Introduction to Software Engineering

ECSE-321

Unit 7 - Analysis

Elicitation vs. Analysis

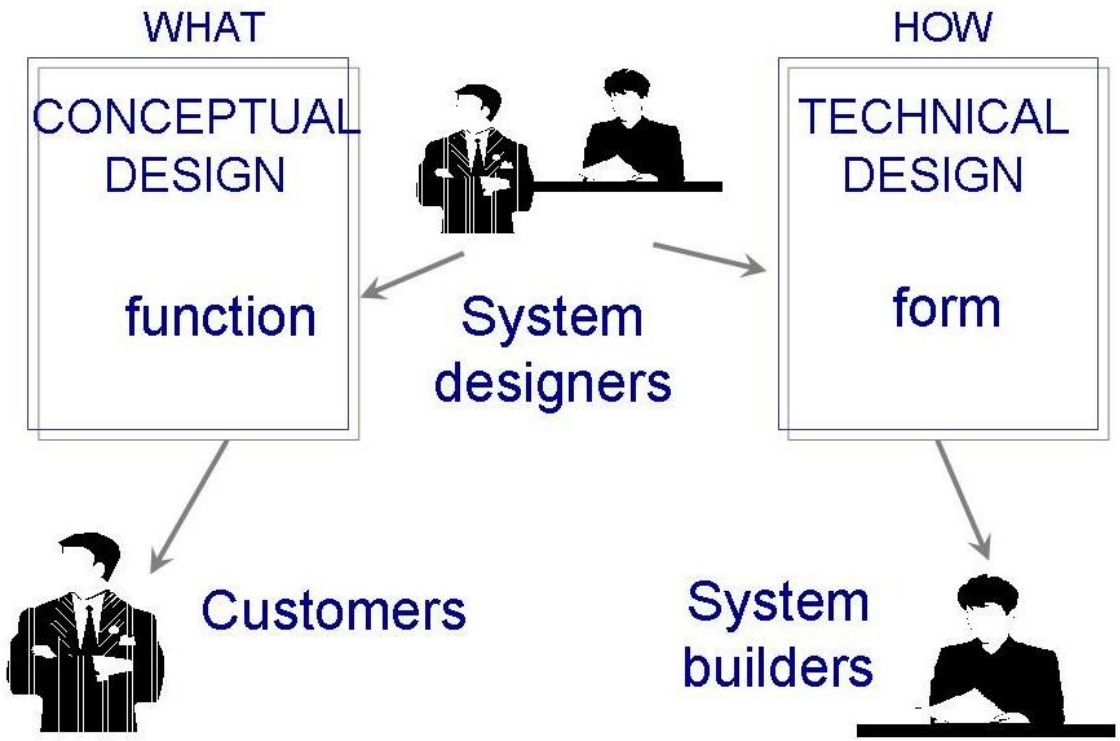
- Requirements Elicitation
  - Understand the user's needs, desires
- Analysis
  - Structure and formalize the requirements
  - Produce a model of the system
    - From user's perspective
    - Still ***NO*** implementation-specific information

What Is Design?

- **Design** is the creative process of transforming the problem into a solution
- The ***description of a solution*** is also known as design
  - The ***requirements specification defines*** the problem
  - The design document specifies a particular solution to the problem

What Is Design?

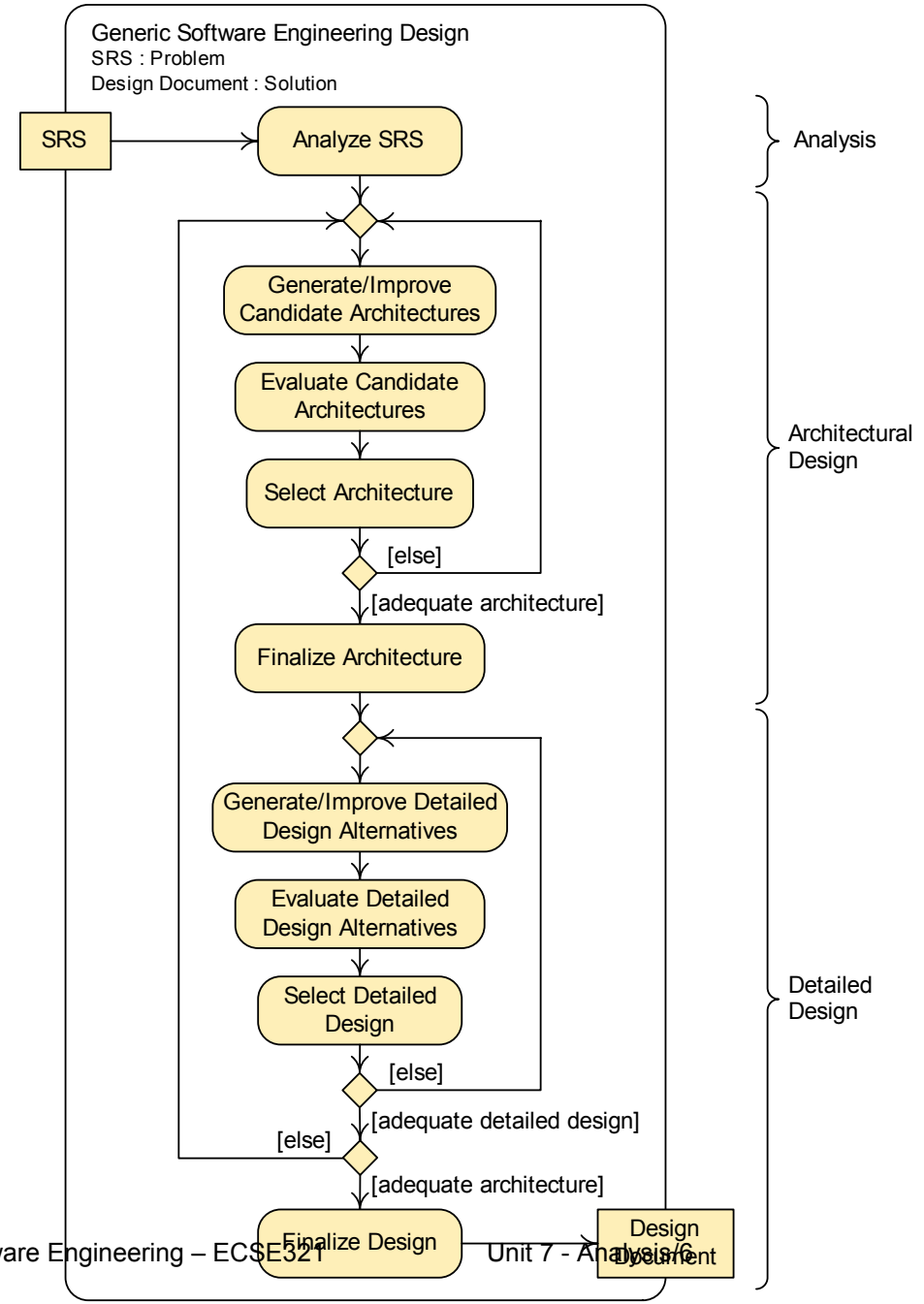
- Design is a two-part interactive process
  - Conceptual design (referred to as ***analysis*** here) – refines requirements
  - Technical design



Analysis Goals, Inputs, and Activities

- Understand an engineering design problem using
  - SRS (software requirements specification)
  - Product design models
- Achieve understanding by
  - Studying the SRS and design models
  - Making analysis models

Engineering Design Process



Analysis Models

An **analysis model** is any representation of a design problem.

- Both static and dynamic models
  - Static models – using information that does not change
  - Dynamic models – using information that does change (behavioural)
- Object-oriented and other kinds of models

Class and Object Models

A **class (object) model** is a representation of classes (objects) in a problem or a software solution.

- Class (object) diagrams are graphical forms of class (object) models

Types of Class Models

- *Analysis or conceptual models*—Important entities or concepts in the problem, their attributes, important relationships

Problem

- *Design class models*—Classes in a software system, attributes, operations, associations, but no implementation details
- *Implementation class models*—Classes in a software system with implementation details

Solution

Conceptual Modeling

Conceptual models are about real-world entities in the problem domain and not about software

- Conceptual models are useful for:
  - understanding the problem domain – identify important entities, their characteristics, relations to one another
  - setting data requirements – data requirements can be extracted from conceptual models
  - validating requirements

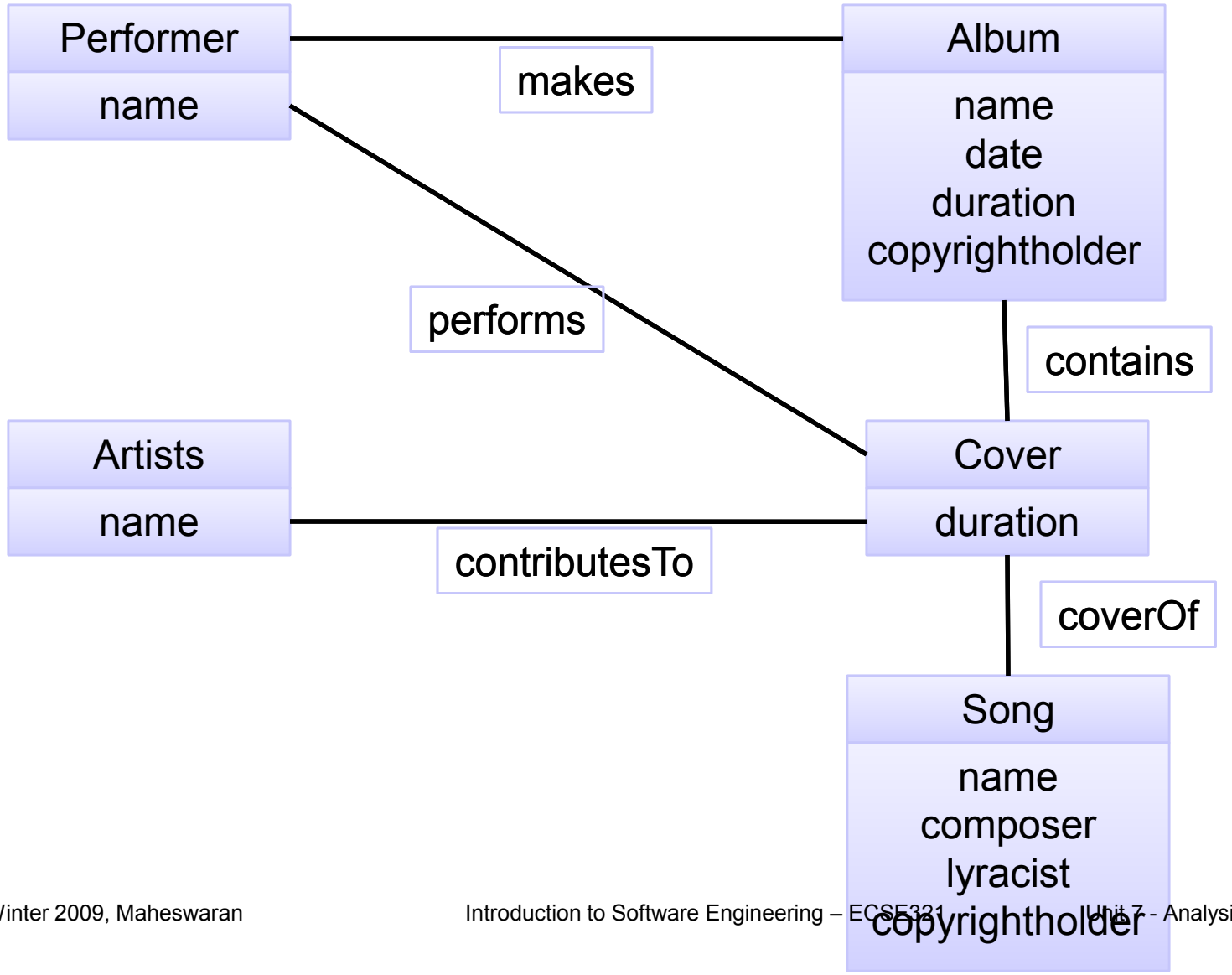
Classes and Objects

- An **object** is an entity that holds data and exhibits behavior.

- A **class** is an abstraction of a set of objects with common operations and attributes.

- An **attribute** is a data item held by an object or class.
- An **operation** is an object or class behavior.
- An **association** is a connection between classes representing a relation on the sets of instances of the connected classes.

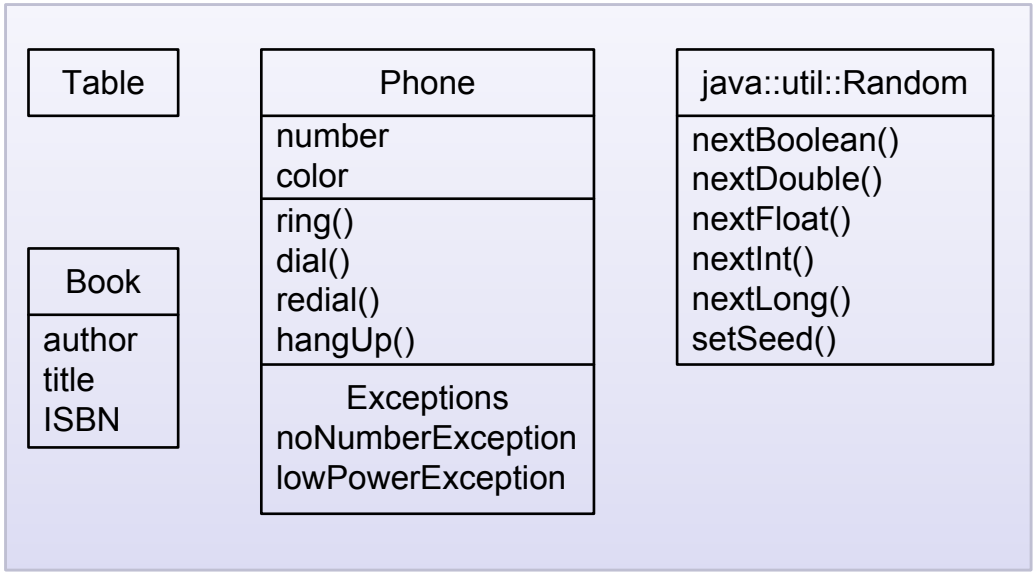
Example Class Diagram



A Brief UML Review - Names

- A **name** in UML is character string that identifies a model element.
  - Simple name: sequence of letters, digits, or punctuation characters
  - Composite name: sequence of simple names separated by the double colon (::)
- Examples
  - Java::util::Vector
  - veryLongNameWithoutPunctuationCharacters
  - short_name

UML Class Symbol



- Compartments
  - Class name
  - Attributes
  - Operations
  - Other compartments

Attribute Specification Format

name : *type* [*multiplicity*] = *initial-value*

- *name*—simple name, cannot be suppressed
- *type*—any string, may be suppressed along with the :
- *multiplicity*—number of values stored in attribute
  - list of ranges of the form *n*..*k*, such that *n* <=*k*
  - *k* may be *
  - *n*..*n* is the same as *n*
  - 0..* is the same as *
  - 1 by default
  - if suppressed, square brackets are omitted
- *initial-value*—any string, may be suppressed along with the =

Attribute Specification Example

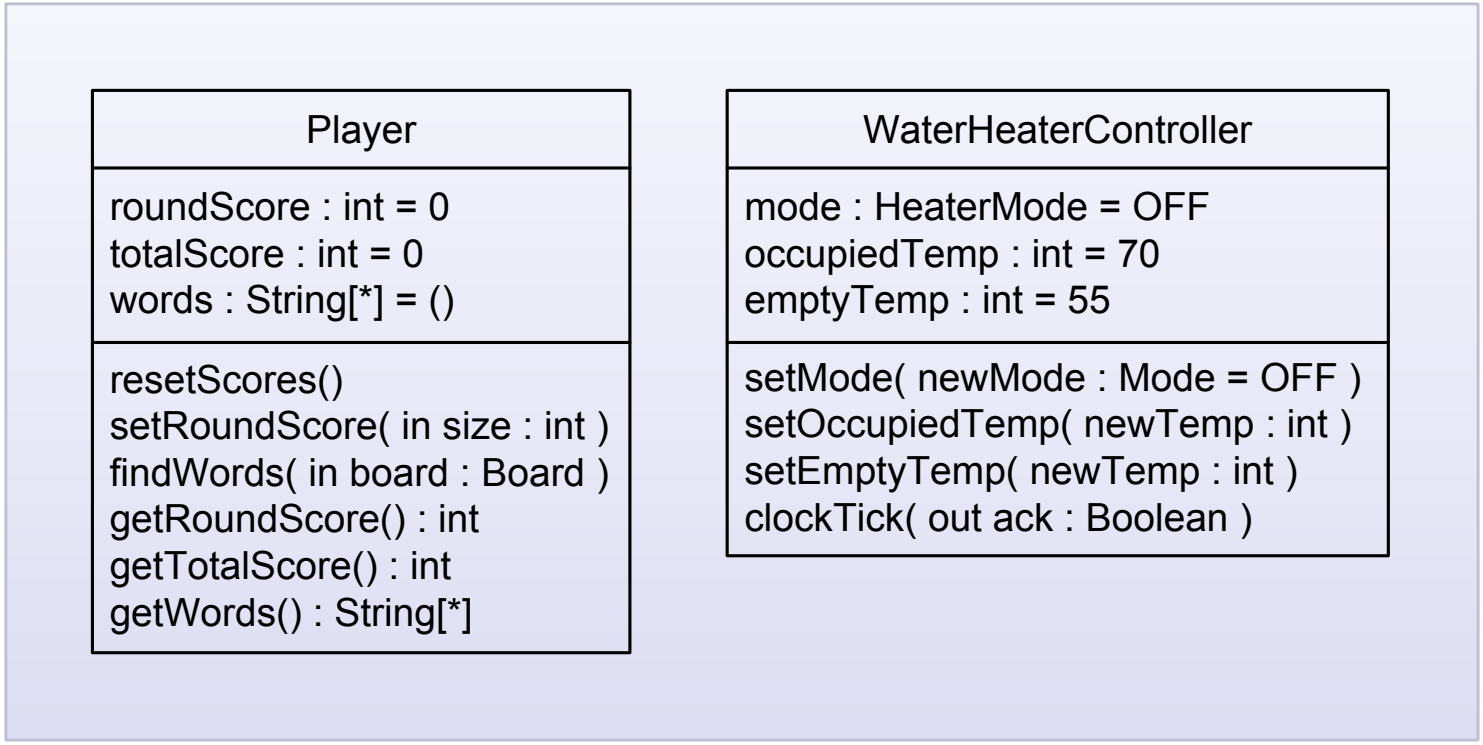
- weight: float – specifies weight as float
- toDo: string[1..10] – specifies a collection of 1 to 10 strings
- size: integer = 128 – specifies an integer holding a default value of 128

Operation Specification Format

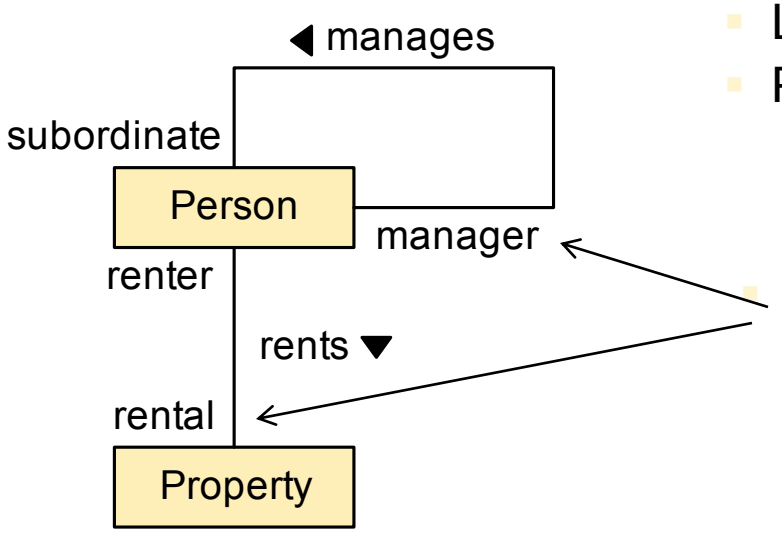
name(*parameter-list*) : *return-type-list*

- *name*—simple name, cannot be suppressed
- *parameter-list*
  - *direction param-name* : *param-type* = *default-value*
  - *direction*—in, out, inout, return; in when suppressed
  - *param-name*—simple name; cannot be suppressed
  - *param-type*—any string; cannot be suppressed
  - *default-value*—any string; if suppressed, so is =
- *return-type-list*—any comma-separated list of strings; if omitted (with :) indicates no return value
- The *parameter-list* and *return-type-list* may be suppressed together.

Attribute and Operation Examples



Association Lines

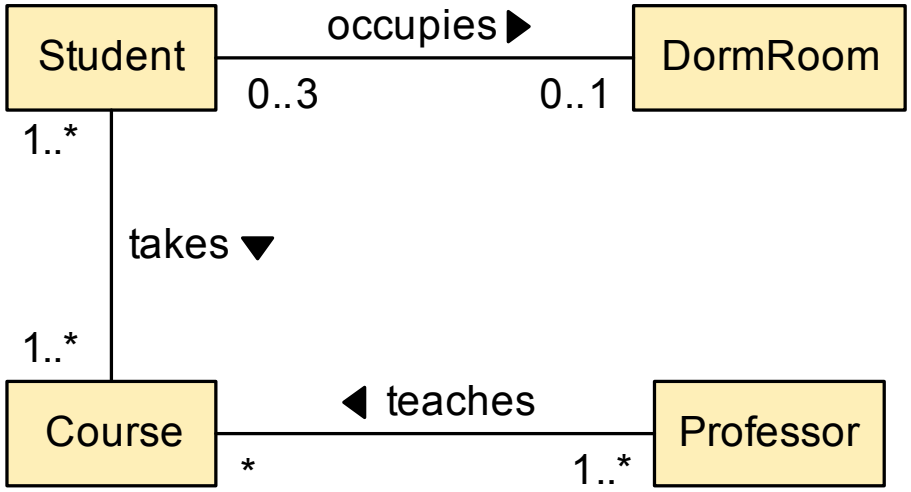


- Labeled or unlabeled lines
- Readable in two directions
- Rolenames

- Person (manager) manages another person (subordinate)
- Person (subordinate) is managed by another person (manager)
- Person (renter) rents property (rental)

Association Multiplicity

The multiplicity at the target class end of an association is the number of instances of the target class that can be associated with a single instance of the source class.



- A course is taught by one or more professors
- A professor teaches zero or more courses

Class Diagram Rules

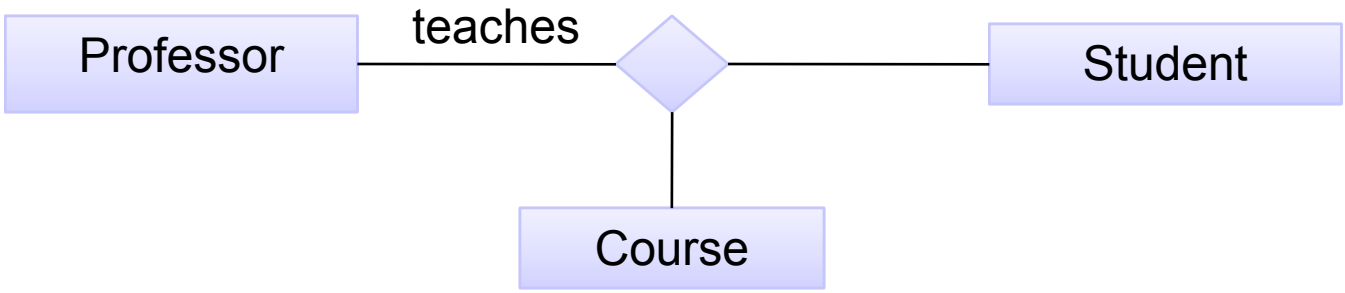
- Class diagrams must have a name compartment.
- Compartments must be in order.
- Attributes and operations must be listed one per line.
- Attribute and operation specifications must be syntactically correct.

Class Diagram Heuristics

- Name classes, attributes, and roles with noun phrases.
- Name operations and associations with verb phrases.
- Capitalize class names only.
- Center class and compartment names but left-justify other compartment contents.

Class Diagram Heuristics…

- Stick to binary associations.



- Prefer association names to rolenames.
- Place association names, rolenames and multiplicities on opposite sides of the line.

Class Diagram Uses

- Central static modeling tool in object-oriented design
  - Conceptual models
  - Design class diagrams
  - Implementation class diagrams
- Can be used throughout the design processes

Object Diagrams

- Object diagrams are used much less often than class diagrams.
- Object symbols have only two compartments:
  - Object name
  - Attributes (may be suppressed)

Object Name Format

object-name : *class-name*
[*stateList*]

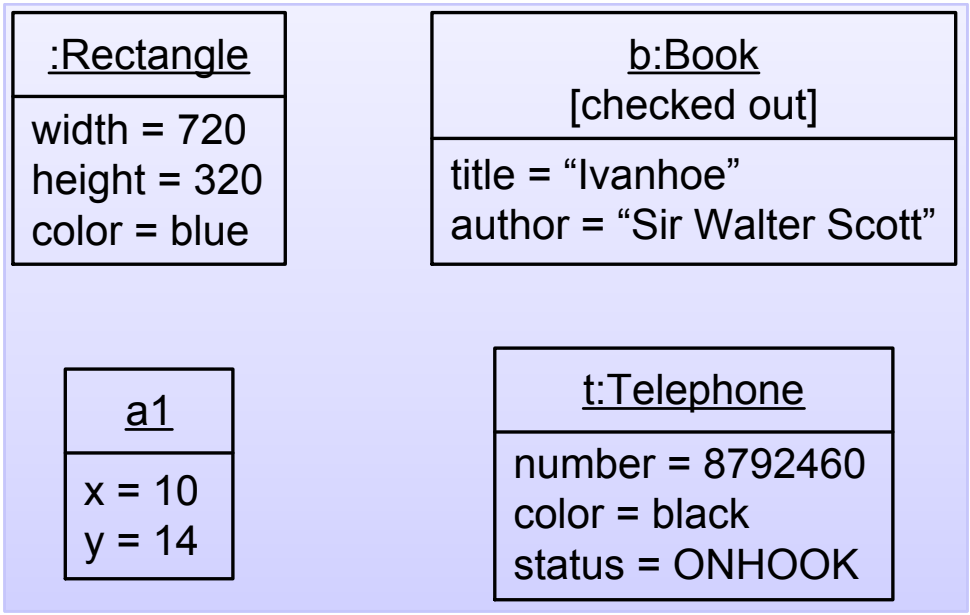
- *object-name*—simple name
- *class-name*—a name (simple or composite)
- *stateList*—list of strings; if suppressed, the square brackets are omitted
- The *object-name* and *class-name* may both be suppressed, but not simultaneously.

Object Attribute Format

attribute-name = *value*

- *attribute-name*—simple name
- *value*—any string
- Any attribute and its current value may be suppressed together.

Examples of Object Symbols



Object Links

- Show that particular objects participate in a relation between sets of objects
- Instances of associations
- Shown using a *link line*
  - Solid line (no arrowheads)
  - Underlined association name
- Link lines *never* have multiplicities

Object Diagram Uses

- Show the state of one or more objects at a moment during execution
- Dynamic (not full agreement) models as opposed to class diagrams, which are static models

Brief Recap

- Engineering design begins with analysis of the SRS and product design models.
- Analysis modeling helps designers understand the design problem.
- Class models include analysis (conceptual), design, and implementation class models.

Brief Recap…

- UML class diagrams can be used for all types of class models, and throughout the design process.
- UML object diagrams represent the state of objects during execution.

Conceptual Modeling

- What is conceptual modeling?
- What is the use of conceptual modeling?
- What is the process for conceptual modeling?
- Heuristics for conceptual modeling

Conceptual Models

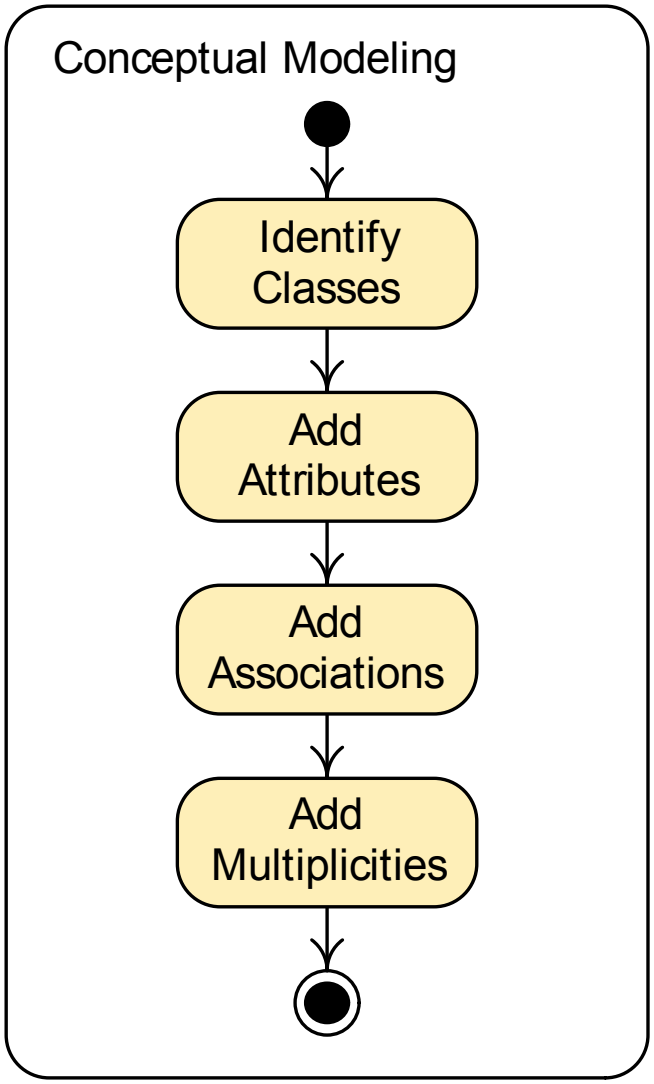
A **conceptual model** is a static model of the important entities in a **problem**, their responsibilities or attributes, the important relationships among them, and perhaps their behaviors.

Conceptual models are about real-world entities in the problem domain and not about software.

Uses of Conceptual Models

- In product design
  - Understanding the problem domain
  - Setting data requirements
  - Validating requirements
- In engineering design
  - Understanding a product design
  - Providing a basis for engineering design modeling

Conceptual Modeling Process



Caldera Example

Caldera is a smart water heater controller that attaches to the thermostat of a water heater. Caldera sets the water heater thermostat high when hot water is much in demand and sets it low when there is not much demand. For example, Caldera can be told to set the thermostat high on weekday mornings and evenings and all day on weekends, and low during the middle of weekdays and at night.

Furthermore, Caldera can be told to set the thermostat high all the time in case of illness or other need, or be told to set the thermostat low all the time in case of vacation or some other prolonged absence from home.

The homeowner can specify values for the following Caldera parameters:
Low temp – temperature when little or no water is needed.
High temp – temperature when much hot water is needed.
Weekend days – days when the thermostat will be set high; Peek times –one to three hour periods during which thermostat will be set high.
Mode – One of the Caldera states: Stay low mode – thermostat is set to low temp, Stay high mode – set to high, Normal mode – on a regular schedule

Identifying Classes—Brainstorming

- Study the product design (SRS, use case models, other models)
- Look for nouns and noun phrases for
  - Physical entities
  - Individuals, roles, groups, organizations
  - Real things managed, tracked, recorded, or represented in the product
  - People, devices, or systems that interact with the product (actors)

Identifying Classes-Rationalizing

- Remove noun phrases designating properties (they may be attributes).
- Remove noun phrases designating behaviors (they may be operations).
- Combine different names for the same thing.

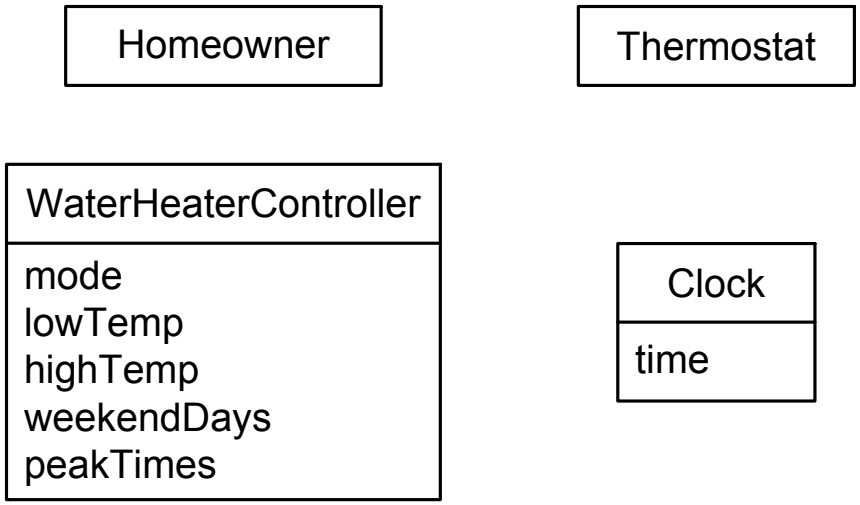
Identifying Classes-Rationalizing…

- Remove entities that do not directly interact with the product.
- Clarify vague nouns or noun phrases.
- Remove irrelevant or implementation entities.

Caldera Example – From Description

Noun Phrases	Comment
Water heater, controller, thermostat, homeowner, clock	concepts
Mode, Low Temp, High Temp, Weekend days, Peak times	attributes of water heater controller
Time	attribute of clock
Caldera, weekday, morning, evening, need, vacation, day, week, night, middle, illness, absence, house, parameter, value, one, three, schedule, second	Irrelevant noun phrases
Water heater, water temperature, hot water	Indirect connection to program

Caldera Example, Draft 1



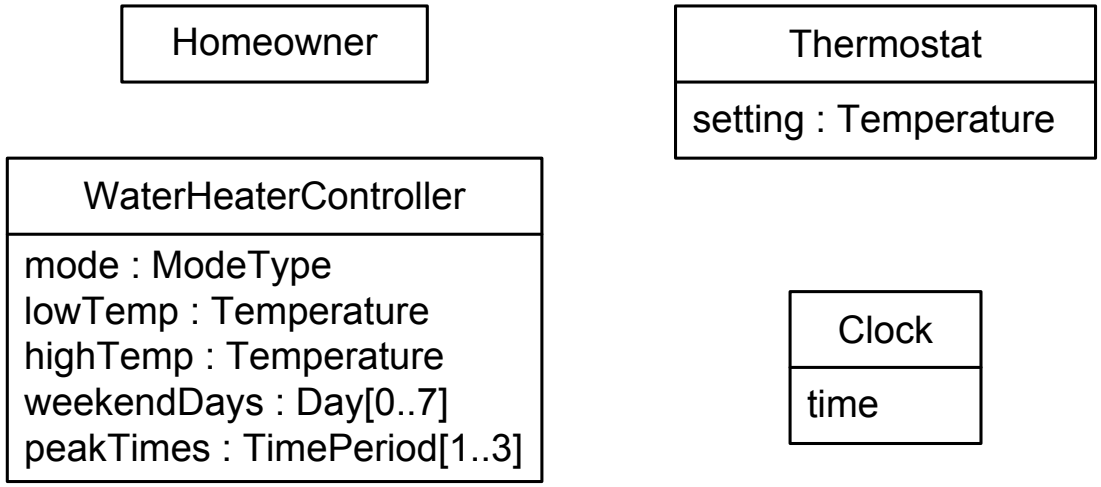
Adding Attributes 1

- Study the SRS and product design models looking for adjectives and other modifiers.
- Use names from the problem domain.
- Include only those types, multiplicities, and initial values specified in the problem.

Adding Attributes 2

- Don’t add object identifiers unless they are important in the problem.
- Don’t add implementation attributes.
- Add operations sparingly.

Caldera Example, Draft 2



Adding Associations - Brainstorming

- Study the SRS and product design models looking for verbs and prepositions describing relationships between model entities.
- Look for relationships such as
  - Physical or organizational proximity;
  - Control, coordination, or influence;
  - Creation, destruction, or modification;
  - Communication; and
  - Ownership or containment.

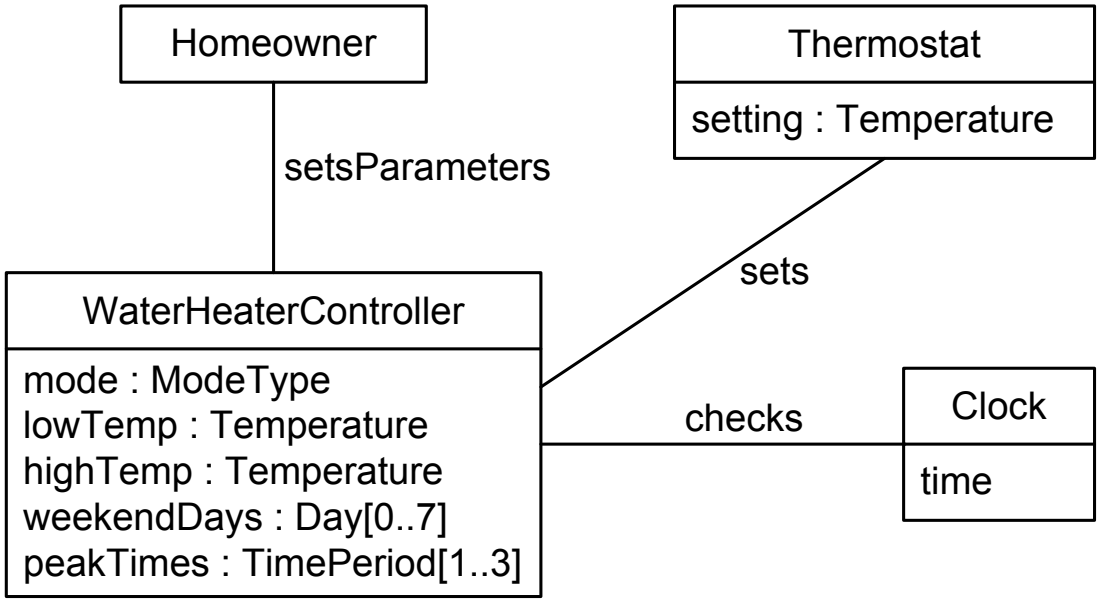
Adding Associations- Rationalizing

- Limit the number of associations to at most one between any pair of classes.
- Combine different names for the same association.
- Break associations among three or more classes into binary associations.

Adding Associations -Rationalizing

- Make association names descriptive and precise.
- Add rolenames where they are needed.

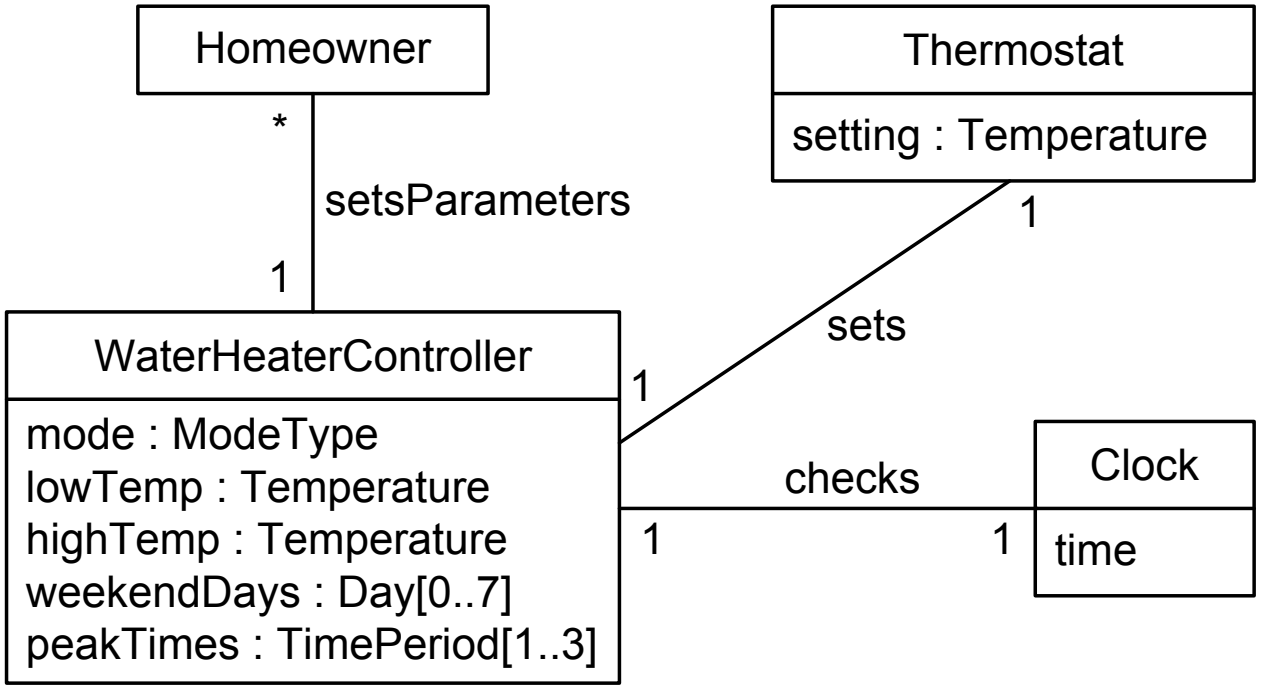
Caldera Example, Draft 3



Adding Multiplicities

- Take pairs of associated entities in turn.
  - Make one class the target, the other the source.
  - Determine how many instances of the target class can be related to a single instance of the source class.
  - Reverse the target and source and determine the other multiplicity.
- Consult the product design.
- Add only multiplicities important in the problem.

Caldera Example, Final Draft



Summary

- A conceptual model represents the important entities in a design problem along with their properties and relationships.
- Conceptual models represent the design *problem*, not the software *solution*.
- Conceptual models are useful throughout product design and in engineering design analysis.

Summary…

- There is a process for conceptual modeling.
- Process steps can be done by analyzing the text of product design artifacts.
- Several heuristics guide designers in conceptual modeling.

What is the “dynamic” model

- A model that captures behaviour
- Diagrams for dynamic modeling
  - **Interaction diagrams** describe the dynamic behavior between objects
  - **State charts** describe the dynamic behavior of a single object
- Interaction diagrams
  - **Sequence Diagram**:
    - Dynamic behavior of a set of objects arranged in time sequence
    - Good for real-time specifications and complex scenarios
- **State Charts**
  - A state machine that describes the response of an object of a given class to outside input (events)

Why do we need a dynamic model?

- Understand the flow of events
- Understand lifespan of classes
- Refine the static model
  - Add classes:
    - Events
    - Sequence diagrams are a source for more objects
- Identify operations for the object model

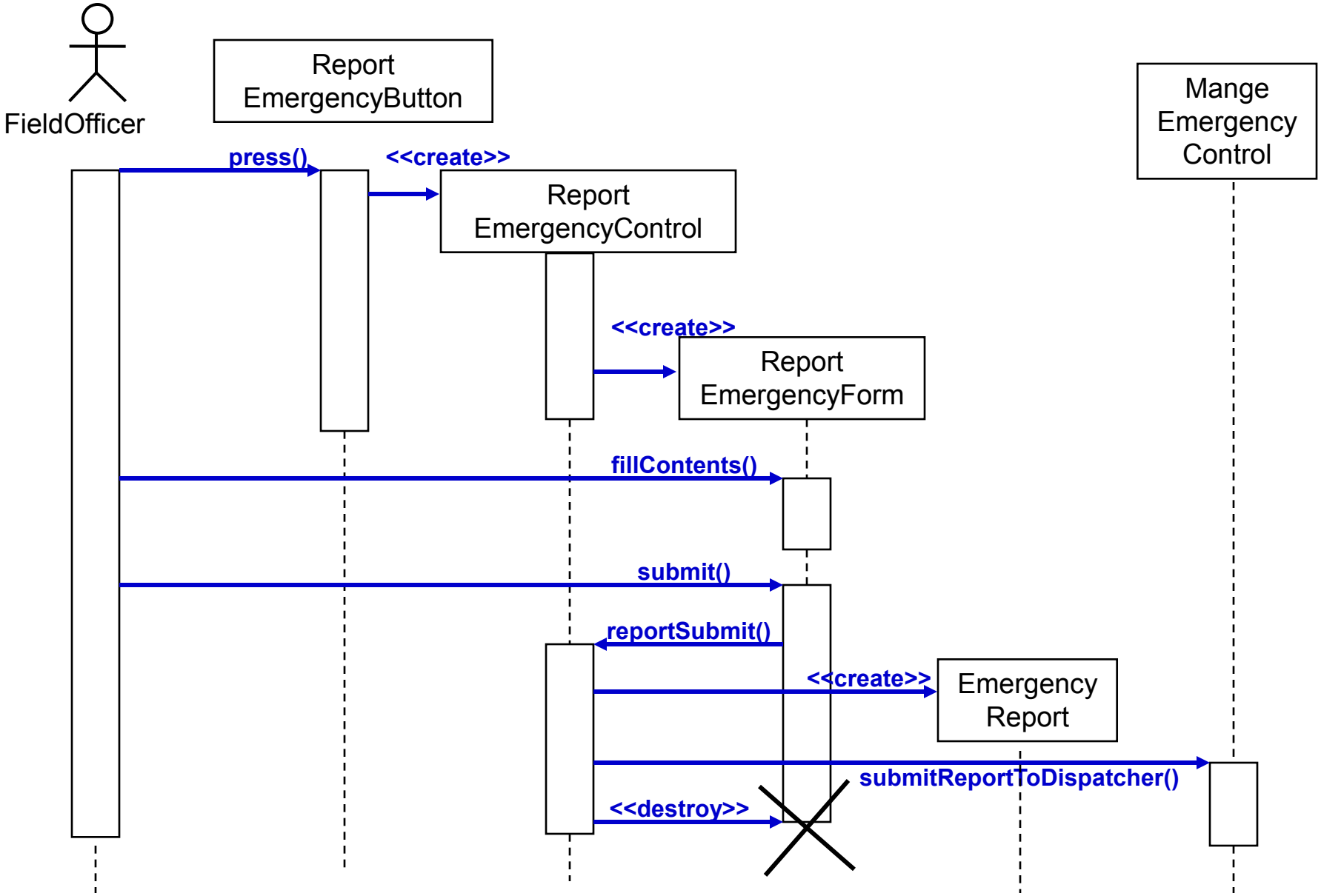
- Dynamic model will typically be incomprehensible to the customer!

How do we find a dynamic model?

- Start with a use case or scenario
- Model interaction between objects ➔ sequence diagram
- Model dynamic behavior of single objects ➔ state chart diagram
- Flow of events from “Dial a Number” Use case:
  - Caller lifts receiver
  - Dial tone begins
  - Caller dials
  - Phone rings
  - Callee answers phone
  - Ringing stops
  - ....

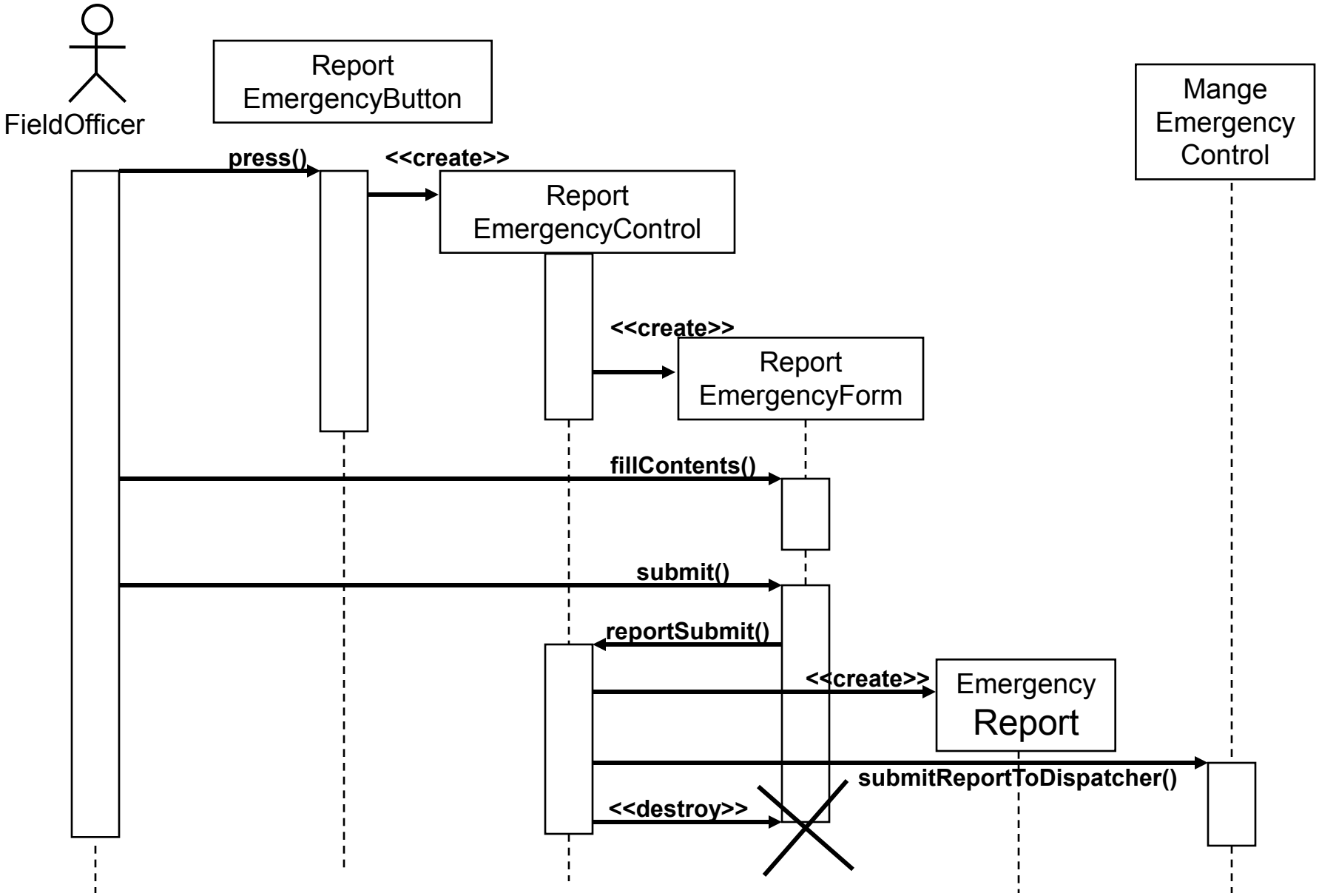
Sequence Diagrams

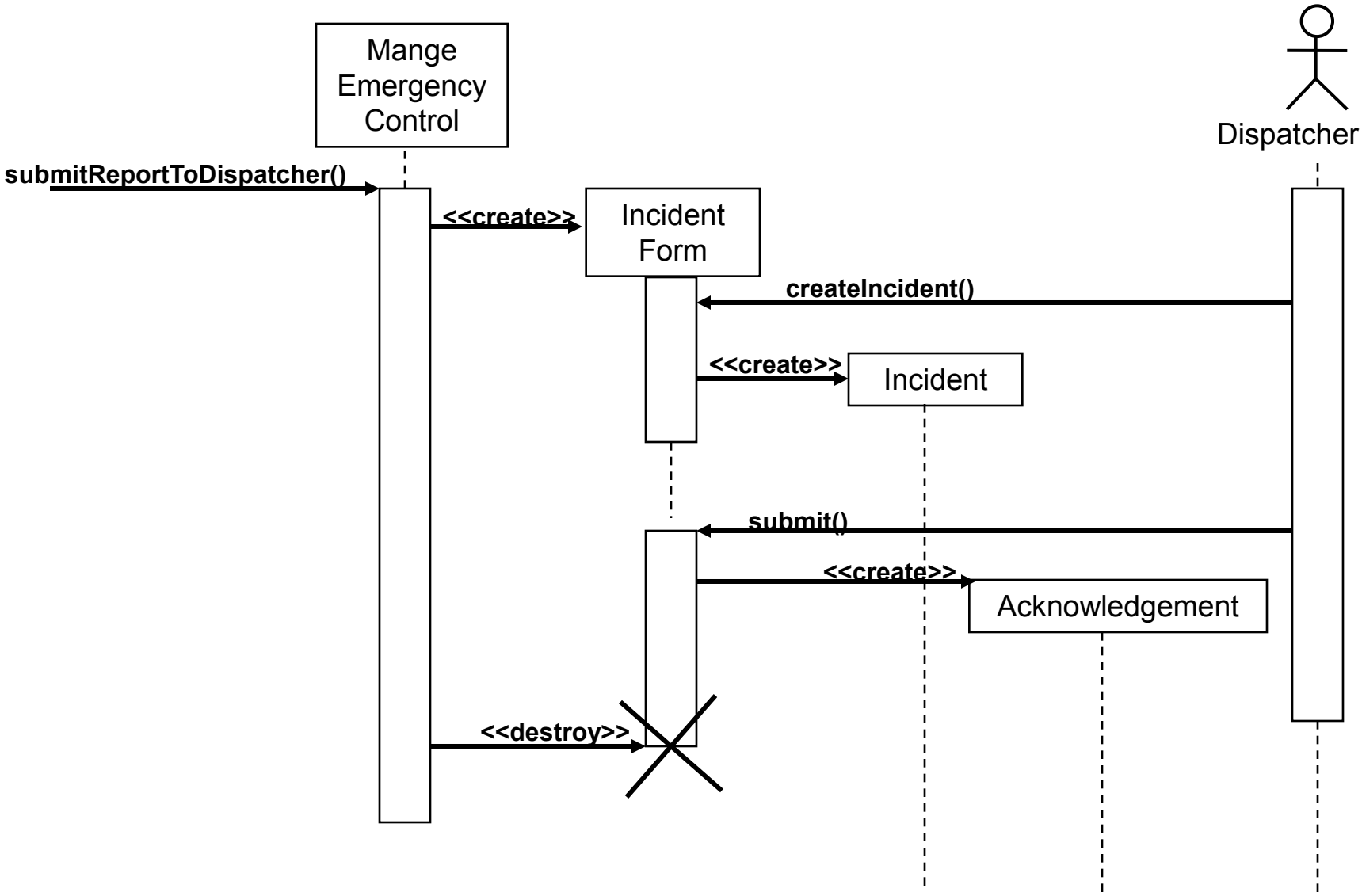
- From the flow of events in the use case or scenario proceed to the sequence diagram
- A sequence diagram is a graphical description of objects participating in a use case or scenario
- Relation to object identification:
  - Objects/classes have already been identified during object modeling
  - Objects are identified as a result of dynamic modeling
- Heuristic:
  - An event always has a sender and a receiver. Find them for each event ➔ These are the objects participating in the use case


FieldOfficer
Report EmergencyButton
Mange Emergency Control
press()
<<create>>
Report EmergencyControl
<<create>>
Report EmergencyForm
fillContents()
submit()
reportSubmit()
<<create>>
Emergency Report
submitReportToDispatcher()
<<destroy>>

Notations

1. Columns = objects
   1. Leftmost column = initiator
2. Horizontal arrows = messages/stimuli
   1. Write name of method/operation on arrow
   2. Have origin and destination
3. Time progresses vertically (top to bottom)
4. Activation period = vertical rectangle
   1. Dotted line – alive but inactive
5. <<create>> = objects that are created in the sequence
6. X = objects that are destroyed
7. Be precise!


FieldOfficer
Report EmergencyButton
Mange Emergency Control
press()
<<create>>
Report EmergencyControl
<<create>>
Report EmergencyForm
fillContents()
submit()
reportSubmit()
<<create>>
Emergency Report
submitReportToDispatcher()
<<destroy>>


Mange Emergency Control
Dispatcher
submitReportToDispatcher()
<<create>>
Incident Form
createIncident()
<<create>>
Incident
submit()
<<create>>
Acknowledgement
<<destroy>>

Sequence diagrams

- Can't build all sequence diagrams
  - Focus on typical ➔ exceptional ➔ bizarre
  - Excellent sanity check
  - Avoid implementation issues
- Heuristic for drawing:
  - First column – actor initiating use case
  - Second column – boundary object
  - Third column – control object
  - Control objects are created by initiating boundary
  - Boundary objects may be created by control
  - Entity objects are accessed by control & boundary
  - Entity objects **NEVER** access control or boundary