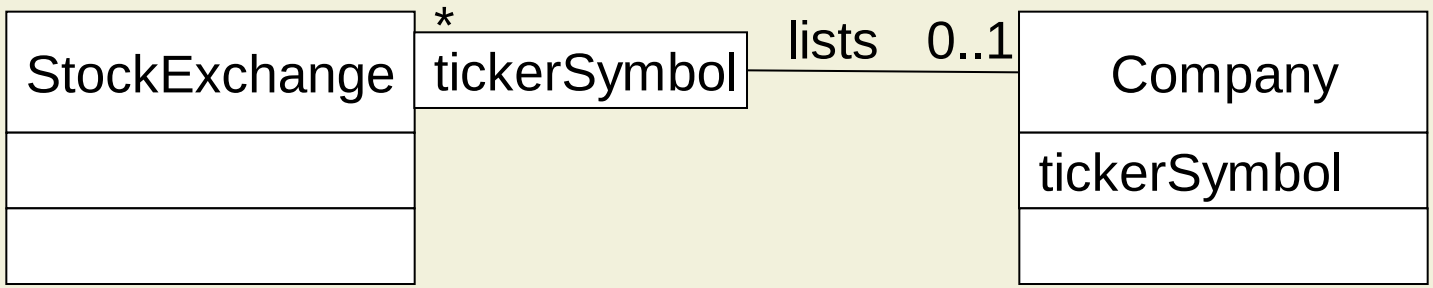


- Diagram does not express that ticker symbols are unique



Qualified Associations

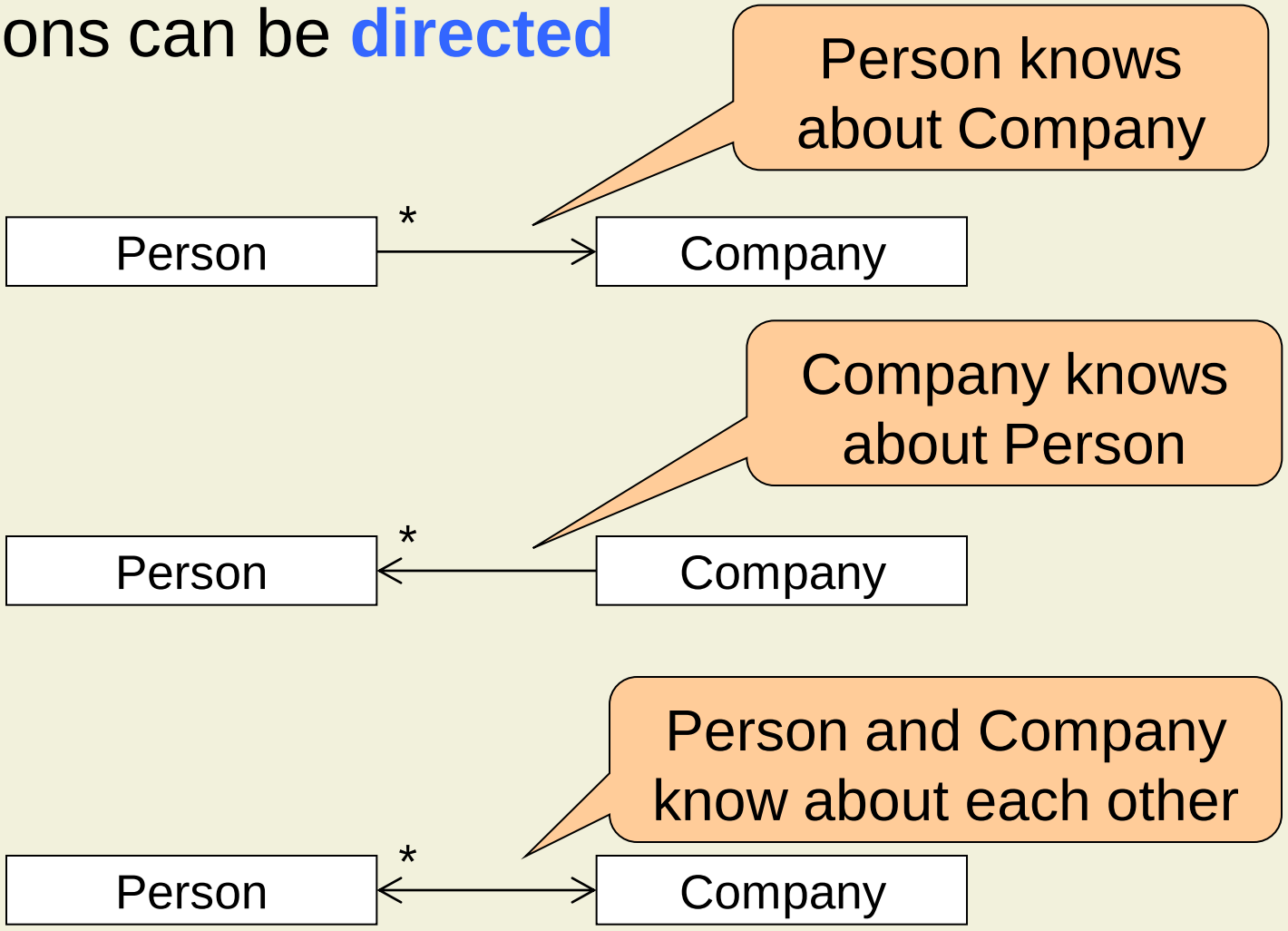
- For each ticker symbol, a stock exchange lists exactly one company



- Qualifiers reduce the multiplicity of associations

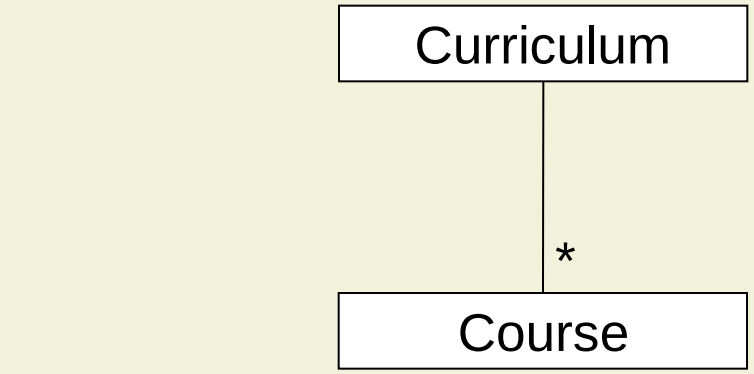
Navigability

- Associations can be **directed**

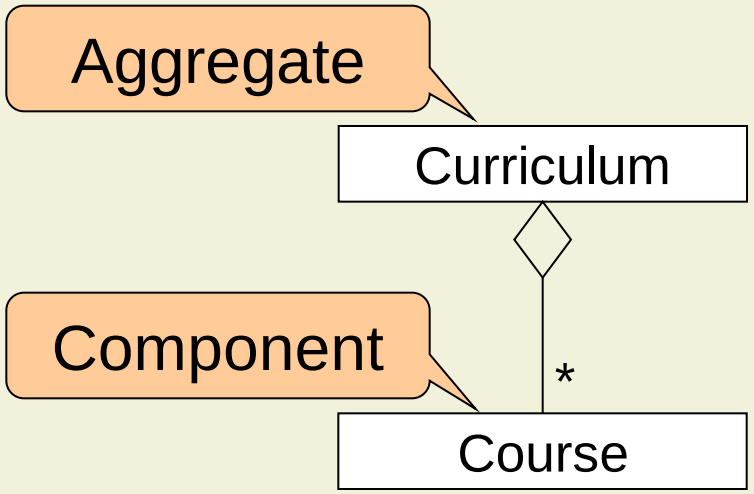


Aggregation

- Aggregation expresses a hierarchical **part-of** (“has-a”) **relationship**
  - Special form of association
  - Objects can simultaneously be part of several aggregates

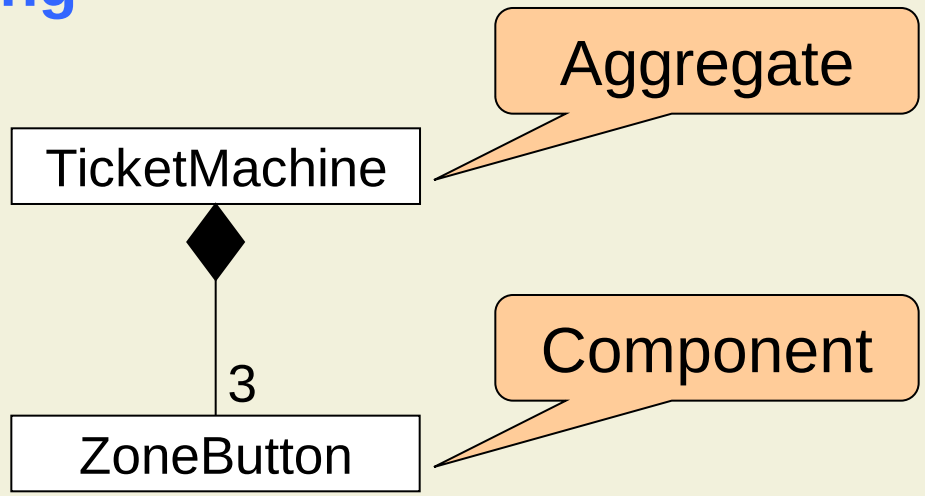


- Used for documentation purposes only
  - No formal information
  - Use with care!



Composition

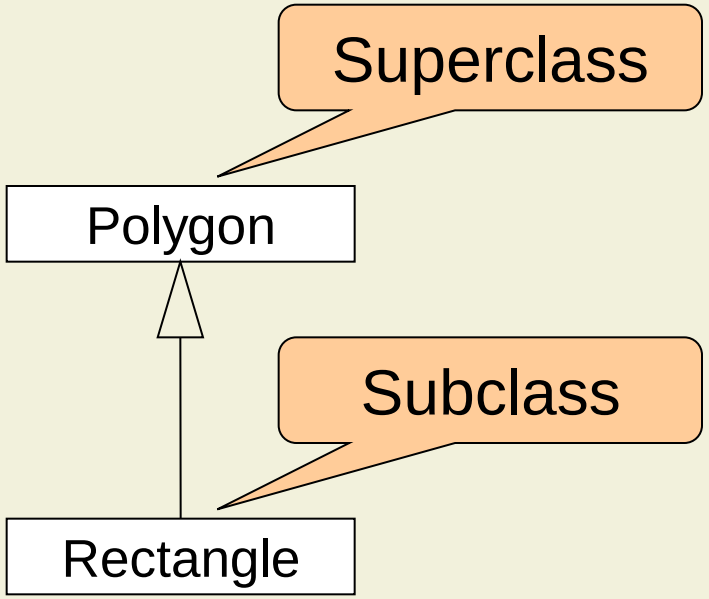
- Composition expresses a strong aggregation
  - **No sharing**



- Aggregation and composition can be documented like other associations
  - Multiplicity, label, roles

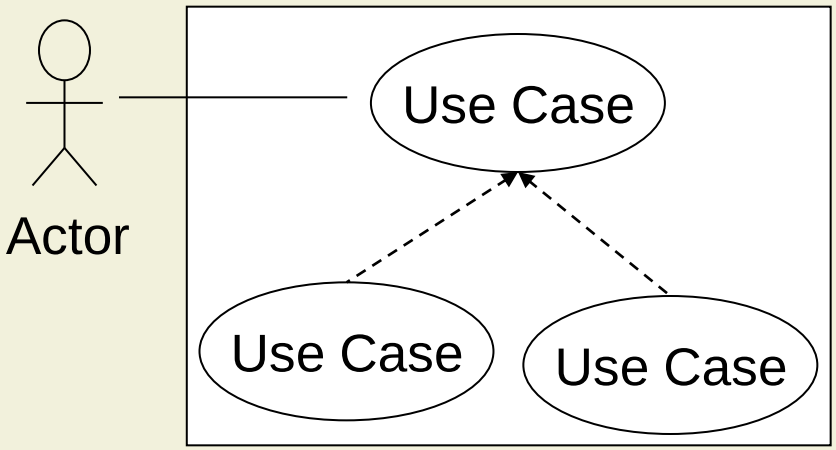
Generalization and Specialization

- Generalization expresses a **kind-of** (“is-a”) **relationship**
- Generalization is implemented by **inheritance**
  - The child classes inherit the attributes and operations of the parent class
- Generalization simplifies the model by **eliminating redundancy**

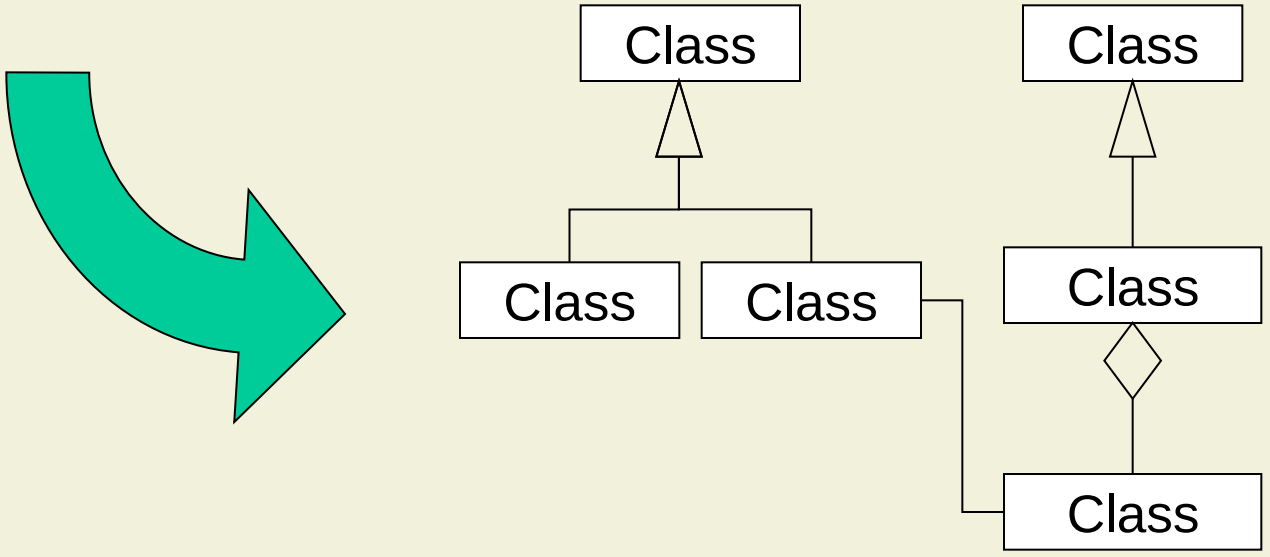


3. Analysis

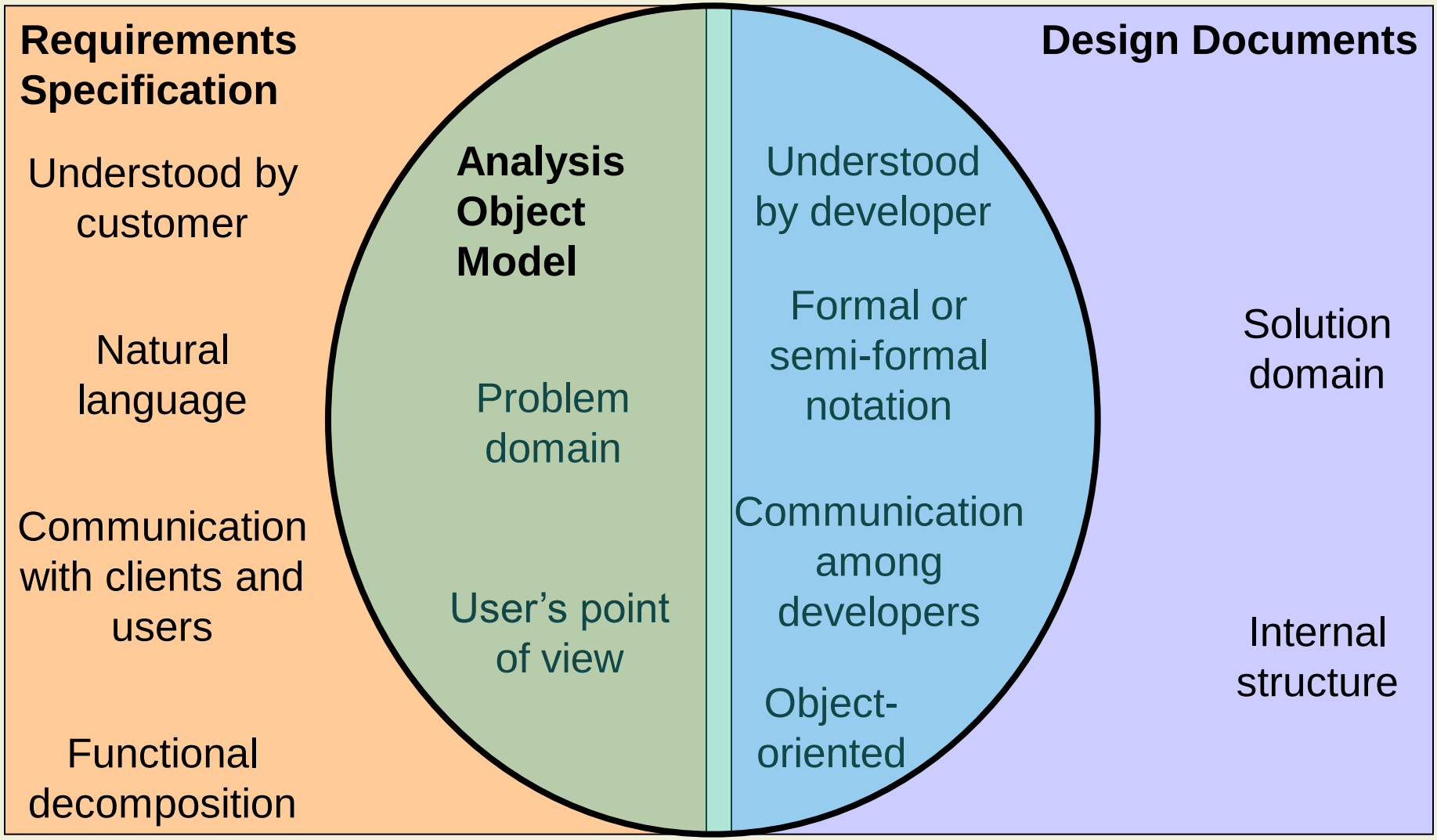
Analysis Object Model: Motivation



The analysis object model bridges the gap between use cases and an object-oriented design

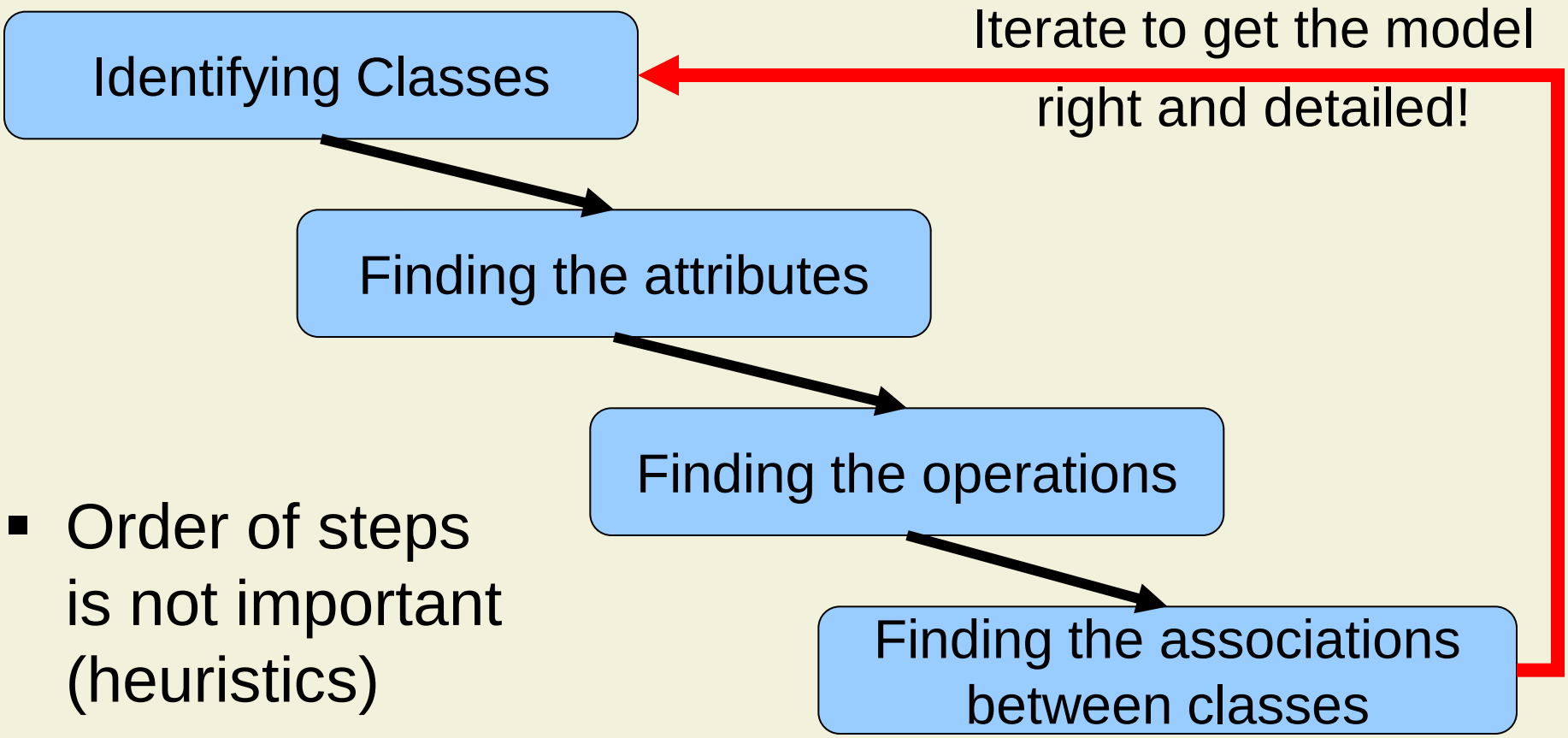


Analysis Object Model: Properties



Activities During Object Modeling

- Main goal: **Find important abstractions**



- Order of steps is not important (heuristics)

Approaches to Class Identification

Application domain approach

- Ask application domain expert to identify relevant abstractions

Syntactic approach

- Extract participating objects from flow of events in use cases
- Use noun-verb analysis to identify components of the object model

Design patterns approach

- Use reusable design patterns

Component-based approach

- Identify existing solution classes

Noun-Verb Analysis (Abbott's Textual Analysis)

- Do a **textual analysis** of problem statement
- Take the flow of events and find participating objects in use cases and scenarios
  - **Nouns** are good candidates for **classes**
  - **Verbs** are good candidates for **operations**

- Works well for short, structured texts
  - **Problem statement**
  - Flow of events in **use cases**

Textual Analysis Example: Problem Statement

The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loans only. All other books can be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of the staff may borrow up to 12 items at one time. Only members of the staff may borrow journals.

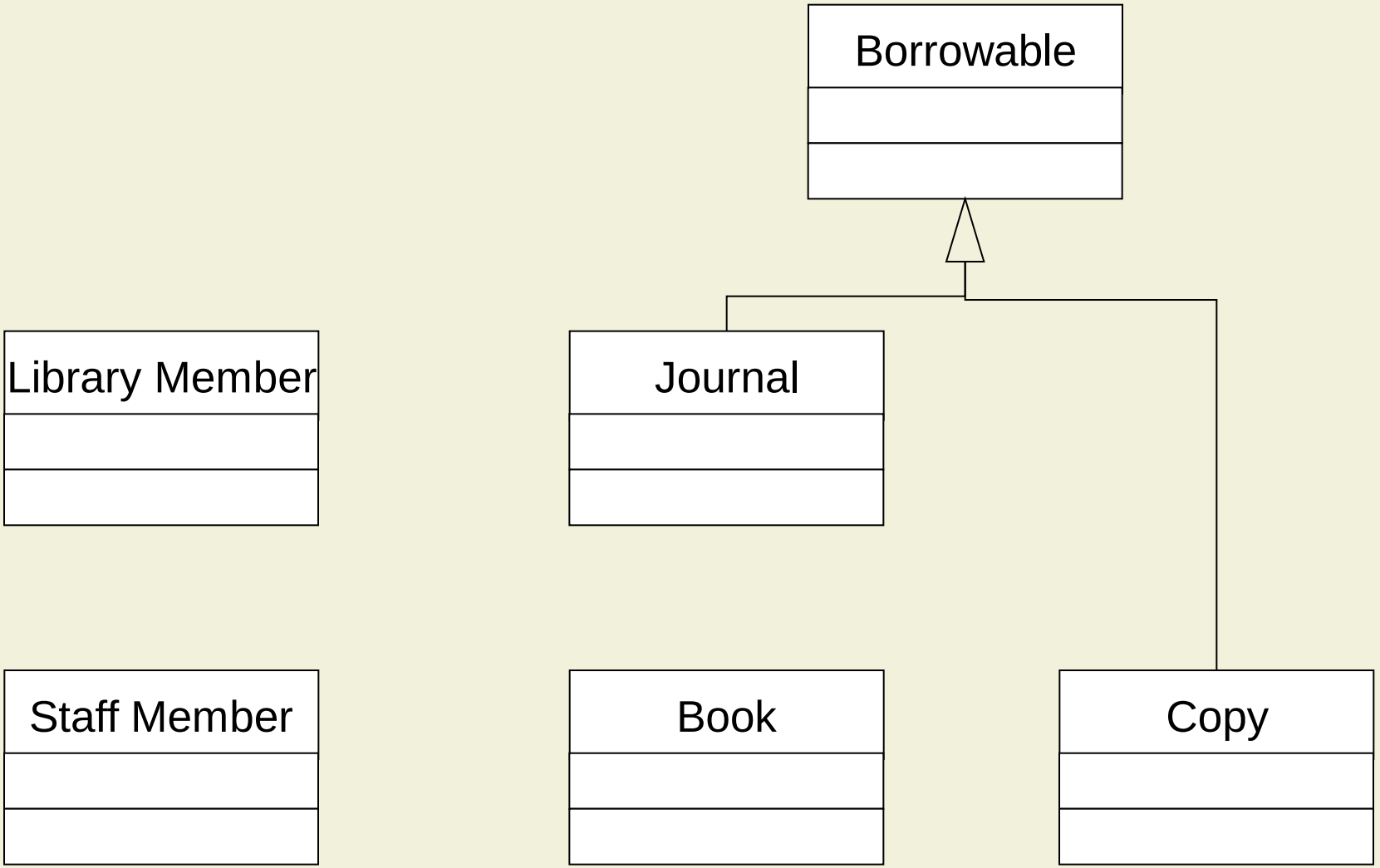
Textual Analysis Example: Nouns

The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loans only. All other books can be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of the staff may borrow up to 12 items at one time. Only members of the staff may borrow journals.

Textual Analysis Example: Selecting Classes

- Library: inside or outside the system?
- Book, journal, copy: candidates for classes
- Loan: property or event
- Library member: candidate for a class
- Week: unit of measurement
- Items: used to refer to books and journals
- Time: event
- Staff members: candidate for a class

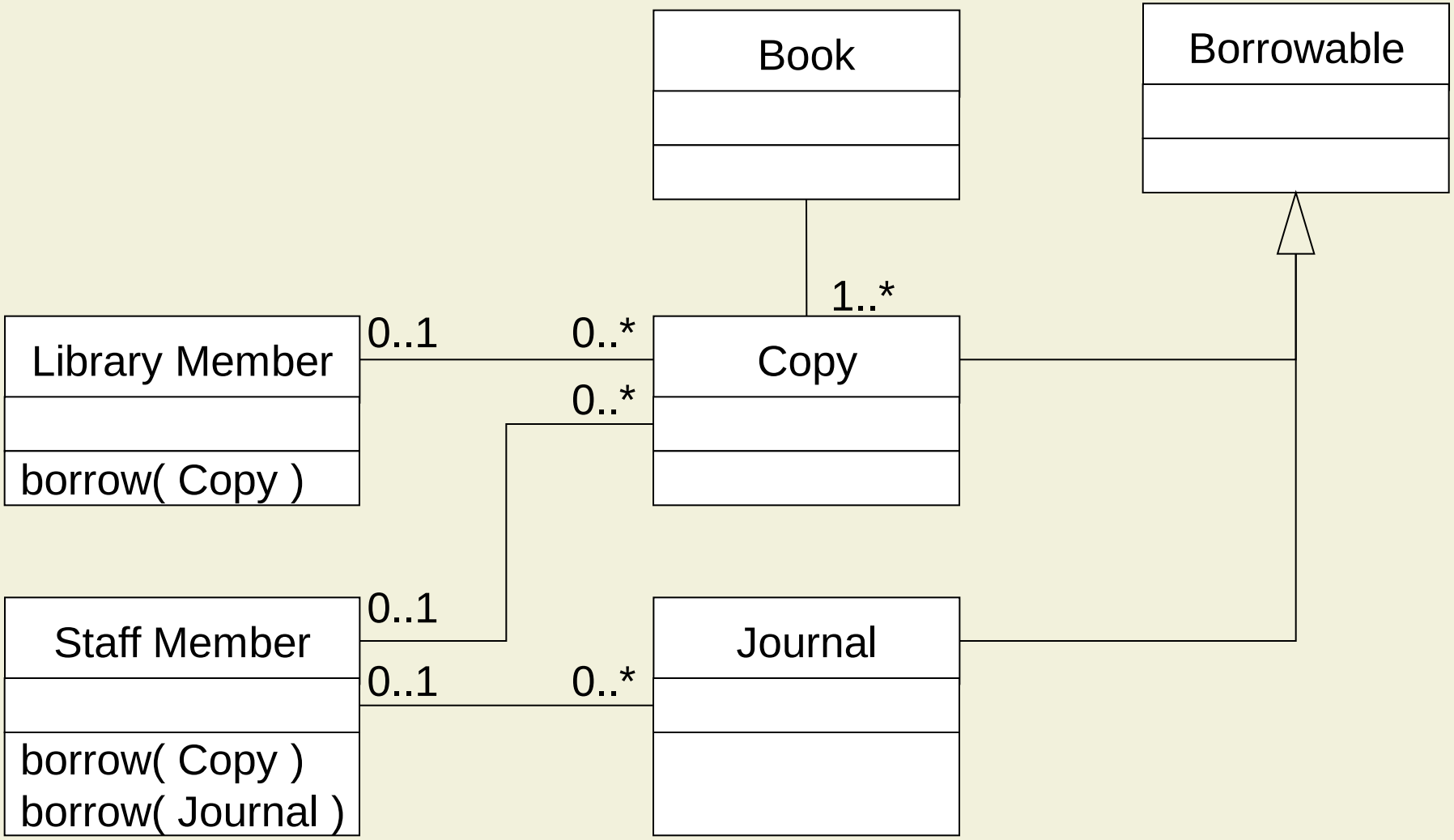
Textual Analysis Example: Class Diagram



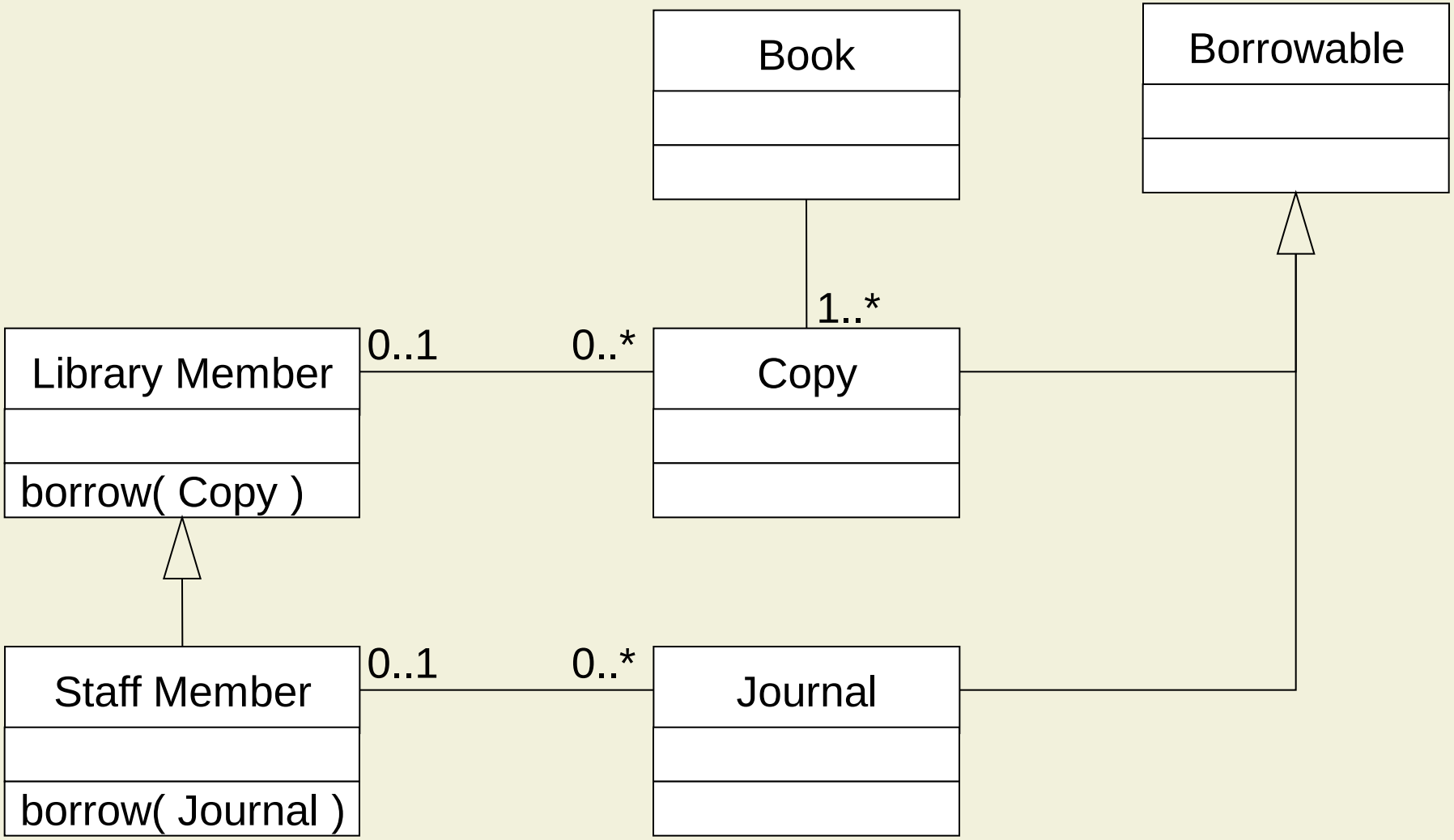
Textual Analysis Example: Verbs

The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loans only. All other books can be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of the staff may borrow up to 12 items at one time. Only members of the staff may borrow journals.

Textual Analysis Example: Class Diagram



Textual Analysis Example: Iteration



Textual Analysis Example: Remainder

The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loans only. All other books can be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of the staff may borrow up to 12 items at one time. Only members of the staff may borrow journals.

Attribute in Borrowable

Precondition for borrow

Mapping Speech to Object Models

Part of speech	Model component	Example
Proper noun	Object	Jim Smith
Improper noun	Class	Toy, doll
Doing verb	Method	Buy, recommend
being verb	Inheritance	is-a (kind-of)
having verb	Aggregation	has a
modal verb	Constraint	must be
adjective	Attribute	3 years old
transitive verb	Method	enter
intransitive verb	Method (event)	sleep

Problems of Noun-Verb Analysis

- Natural language is **imprecise**
  - Identify and **standardize** terms
  - Rephrase and **clarify** requirements specification

- Many more nouns than relevant classes
  - **Eliminate synonyms**; use same word for the same thing
  - Many nouns correspond to attributes

Different Kinds of Objects

Entity Objects

- Represent the persistent information tracked by the system
- Application domain objects, "business objects"

Boundary Objects

- Represent the interaction between the user and the system

Control Objects

- Represent the control tasks performed by the system

- Having three kinds of objects makes models more resilient to change
  - Interface of system changes more likely than control
  - Control of system changes more likely than application domain

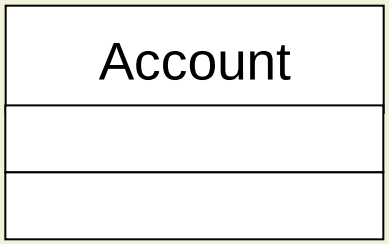
Identifying Entity Objects

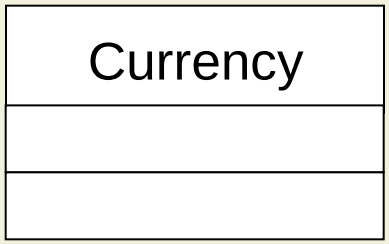
- For each use case, **participating objects** are
  - Identified (e.g., by noun-verb analysis)
  - Named by application domain terms
  - Described and collated in a glossary

- Results in the **initial analysis model**

Heuristics for Identifying Entity Objects

- **Terms** the developers or users must **clarify** to understand the use case (e.g., account)
- **Recurring nouns** in the use case (e.g., card)
- **Real-world entities** that the system must track (e.g., cash dispenser)
- **Real-world processes** that the system must track
- **Data sources** or **sinks** (e.g., host)





Cross Checks

- Use cases and initial analysis models can be **improved by cross-checking**

- Which use case creates this object?
- Which actors can access this information?
- Which use cases modify and destroy this object?
- Which actors can initiate these use cases?
- Is this object needed? (Is there at least one use case that depends on this information?)

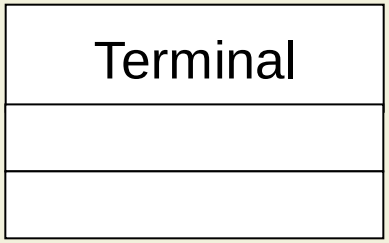
Identifying Boundary Objects

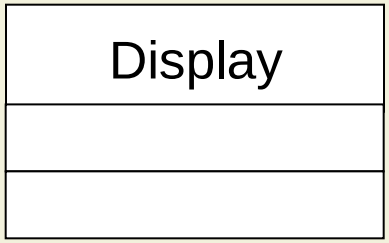
- Boundary objects **collect information** from actor
- Boundary objects **translate information** into format for entity and control objects
- Boundary objects do not model details and visual aspects (e.g., menu item, scrollbar)

- Each actor interacts with at least one boundary object

Heuristics for Identifying Boundary Objects

- **User interface controls** to initiate the use case (e.g., bank card)
- **Forms** to enter data (e.g., option screen)
- **Messages** the system uses to respond (e.g., termination message)



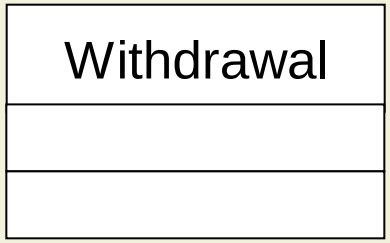


Identifying Control Objects

- Control objects **coordinate** boundary and entity objects
- Control objects usually **do not have** a concrete **counterpart in the real world**
- Control objects are typically **created at beginning** of use case and exist to its end
- Control objects collect information from boundary objects and dispatch it to entity objects
- Examples
  - Sequencing of forms, undo and history queues
  - Dispatching information in distributed systems

Heuristics for Identifying Control Objects

- Identify **one control object per use case**
- Identify **one control object per actor** in the use case
- **Life span of a control object** should **cover the extent of a use case** or user session



Stereotypes and Conventions

- UML provides stereotypes to attach **extra classifications**

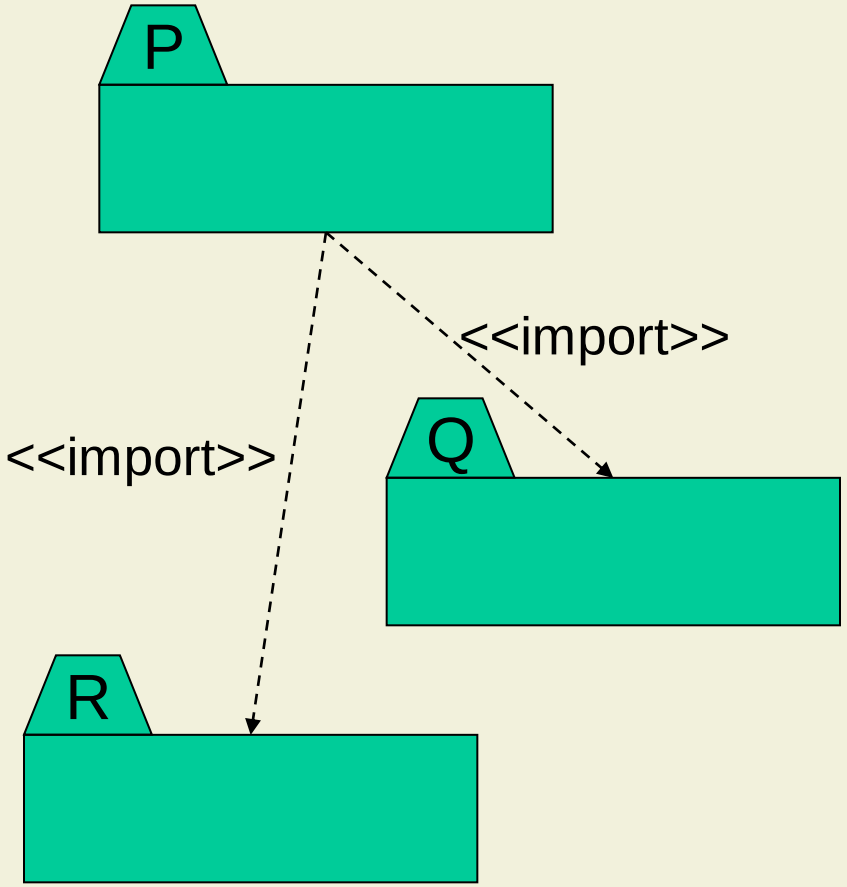
<<Entity>> Account	<<Boundary>> Terminal	<<Control>> Withdrawal

- Naming conventions help to distinguish kinds of objects

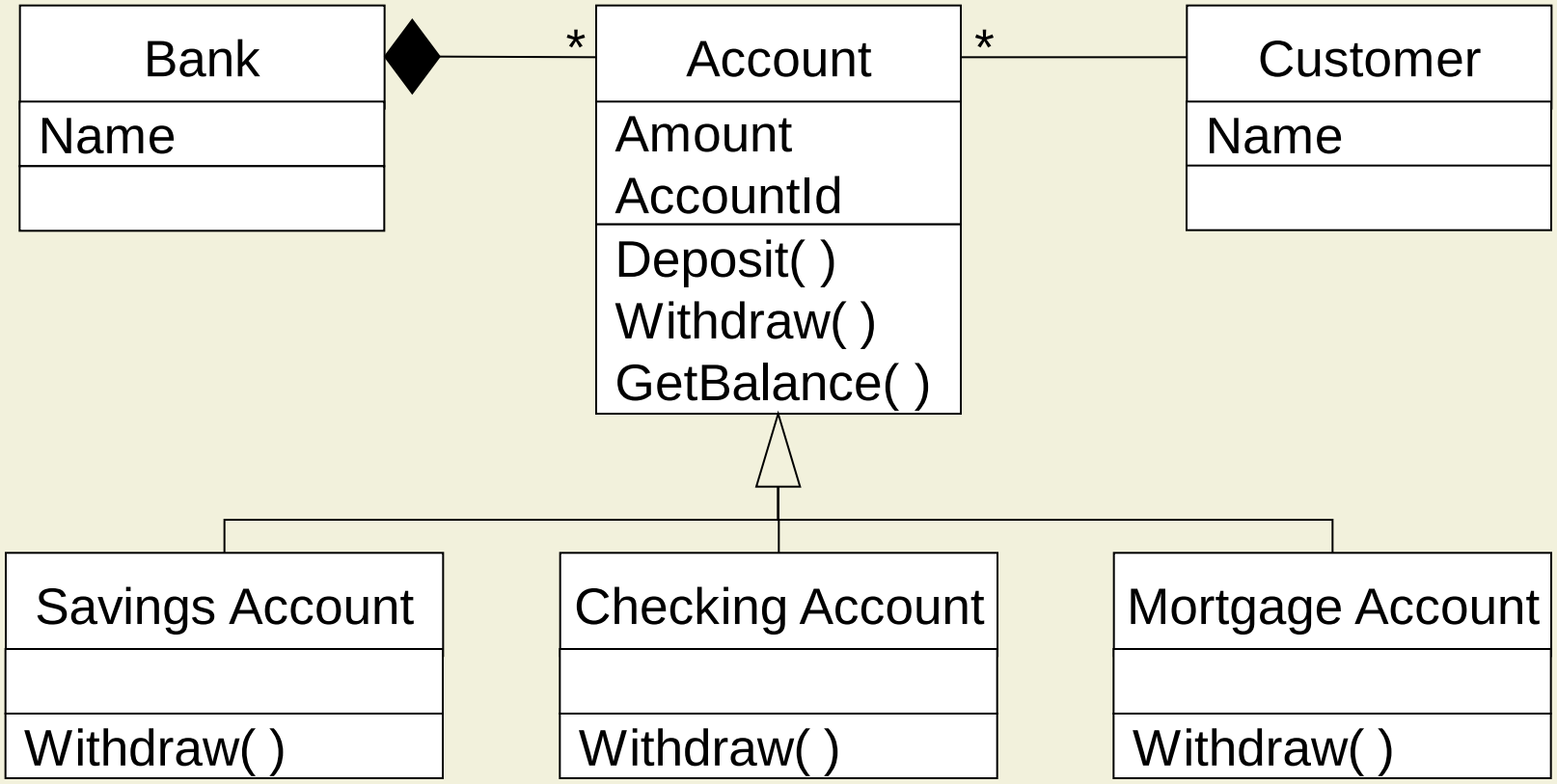
<<Entity>> Account	<<Boundary>> Terminal_Boundary	<<Control>> Withdrawal_Control

UML Packages

- A package is a UML mechanism for **organizing elements** into groups
  - Usually not an application domain concept
  - Increase readability of UML models
- **Decompose** complex systems into subsystems
  - Each subsystem is modeled as a package

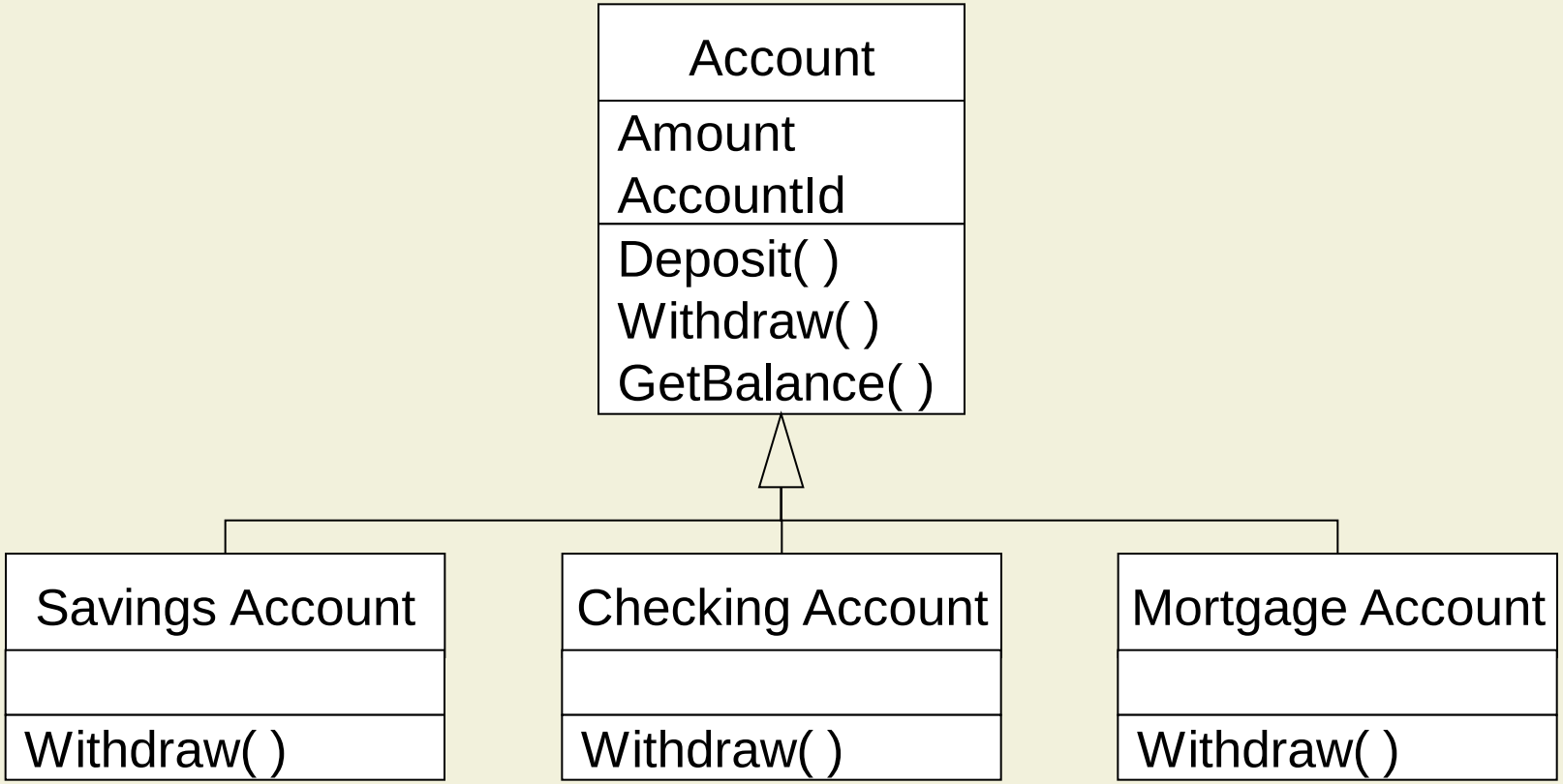


Avoid Ravioli Models



- Don’t put too many classes into the same package: 7 ± 2 (or even 5 ± 2)

Put Taxonomies on a Separate Diagram



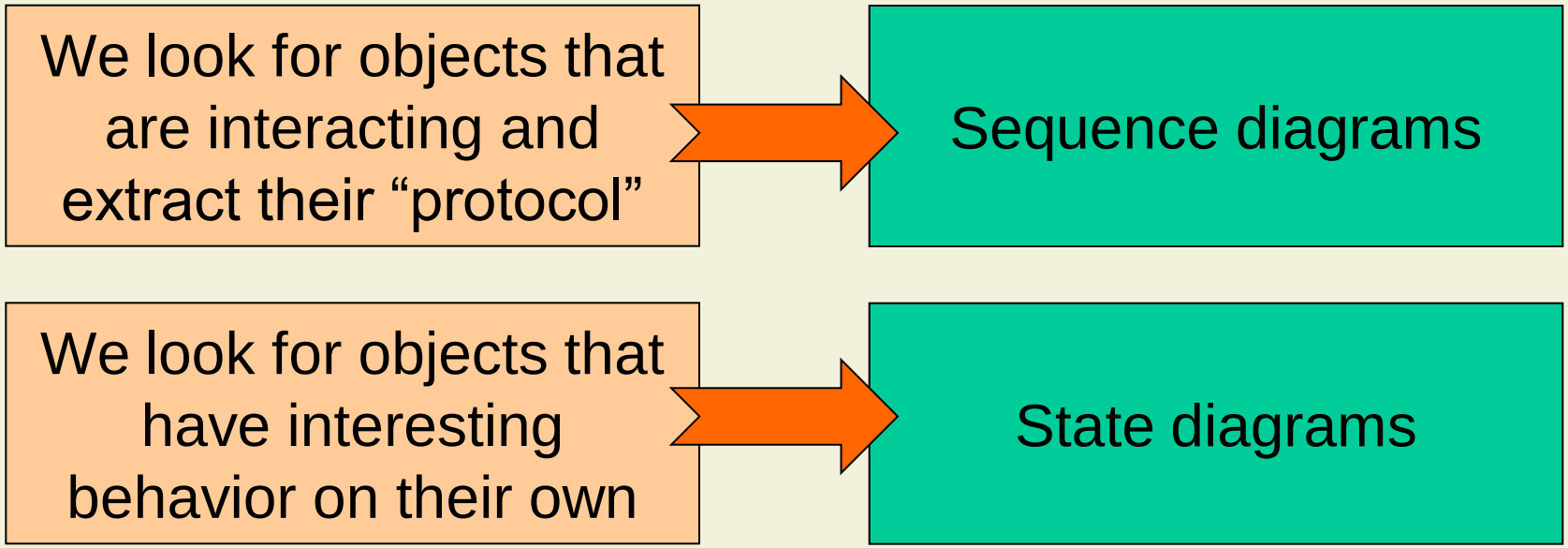
Summary: Ways to Find Objects

- **Syntactical investigation** with Abbott's technique
  - In the problem statement
  - In the flow of events of use cases
- Use of various knowledge sources
  - **Application knowledge**: Interviews of users and experts to determine the abstractions of the application domain
  - **Design knowledge**: Reusable abstractions in the solution domain
  - **General world knowledge**: Use your empirical knowledge and intuition

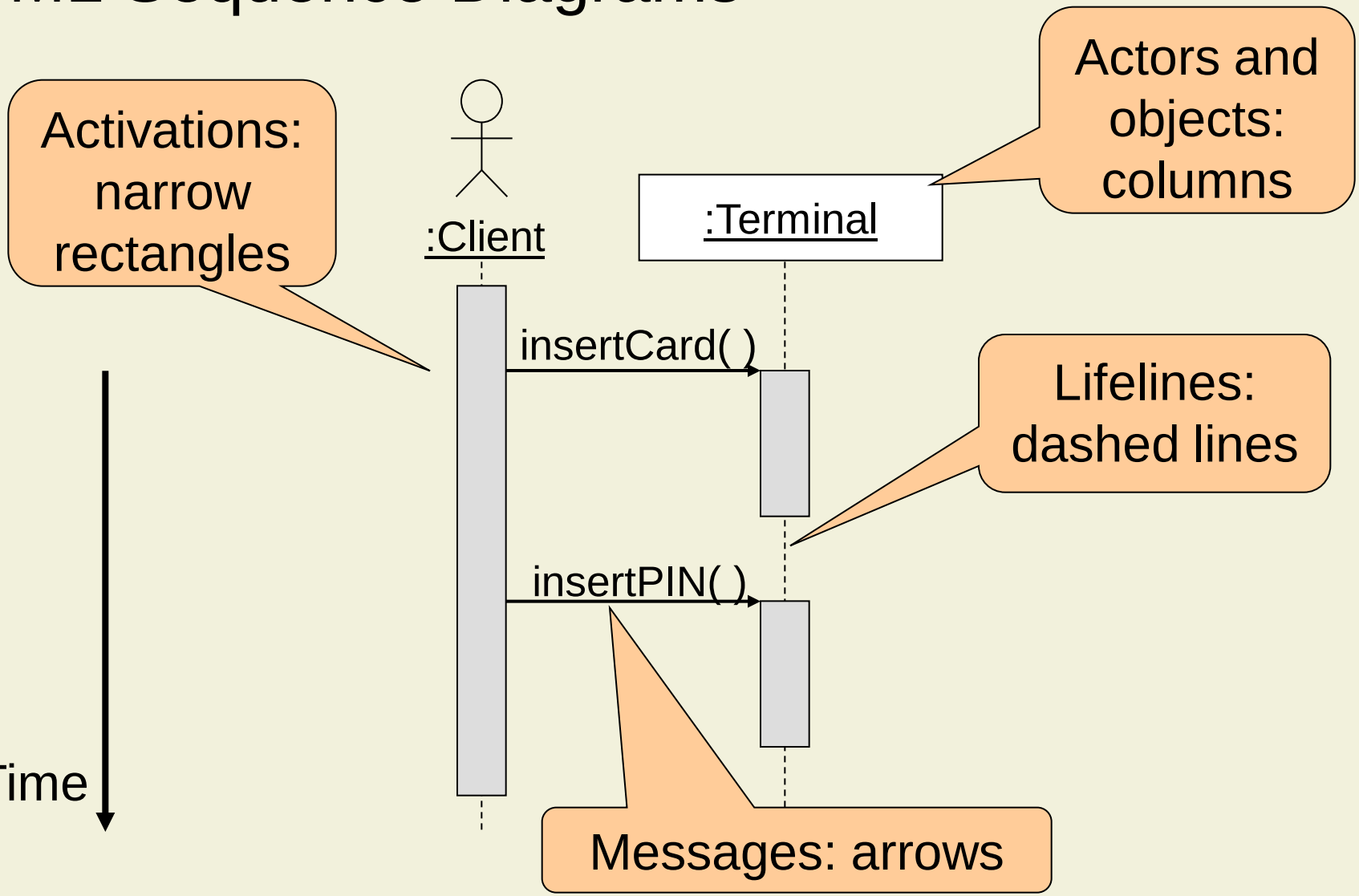
3. Analysis

Overview

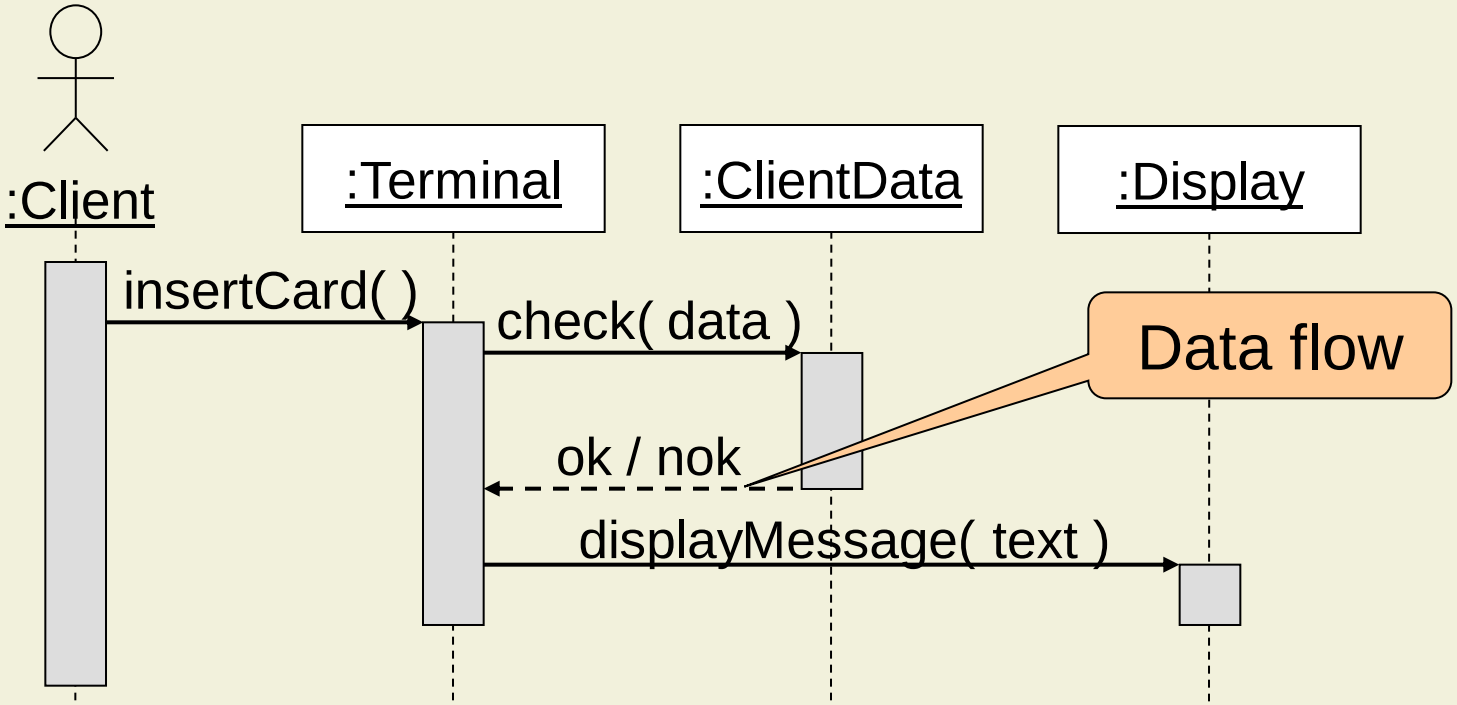
- **Object model** describes **structure** of system
- **Dynamic model** describes **behavior**
- Purpose: Detect and supply operations (methods) for the object model



UML Sequence Diagrams

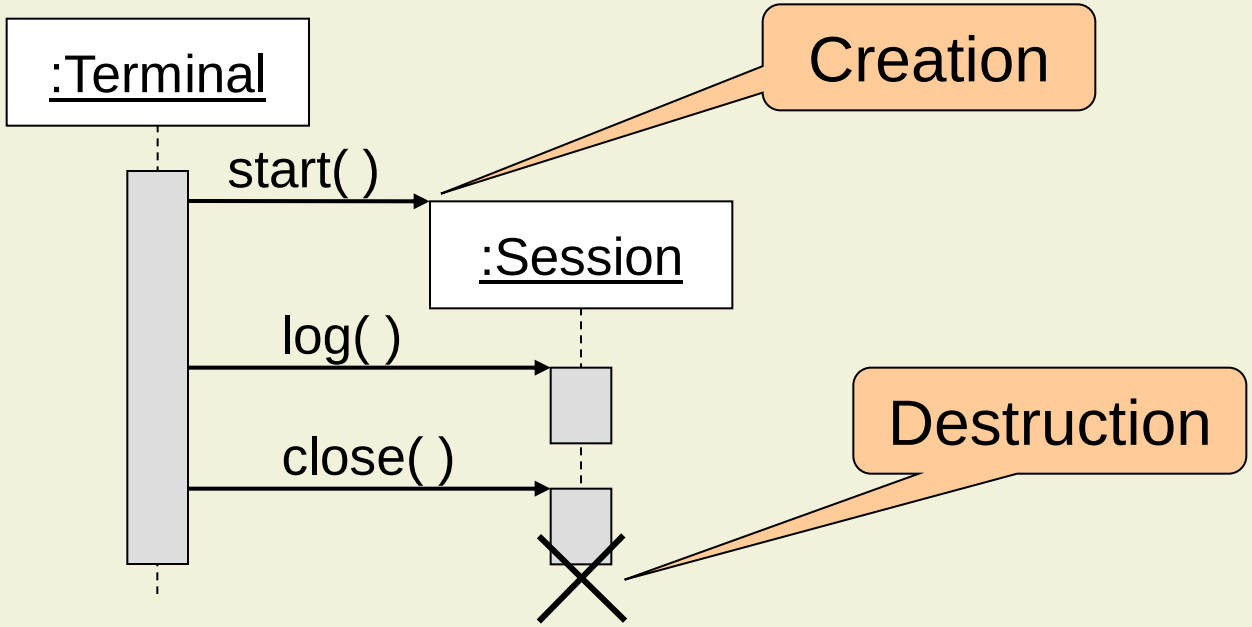


Nested Messages



- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

Creation and Destruction



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

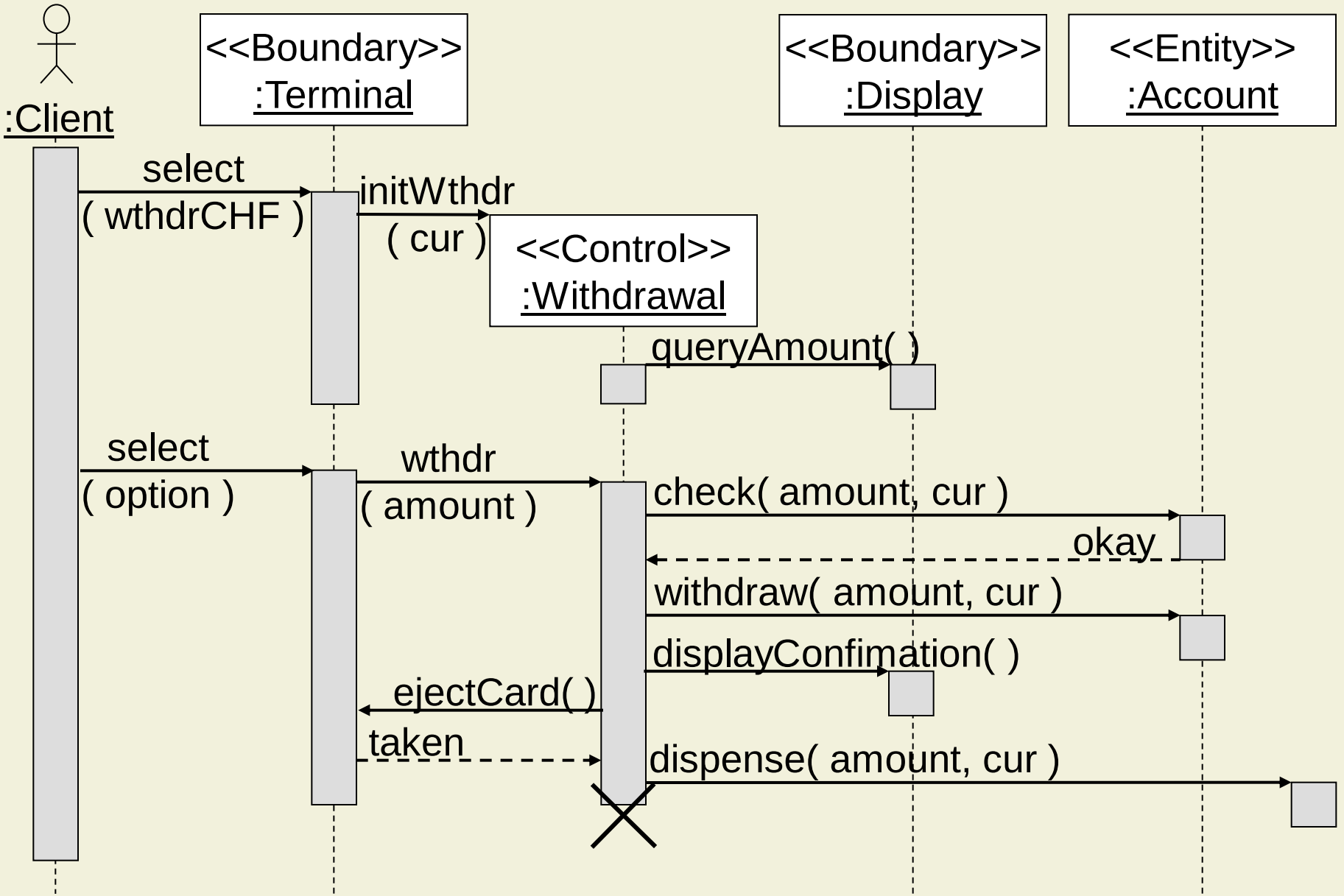
From Use Cases to Sequence Diagrams

- Sequence diagrams are **derived from flows of events** of use cases

- An event always has a **sender** and a **receiver**
  - Find the objects for each event
- Relation to object identification
  - Objects/classes have already been identified during object modeling
  - Additional objects are identified as a result of dynamic modeling

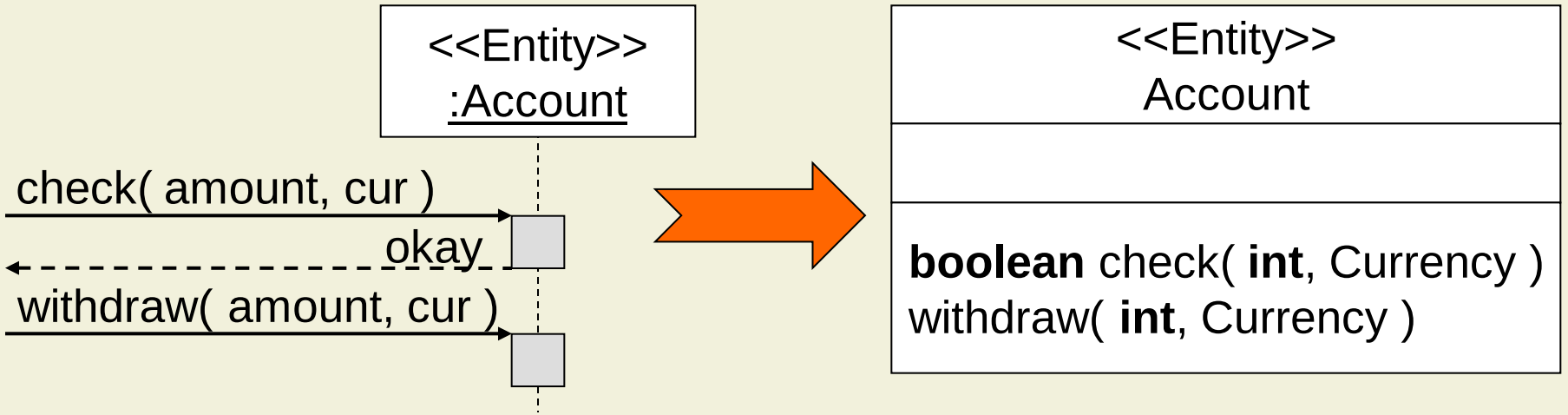
Bankomat Example: Withdraw Event Flow

Actor steps	System Steps
1. Authenticate (use case Authenticate)	
	2. Bankomat displays options
3. Client selects “Withdraw CHF”	
	4. Bankomat queries amount
5. Client enters amount	
	6. Bankomat returns bank card
	7. Bankomat outputs specified amount in CHF


:Client
<<Boundary>>
:Terminal
<<Boundary>>
:Display
<<Entity>>
:Account
select
(wthdrCHF)
initWthdr
(cur)
<<Control>>
:Withdrawal
queryAmount( )
select
(option)
wthdr
(amount)
check(amount, cur)
okay
withdraw(amount, cur)
displayConfimation( )
ejectCard( )
taken
dispense(amount, cur)

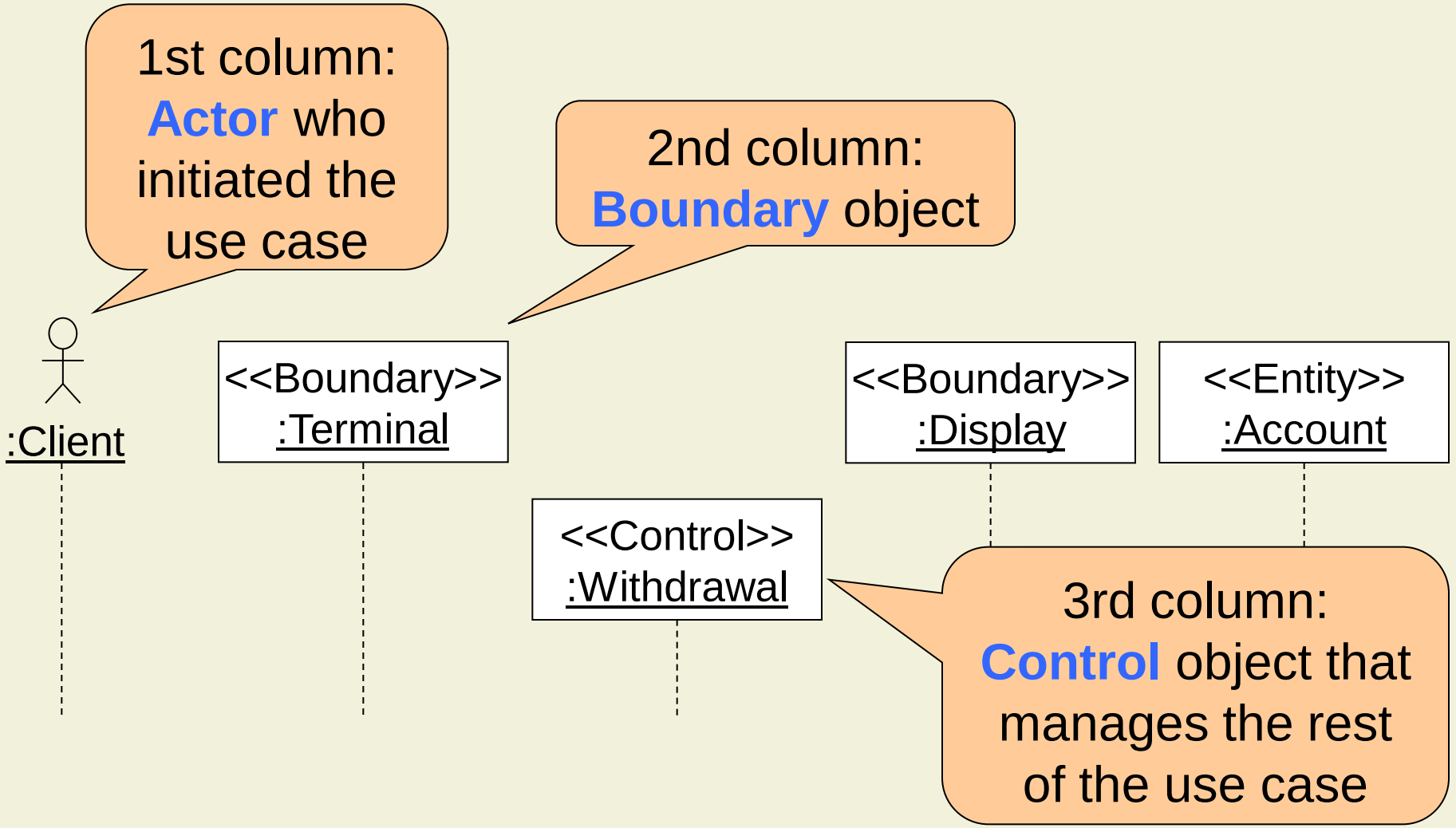
Impact on Object Model

- For each object that receives an event there is a **public operation** in the associated class



- Identify additional objects and classes
  - In the example: Sink for dispense message (CashDispenser)

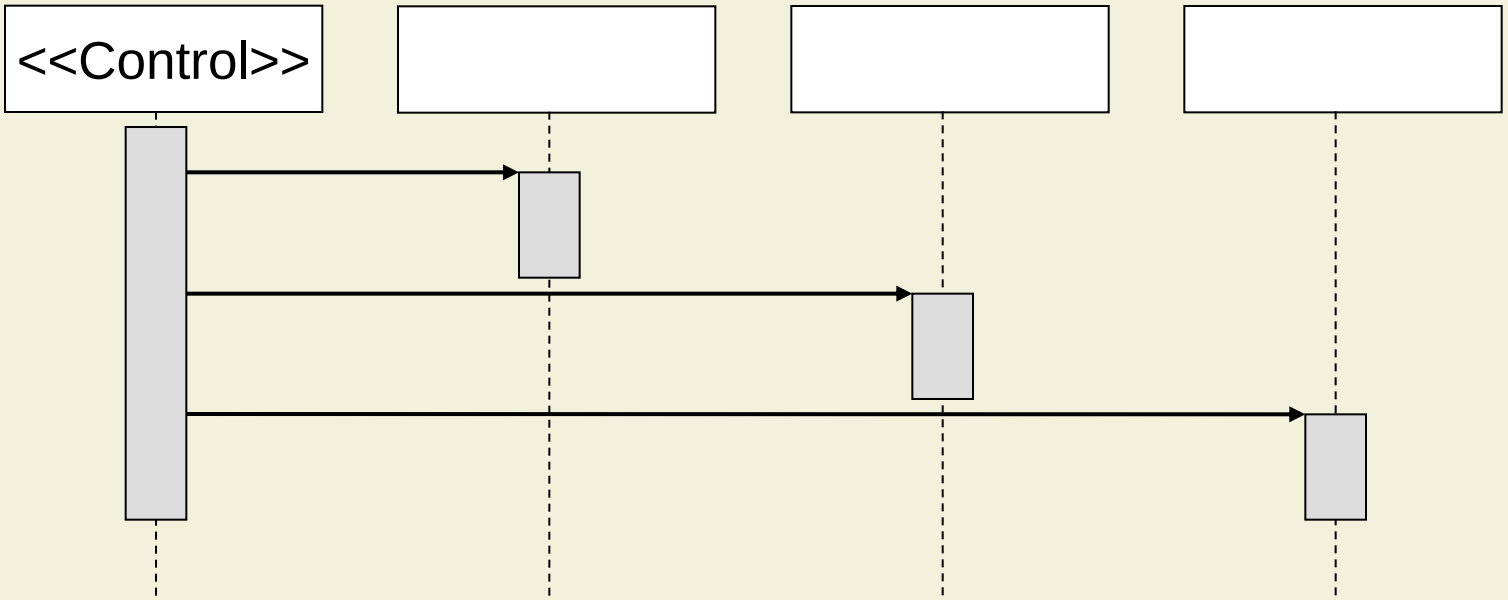
Recommended Layout of Sequence Diagrams



Heuristics for Sequence Diagrams

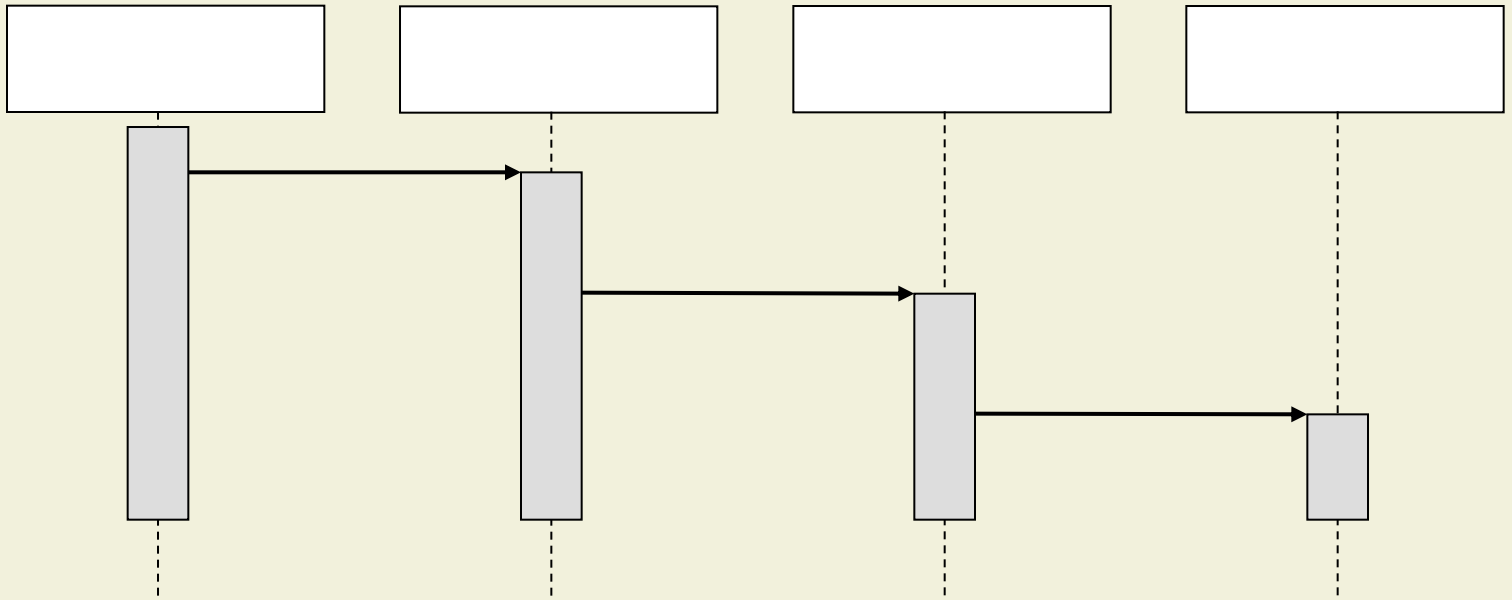
- Creation of objects
  - Control objects are created at the initiation of a use case
  - Boundary objects are often created by control objects
- Access of objects
  - Entity objects are accessed by control and boundary objects
  - Entity objects should never access boundary or control objects
    - Easier to share entity objects across use cases
    - Makes entity objects resilient against technology-induced changes in boundary objects

Fork Structure



- The **dynamic behavior is placed in a single object**, usually a control object
- It knows all the other objects and often uses them for direct queries and commands

Stair Structure



- The **dynamic behavior is distributed**
  - Each object delegates some responsibility to other objects
  - Each object knows only a few of the other objects and knows which objects can help with a specific behavior

Fork or Stair?

- Choose the **stair** (decentralized control) if
  - The operations have a **strong connection**
  - The operations will **always** be performed in the **same order**

- Choose the **fork** (centralized control) if
  - The operations can **change order**
  - **New operations** are expected to be added as a result of new requirements

Sequence Diagrams Summary

- Sequence diagrams represent **behavior** in terms of **interactions**
- **Complement the class diagrams** (which represent structure)

- Useful
  - To find missing objects
  - To detect and supply operations for the object model

- Time consuming to build, but worth the investment

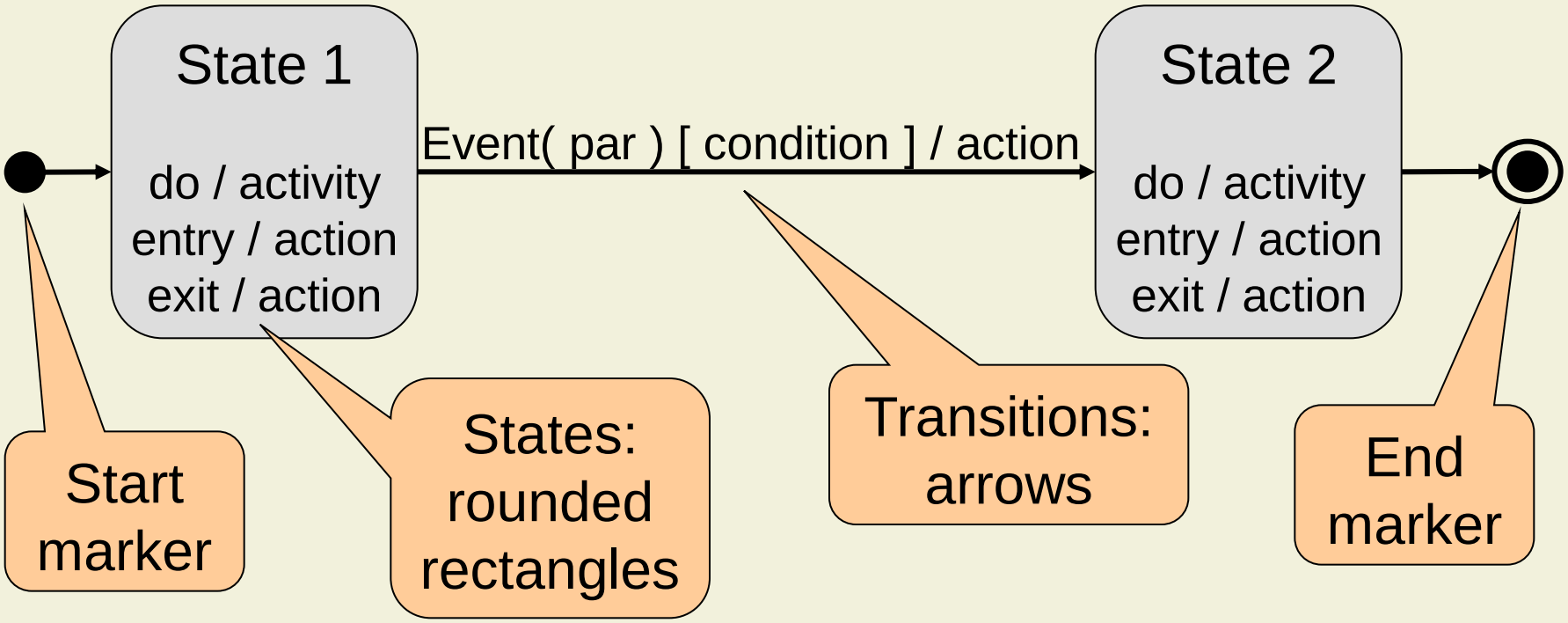
State-Dependent Behavior

- Objects with extended lifespan often have state-dependent behavior
  - Typical for **control objects**
  - Less often for entity objects
  - Almost never for boundary objects
- Examples
  - Withdrawal: has state-dependent behavior
  - Account: has state-dependent behavior (e.g., locked)
  - Display: does not have state-dependent behavior
- State-dependent behavior is modeled **only if necessary**

Events, Actions, and Activities

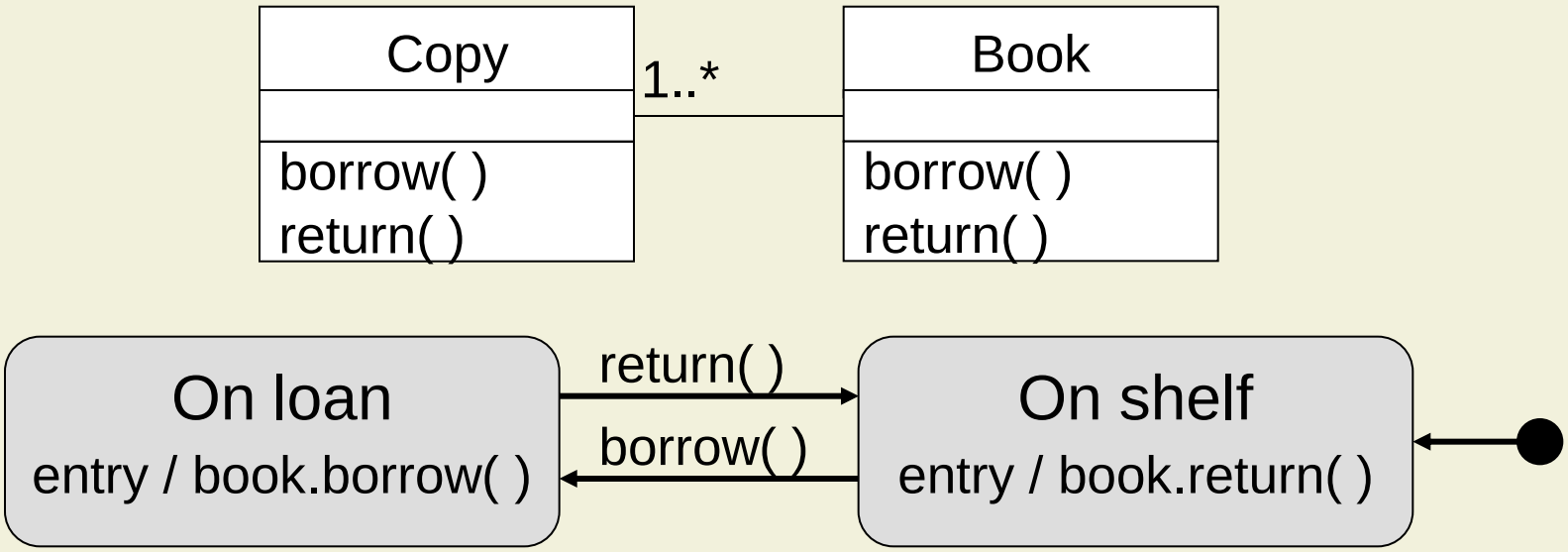
- **Event**: Something that happens at a point in time
  - Typical event: Receipt of a message
  - Other events: Change event for a condition, time event
- **Action**: Operation in **response to an event**
  - Example: Object performs a computation upon receipt of a message
- **Activity**: **Operation performed** as long as object is **in some state**
  - Example: Object performs a computation without external trigger

UML State Diagrams



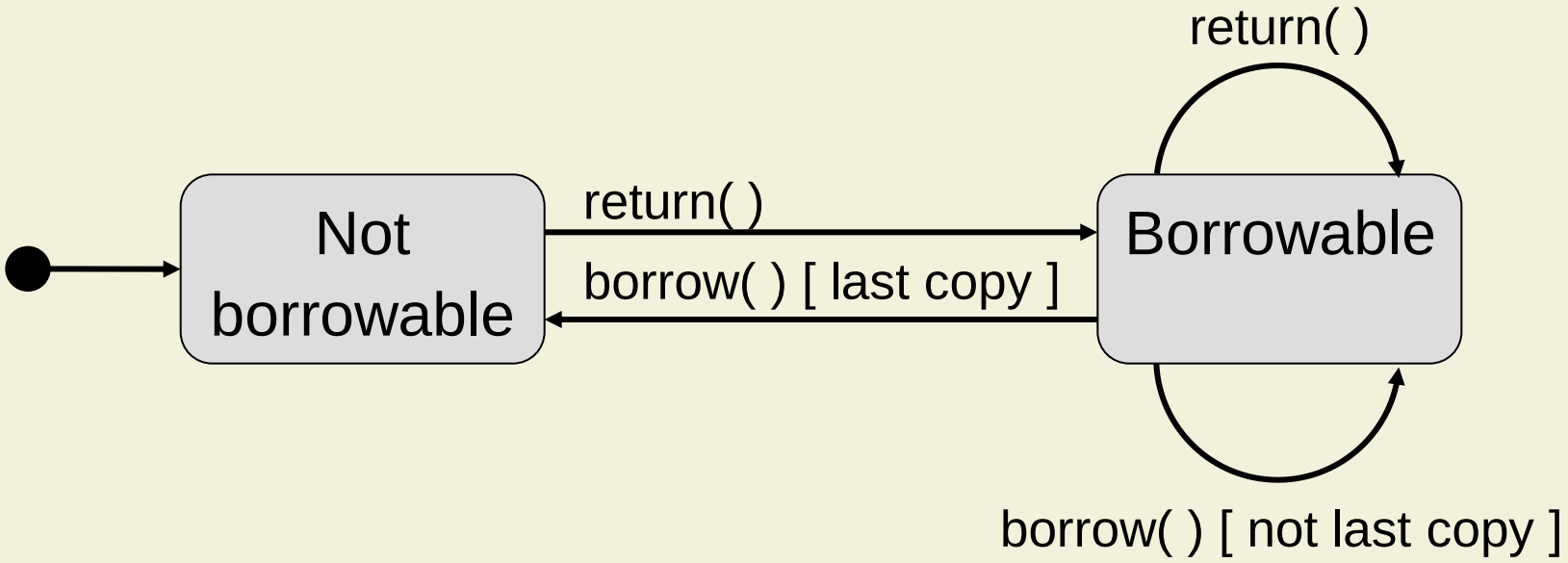
- State diagram relates events and states for a class
- Often called “state chart” or “state chart diagram”

Example 1: States of Copy Objects



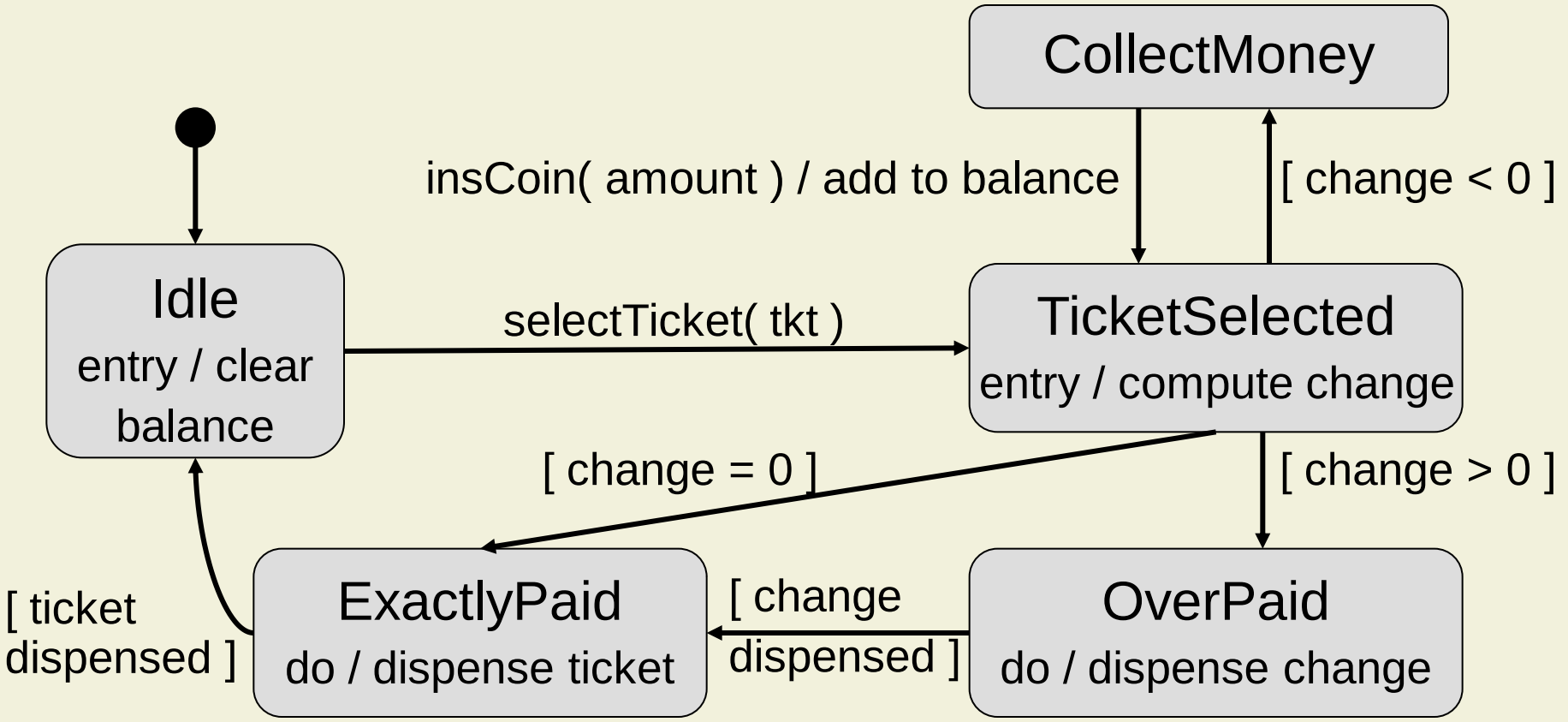
- Implementation has to take care of **unexpected messages**, e.g., return in state “on shelf”
  - Specify precondition
  - Report an error, throw an exception

Example 2: States of Book Objects



- Events can have different effects depending on guard conditions
- Some state diagrams do not have end markers

Example 3: Ticket Vending Machine



State

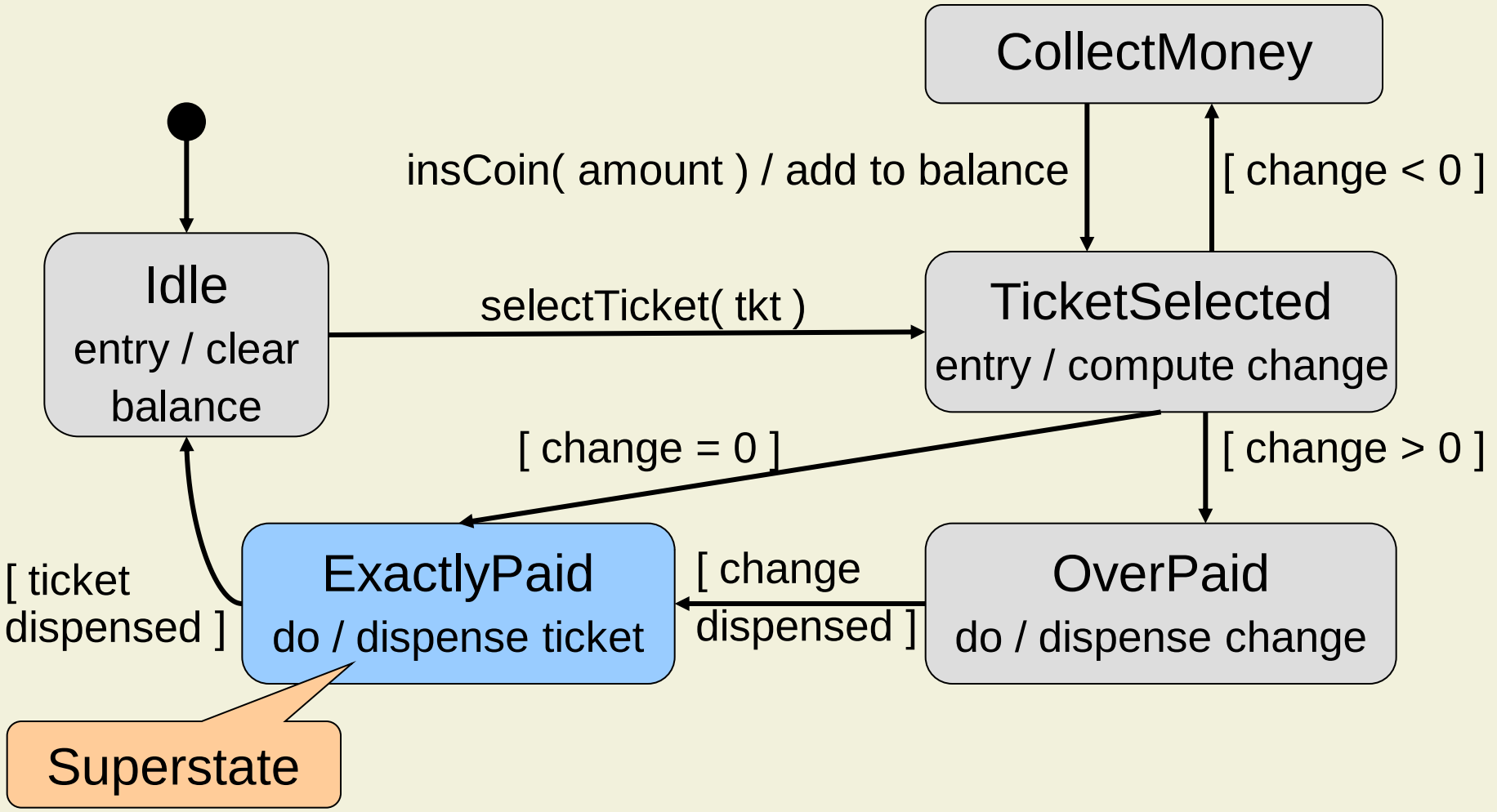
- An **abstraction** of the **attribute values** of an object
- A state is an equivalence class of all those attribute values and links that do not need to be distinguished as far as the control structure of the class or the system is concerned
- Example: State of a book
  - A book is either borrowable or not
  - Omissions: bibliographic data
  - All borrowable books are in the same equivalence class, independent of their author, title, etc.

Nested State Diagrams

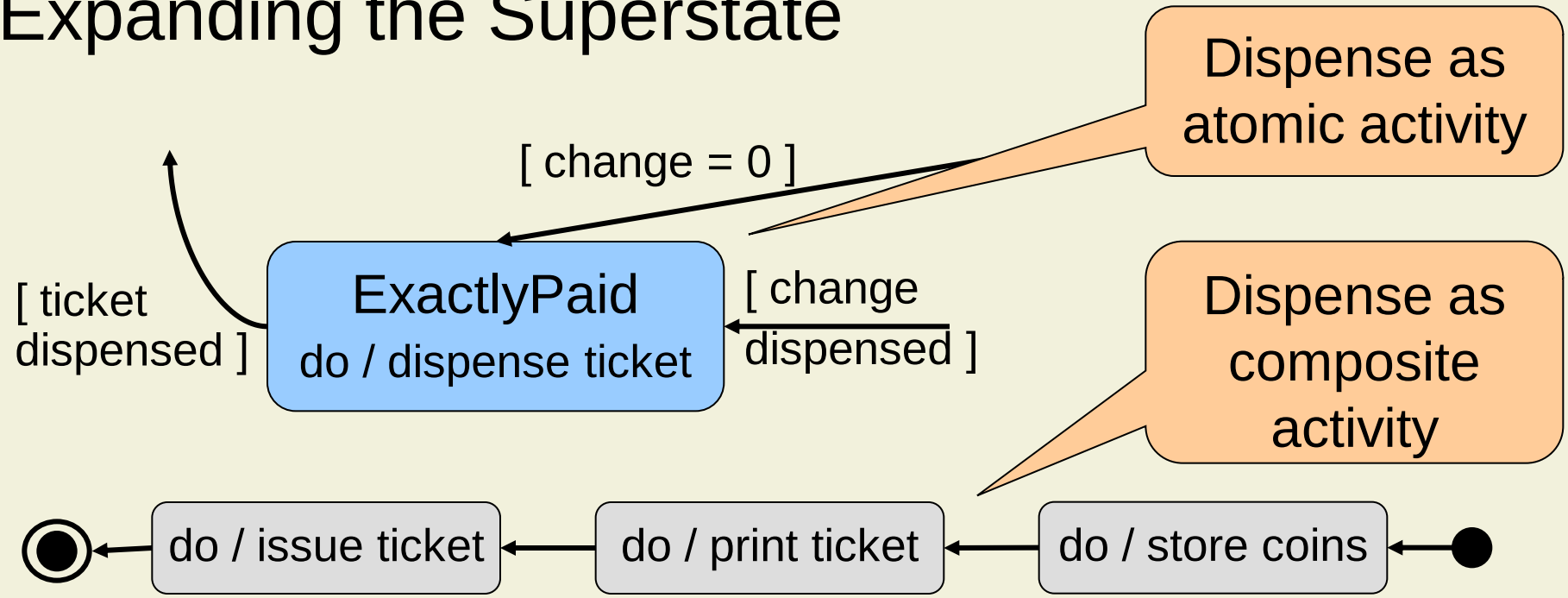
- Activities in states can be **composite items** that denote other state diagrams

- Sets of substates in a nested state diagram can be denoted with a superstate
  - Avoid spaghetti models
  - Reduce the number of lines in a state diagram

Example: Superstate



Expanding the Superstate



- **Transitions from** other states to the superstate **enter the first substate** of the superstate
- **Transitions to** other states from a superstate are **inherited by all the substates** (state inheritance)

State Diagram vs. Sequence Diagram

- **State diagrams** help to identify
  - Changes to an individual object over time

- **Sequence diagrams** help to identify
  - The temporal relationship of between objects
  - Sequence of operations as a response to one or more events

Practical Tips for Dynamic Modeling

- Construct dynamic models only for classes with **significant** dynamic behavior
  - Avoid “analysis paralysis”
- Consider only **relevant** attributes
  - Use abstraction if necessary
- Look at the granularity of the application when deciding on actions and activities
- Reduce notational clutter
  - Try to put actions into superstate boxes (look for identical actions on events leading to the same state)

Requirements Analysis Document

1. Introduction
   1. Purpose and scope of the System
   2. Objectives and success criteria of the project
   3. Definitions, acronyms, references, overview
2. Current System
3. Proposed System
   1. Overview
   2. Functional requirements
   3. Nonfunctional requirements
   4. **System models**
4. Glossary

Section 3.4 System Model

3.4.1 Scenarios

- As-is scenarios, visionary scenarios

3.4.2 Use case model

- Actors and use cases

3.4.3 Object model

- Data dictionary
- Class diagrams: classes, associations, attributes, operations

3.4.4 Dynamic model

- State diagrams for classes with significant dynamic behavior
- Sequence diagrams for collaborating objects (protocol)

3.4.5 User Interface

Summary: System Models

1. What are the transformations? → **Functional Model**
   - Create **scenarios** and **use case diagrams**
   - Talk to client, observe, get historical records
2. What is the structure of the system? → **Object Model**
   - Create **class diagrams**
   - Identify objects, associations and their multiplicity, attributes, operations
3. What is its behavior? → **Dynamic Model**
   - Create **sequence diagrams**
   - Show senders, receivers, and sequence of events
   - Create **state diagrams** (for the interesting objects)

Dominance of Models

- Object model
  - The system has classes with **nontrivial states** and **many relationships** between the classes
- Dynamic model
  - The model has **many different** types of **events**: Input, output, exceptions, errors, etc.
- Functional model
  - The model performs **complicated transformations** (e.g., computations consisting of many steps)

Dominance of Models: Examples

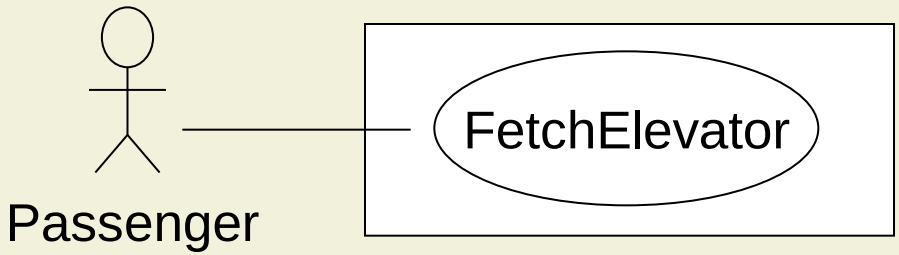
- Compiler: Functional model
  - Dynamic model is trivial (there is only one type input and only a few outputs)
- Database systems: Object model
  - Functional model is trivial (the purpose of the functions is usually to store, organize, and retrieve data)
- Spreadsheet program: Functional model
  - Dynamic model is interesting if the program allows computations on a cell
  - Object model is trivial (spreadsheet values are trivial; the only interesting object is the cell)

3. Analysis

Elevator Control: Problem Statement

- The elevator has one button for each floor
  - Illuminate when pressed
  - Cause the elevator to visit the corresponding floor
  - Illumination is canceled when the elevator visits the corresponding floor
- Each floor, except the first floor and top floor has two buttons to request the elevator to go up or down, respectively
  - Illuminate when pressed
  - Causing the elevator to visit the corresponding floor
  - Illumination is canceled when the elevator visits the floor and then moves in the desired direction

Use Case: Fetch Elevator

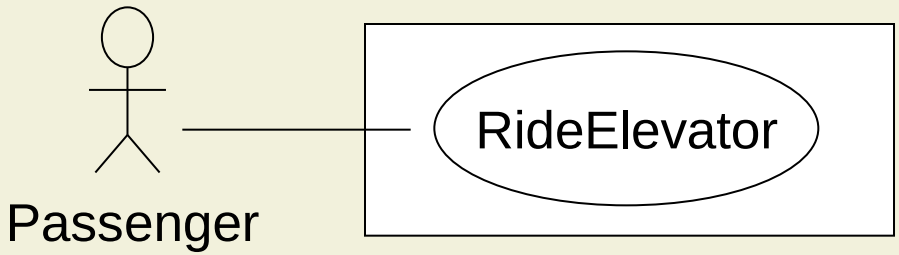


- Initiating actor: Passenger
- Entry condition:
  Passenger is in the hall
- Exit condition:
  Elevator is on requested floor with doors open

Flow of Events:

- Passenger pushes hall button
- System illuminates button
- System closes elevator doors
- System moves elevator to requested floor
- System cancels illumination
- System opens elevator doors

Use Case: Ride Elevator

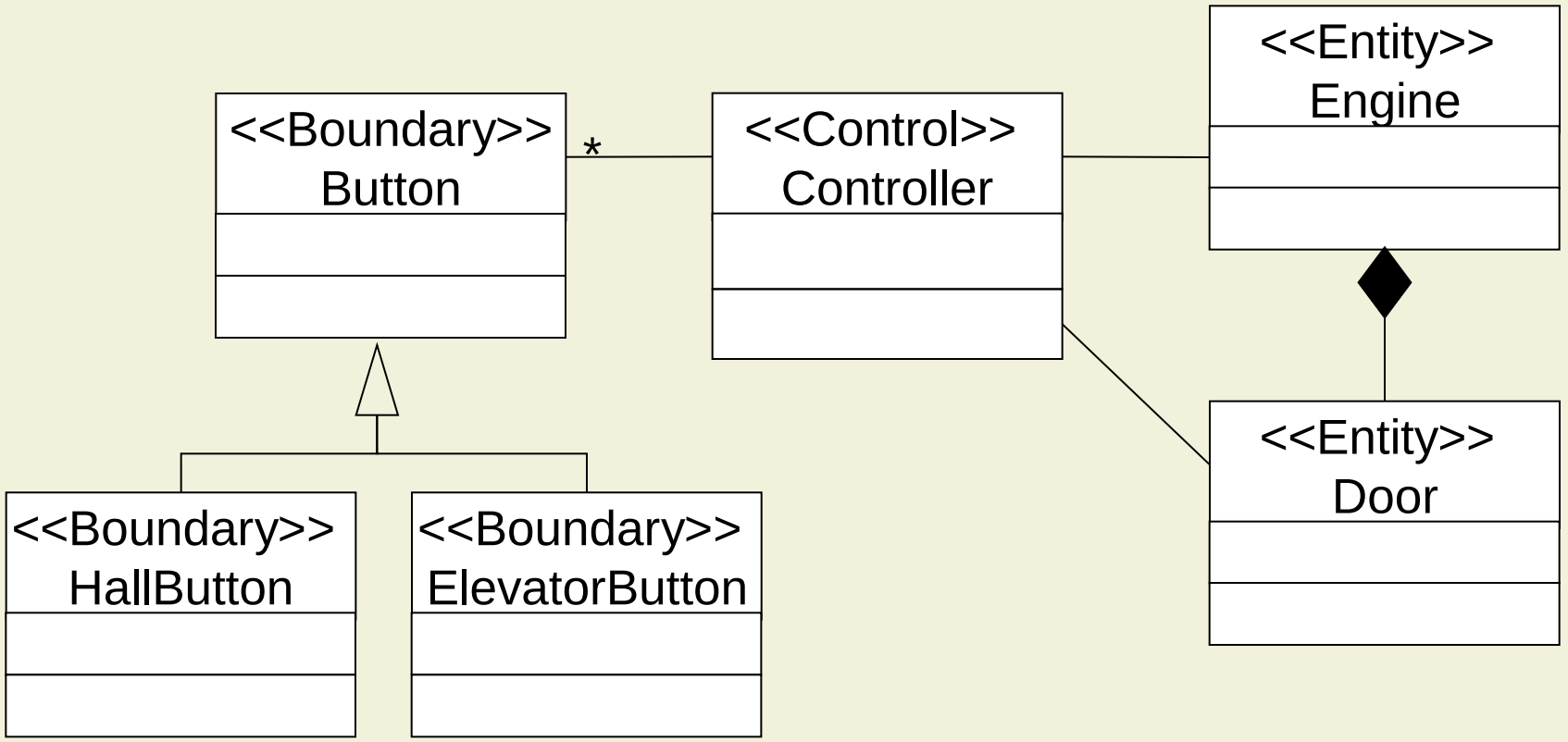


- Initiating actor: Passenger
- Entry condition: Passenger is inside the elevator
- Exit condition: Elevator is on requested floor with doors open

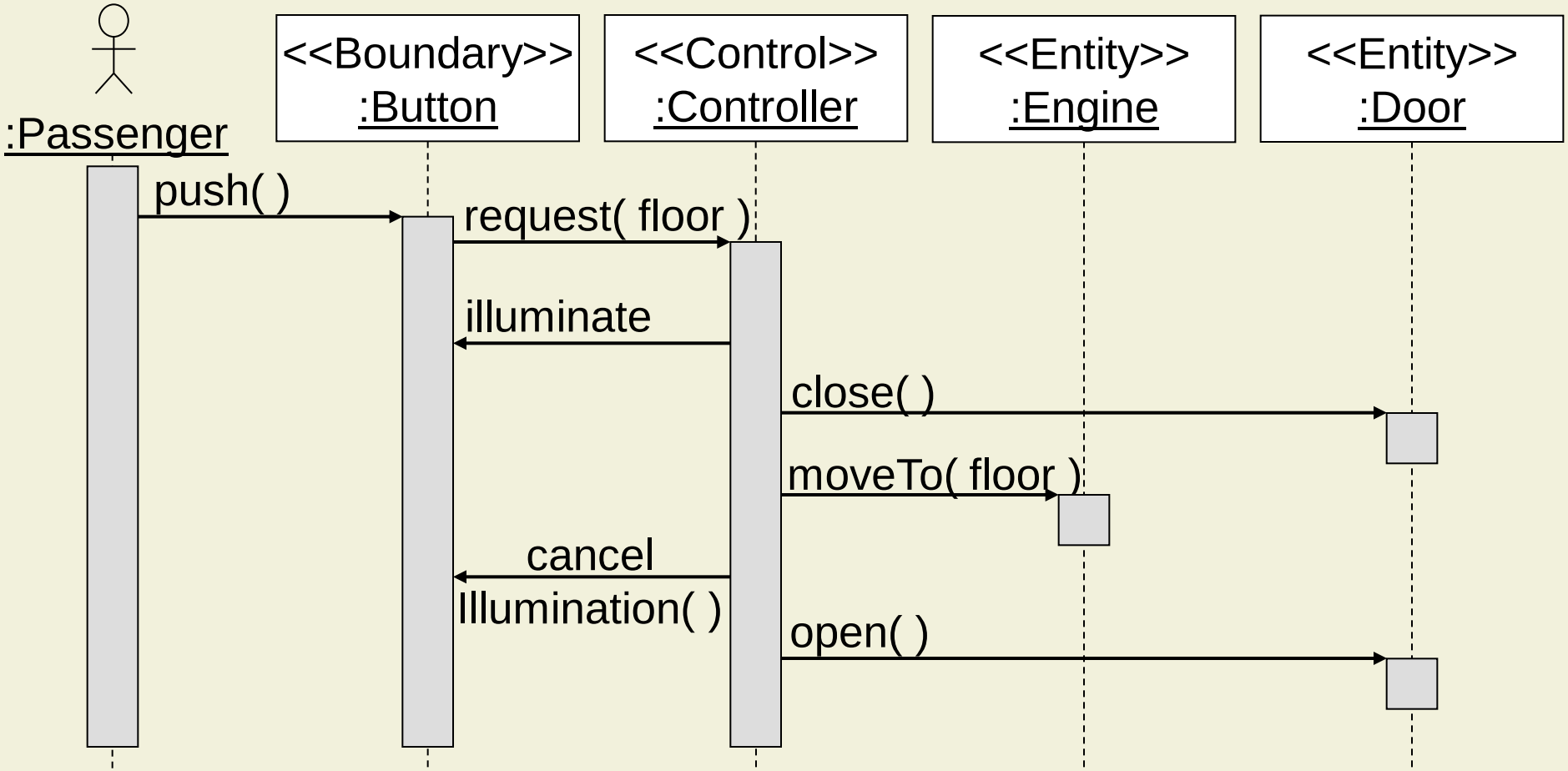
Flow of Events:

- Passenger pushes elevator button
- System illuminates button
- System closes elevator doors
- System moves elevator to requested floor
- System cancels illumination
- System opens elevator doors

Initial Analysis Object Model



Sequence Diagram: Fetch and Ride Elevator

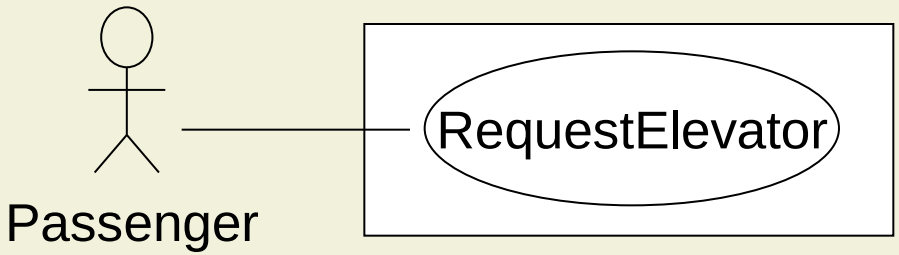


Iteration: Missed Requirements

- The project manager decides that the analysis results should also be discussed with the hardware engineer

- Engine cannot be told to move to a given floor
- Messages understood by the engine:
  - Start moving in a given direction
  - Stop moving
- Sensors are used to determine position of elevator
  - Sensors send signal when floor is reached

Use Case: Request Elevator

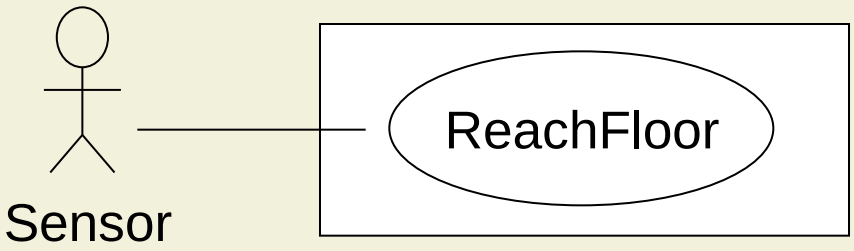


- Initiating actor: Passenger
- Entry condition: –
- Exit condition:
  Elevator starts moving towards requested floor

Flow of Events:

- Passenger pushes button
- System illuminates button
- System closes elevator doors
- System initiates elevator to move to requested floor

Use Case: Reach Floor

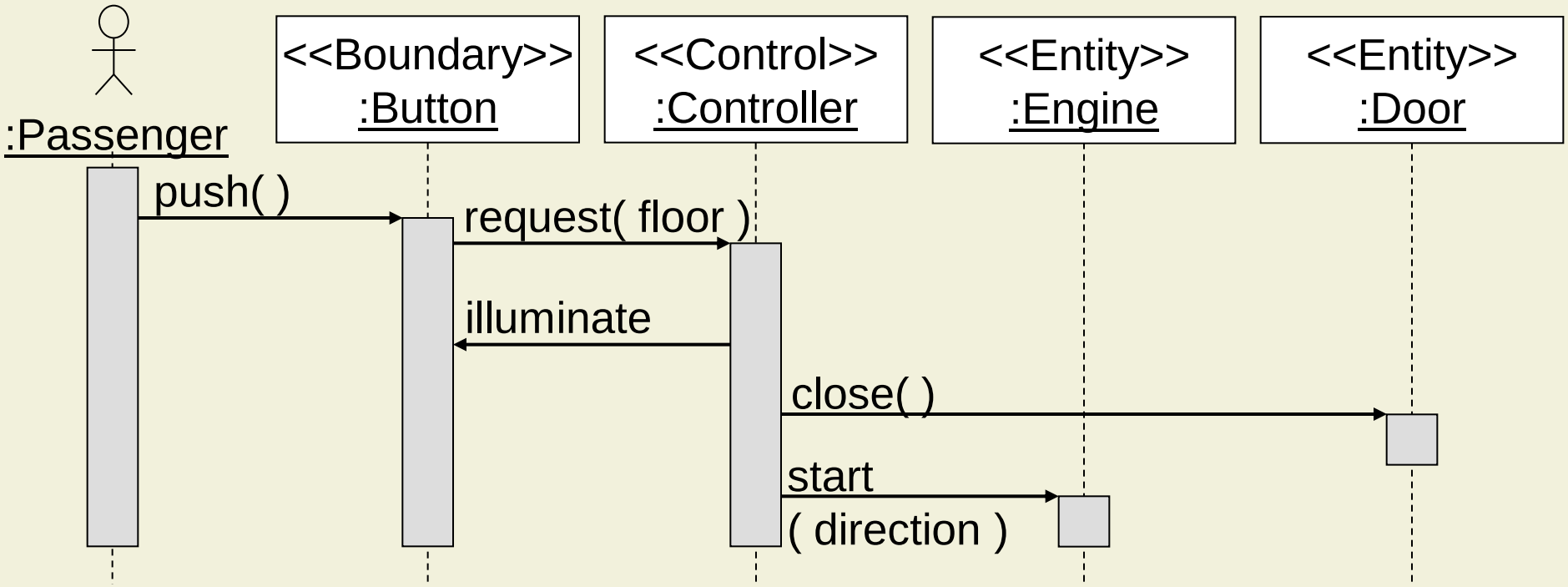


- Initiating actor: Sensor
- Entry condition:
  Elevator is moving to requested floor
- Exit condition:
  Elevator is stopped on requested floor with doors open

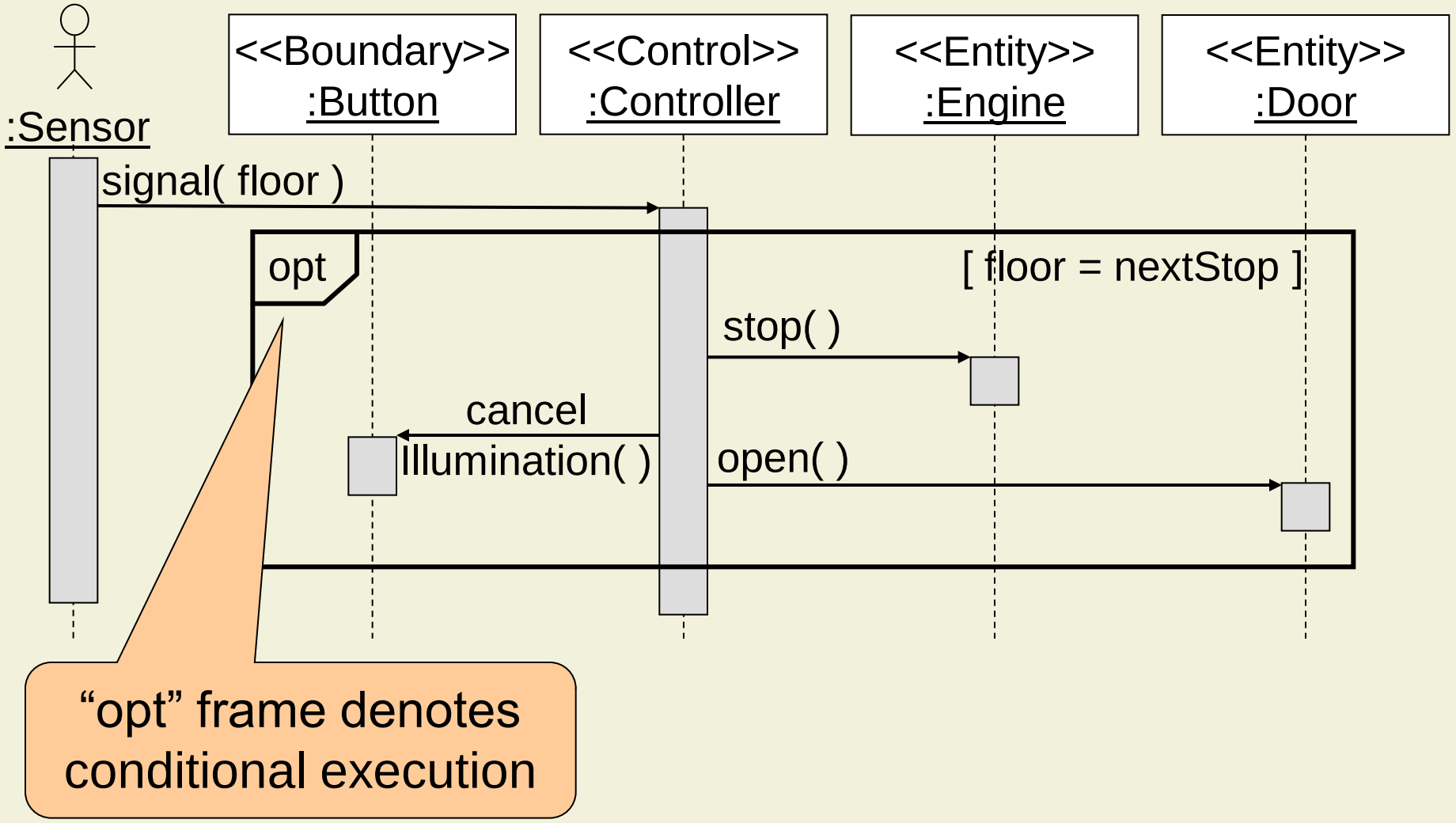
Flow of Events:

- Sensor signals that some floor is reached
- System stops elevator
- System cancels illumination of button
- System opens elevator doors

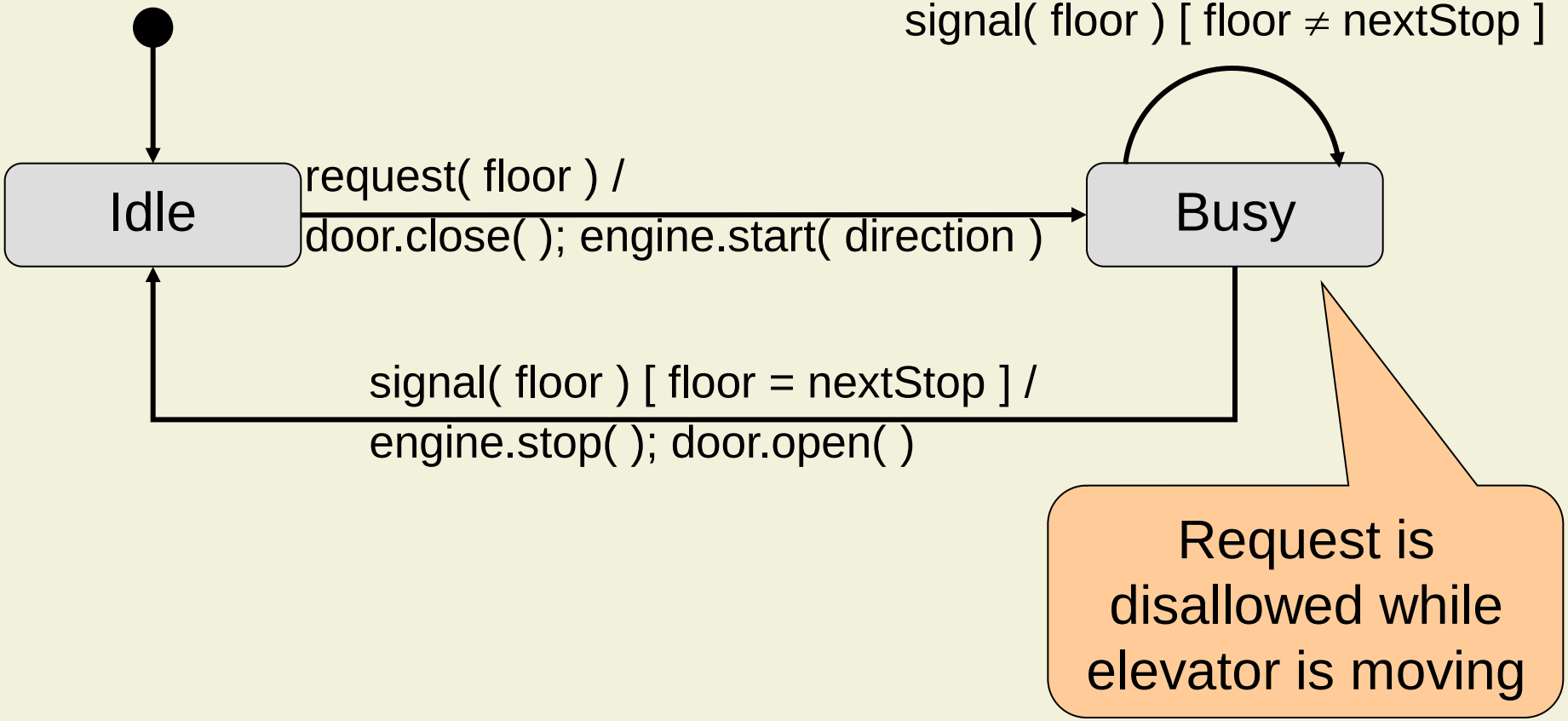
Sequence Diagram: Request Elevator



Sequence Diagram: Reach Floor



State Diagram: Controller



A More Realistic Elevator

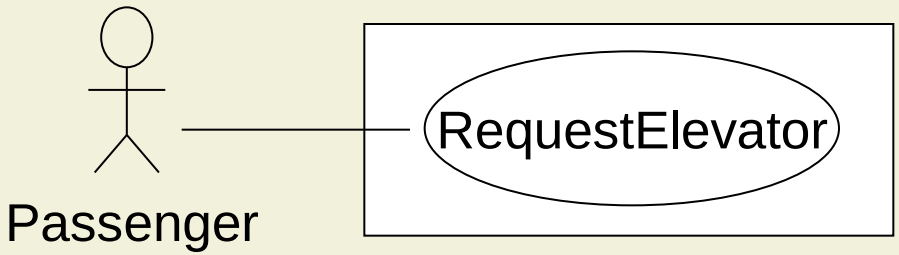
- Additional business requirements
- Requests shall be accepted at any time
  - Also when elevator is moving
- System keeps track of all pending requests
  - Processing order not specified
- Elevator serves requests on its way immediately
  - Detailed by scenario

- We ignore illumination of buttons and operation of doors in the following

Scenario: Processing Requests on the Way

1. Alice enters elevator on first floor and pushes button for fifth floor
2. System initiates elevator to move to fifth floor
3. When elevator is on second floor, Bob pushes hall button on third floor
4. System stops elevator on third floor
5. Bob enters elevator and pushes button for sixth floor
6. System initiates elevator to move to fifth floor
7. System stops elevator on fifth floor
8. Alice gets off
9. System initiates elevator to move to sixth floor
10. System stops elevator on sixth floor
11. Bob gets off

Use Case: Request Elevator

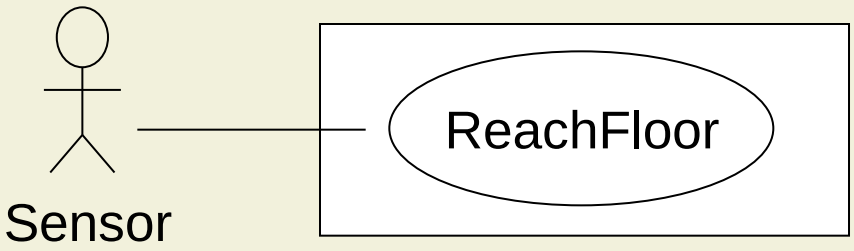


- Initiating actor: Passenger
- Entry condition: –
- Exit condition:
  - System stores new request
  - If idle, elevator started moving towards requested floor

Flow of Events:

- Passenger pushes button
- System determines next stop (a previous or new request)
- System initiates elevator to move to determined next stop

Use Case: Reach Floor

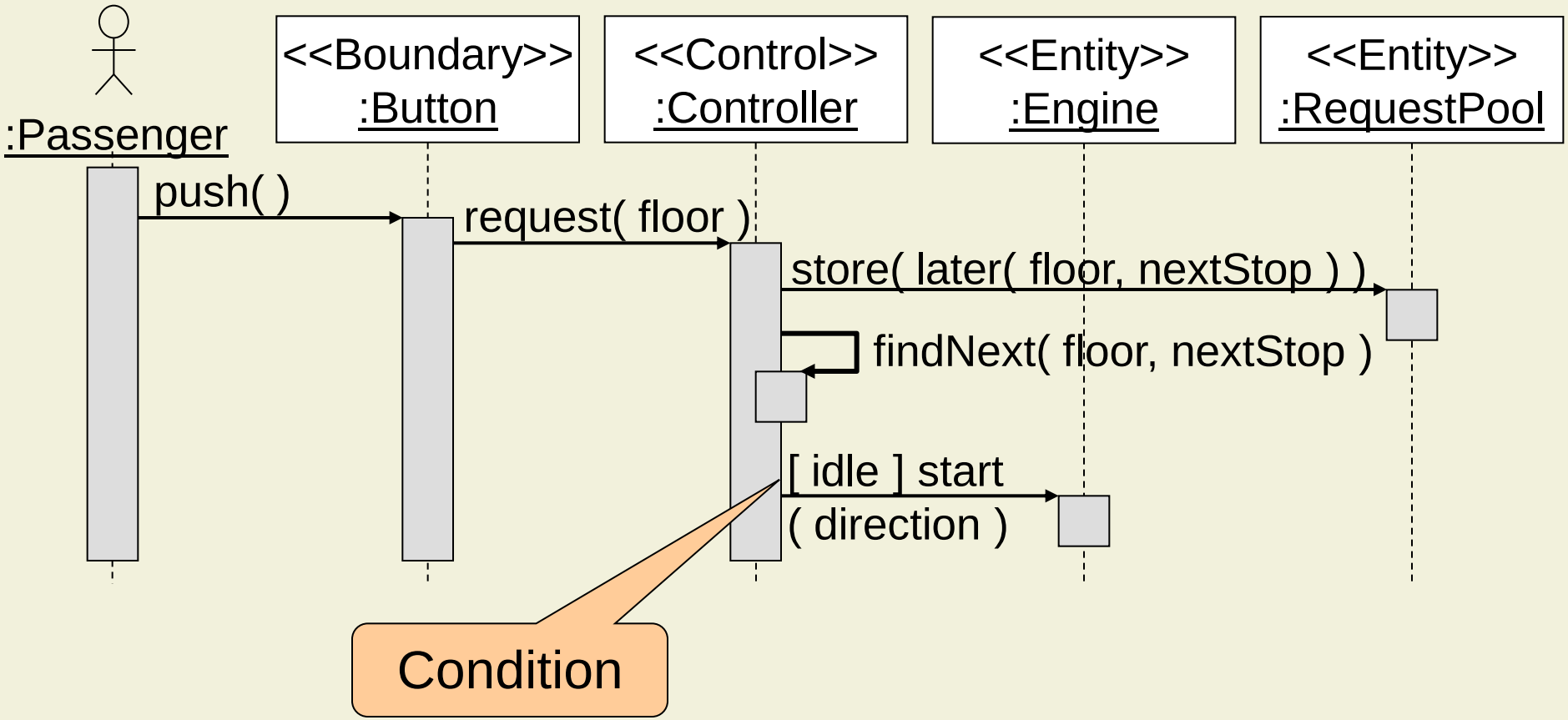


- Initiating actor: Sensor
- Entry condition:
  - Elevator is moving towards requested floor
- Exit condition:
  - Elevator had stopped on a requested floor
  - Elevator is moving to next requested floor

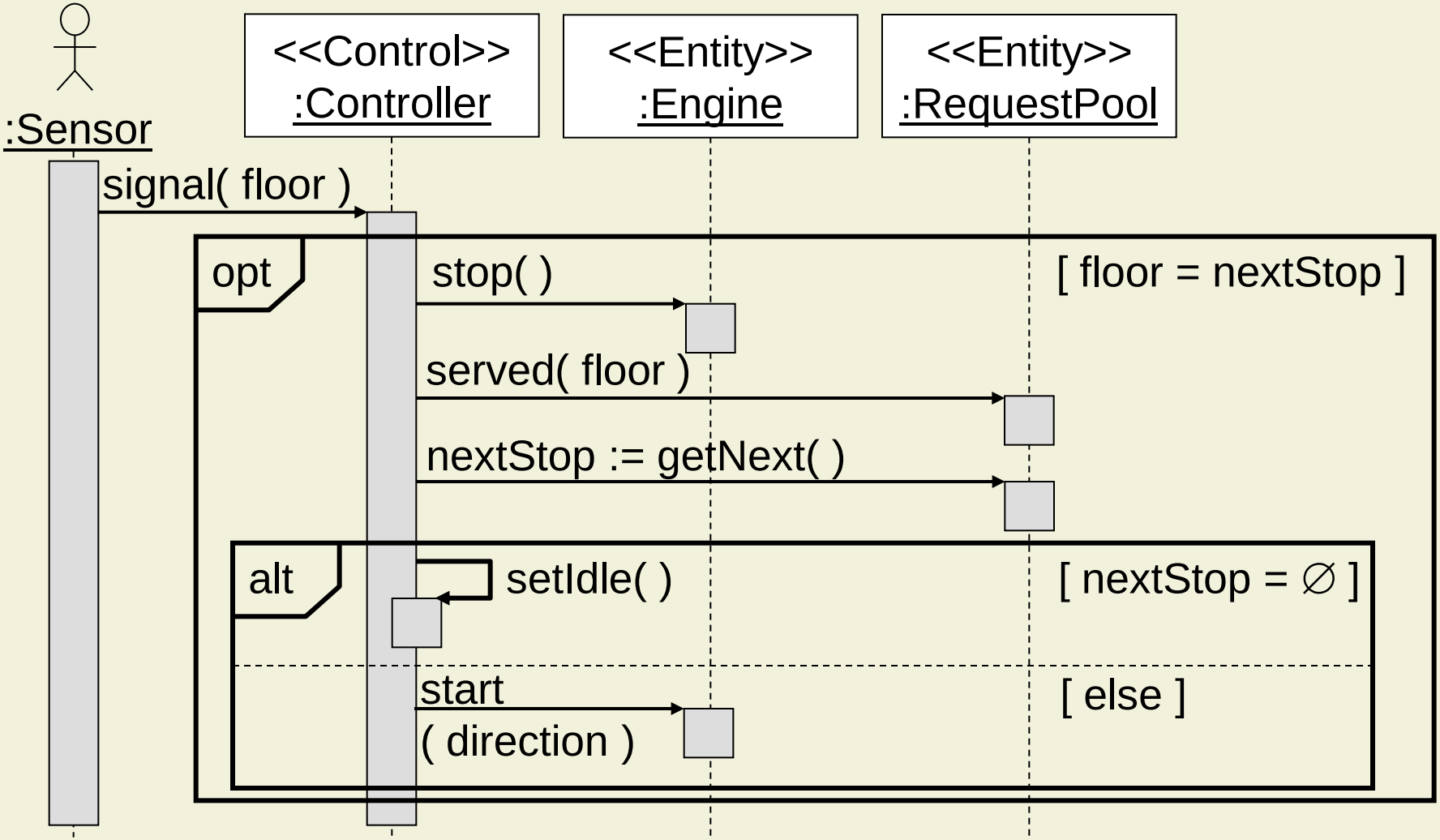
Flow of Events:

- Sensor signals that some floor is reached
- System stops elevator if a requested floor is reached
- System chooses next request (extension point)
- System initiates elevator to move to requested floor

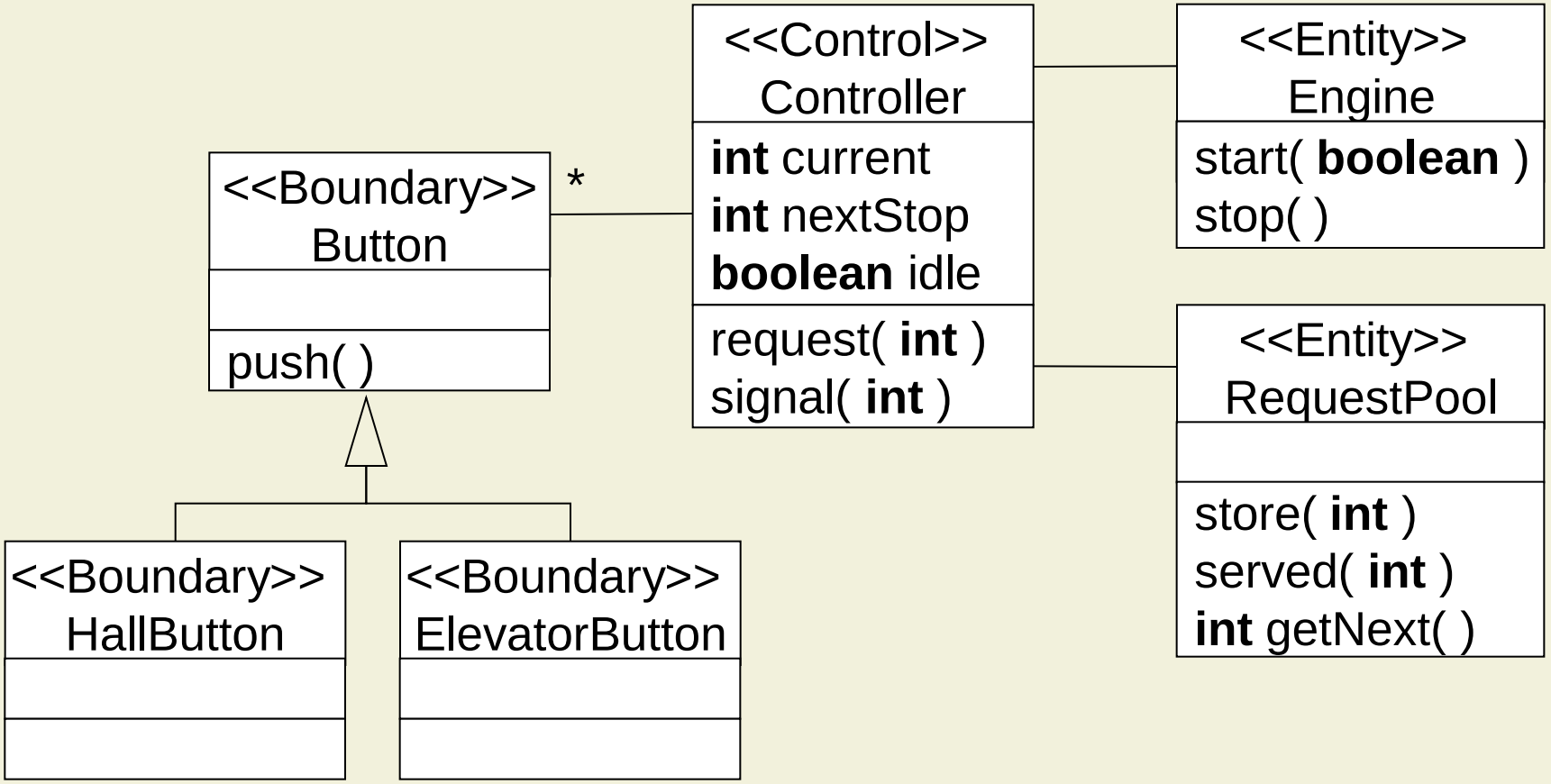
Sequence Diagram: Request Elevator



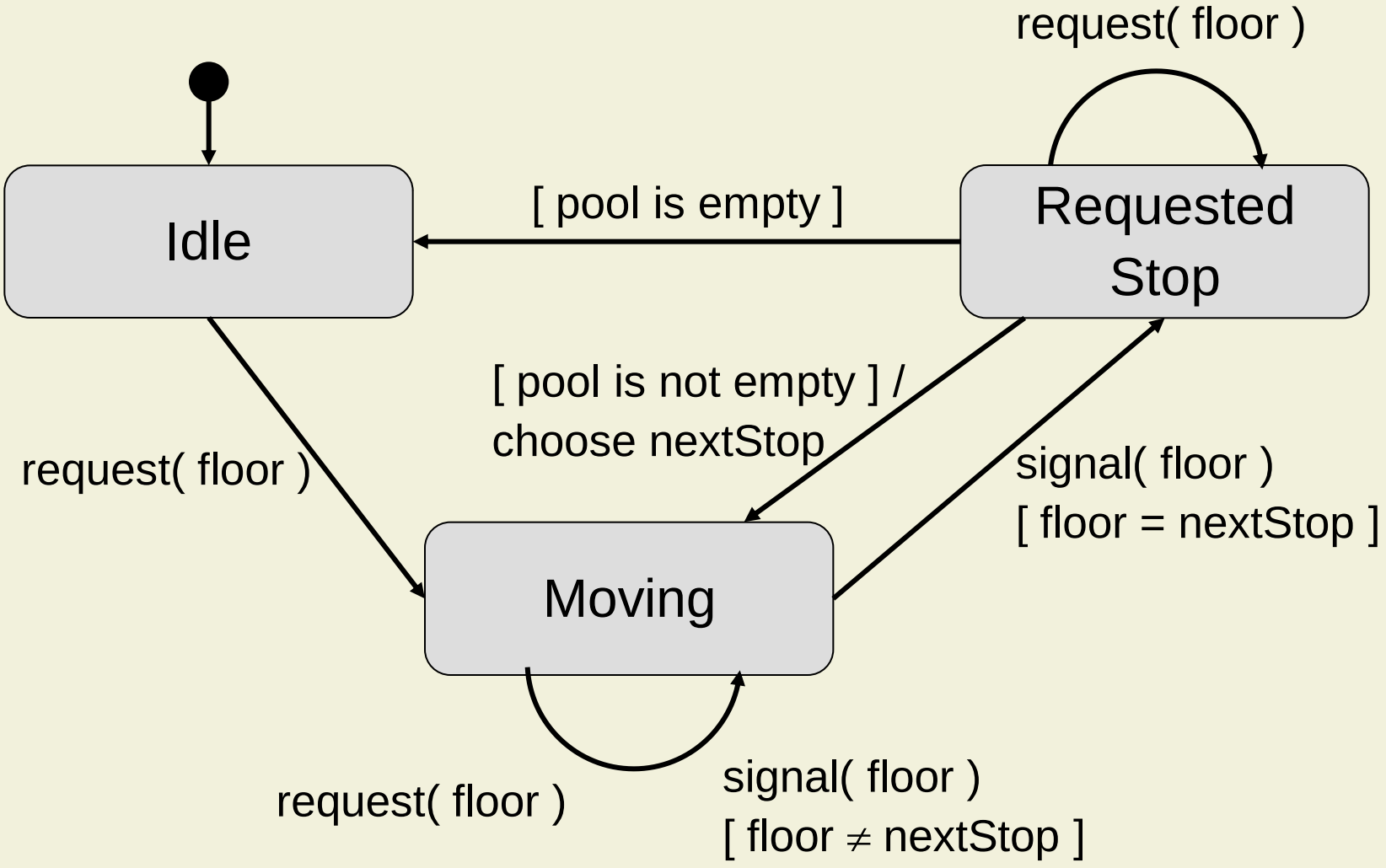
Sequence Diagram: Reach Floor



Class Diagram

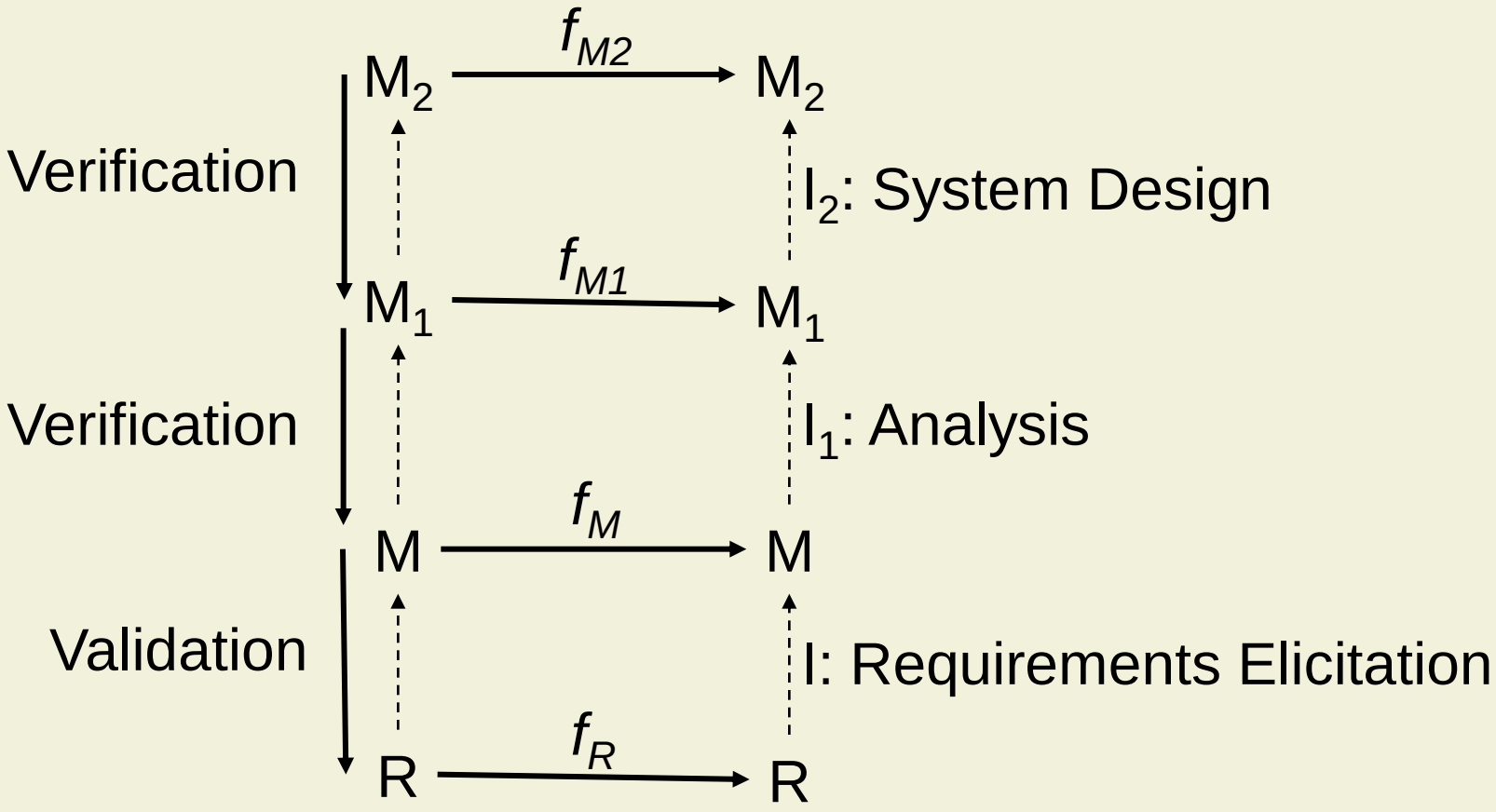


State Diagram: Controller



3. Analysis

Validation and Verification of Models



Validation and Verification of Models (cont'd)

- Verification is a comparison of **two models**
  - Determining that a model accurately represents another model
  - One can prove a **refinement** relation (rarely done in practice)
- Validation is a comparison of a **model to reality**
  - Reality can be an artificial system, (e.g., legacy system)
  - **Validation is** a **critical** step in the development process
- Requirements should be validated with the client and the user
  - Technique: Formal and informal **requirements reviews**

Checklist for a Requirements Review

- Is the model **correct**?
  - Everything is the model represents an aspect of reality
- Is the model **complete**?
  - Every scenario, including exceptions, is described
- Is the model **consistent**?
  - The model does not have components that contradict themselves (for example, deliver contradicting results)
- Is the model **unambiguous**?
  - The model describes one system (one reality), not many
- Is the model **realistic**?
  - The model can be implemented without problems

Checklist for a Requirements Review (cont'd)

- One problem with modeling: We describe a system model with **many different views**
  - Use cases, class, sequence, and state diagrams
- We need to check the **equivalence** of these views
- **Syntactical check** of the models
  - Consistent naming of classes, attributes, methods
  - No dangling associations ("pointing to nowhere")
  - No double-defined classes
  - No missing classes (mentioned but not defined)
  - No classes with the same name but different meanings

Analysis Activities Summary