Modern methods in Software Engineering

UML

Introduction Content

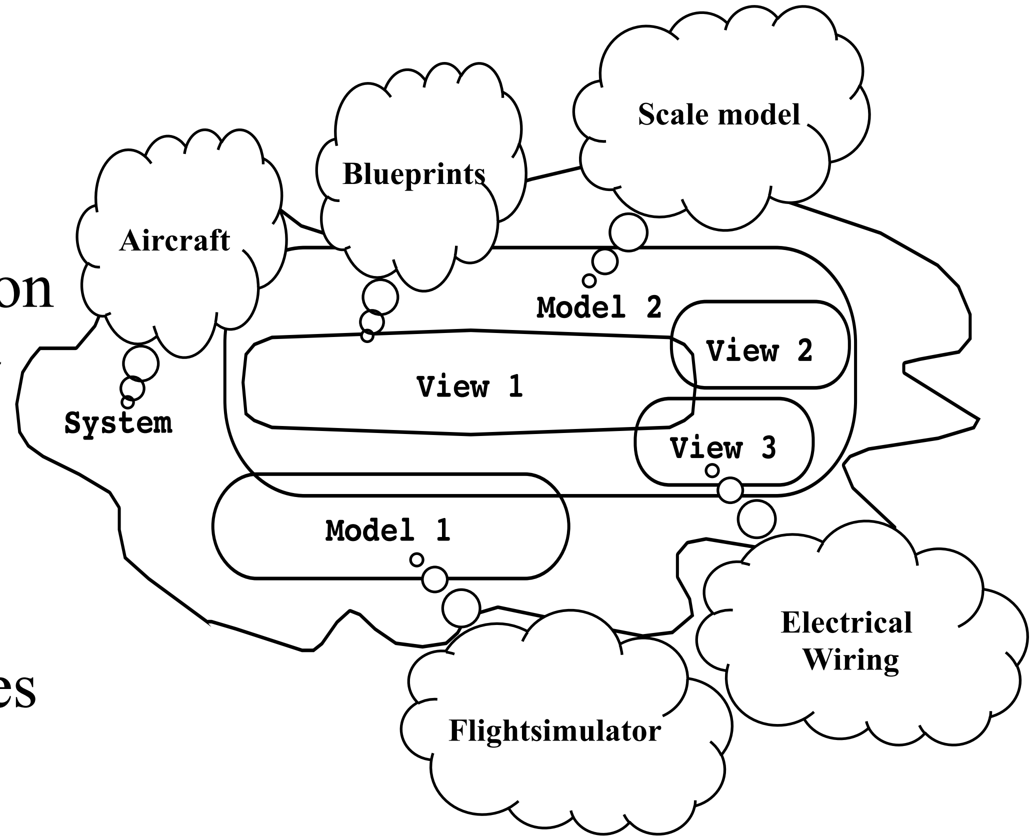
- Software modeling (concepts and phenomena)
- OO modeling
- UML
  - Use case Diagrams
  - Organizational diagrams
  - Class diagrams
  - Sequence diagrams
  - Communication diagrams
  - Statechart diagrams
  - Activity Diagrams
  - Deployment diagrams

Literature used

- Chapter 2 in the text-book

Software modeling

- What is modeling?
- Why model software?
- System, Notation, Models and Views
  - A ***model*** is an abstraction describing a subset of a system
  - A ***view*** depicts selected aspects of a model
  - A ***notation*** is a set of graphical or textual rules for depicting views
  - Views and models of a single system may overlap each other



Concepts and Phenomena

Phenomenon
- An object in the world of a domain as you perceive it
- *Example:* The lecture you are attending

Concept
- Describes the properties of phenomena that are common.
- *Example:* Lectures in software engineering course

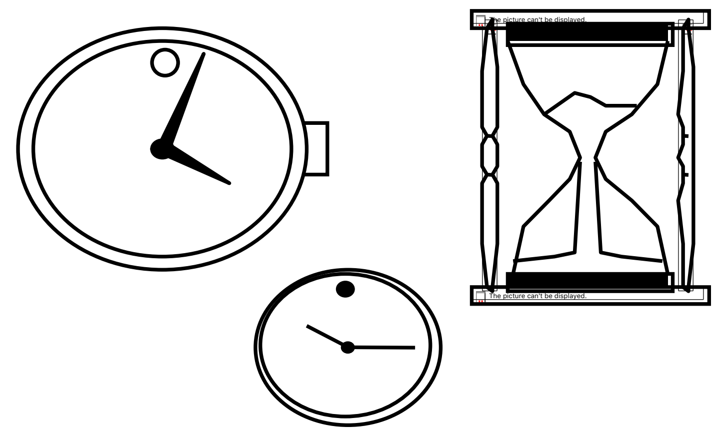
Concept is a 3-tuple:
- Name (To distinguish it from other concepts)
- Purpose (Properties that determine if a phenomenon is a member of a concept)
- Members (The set of phenomena which are part of the concept)

Concepts and phenomena

Name | **Purpose** | **Members**

Clock

A device that measures time.



- Abstraction
  - Classification of phenomena into concepts
- Modeling
  - Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Concepts in software

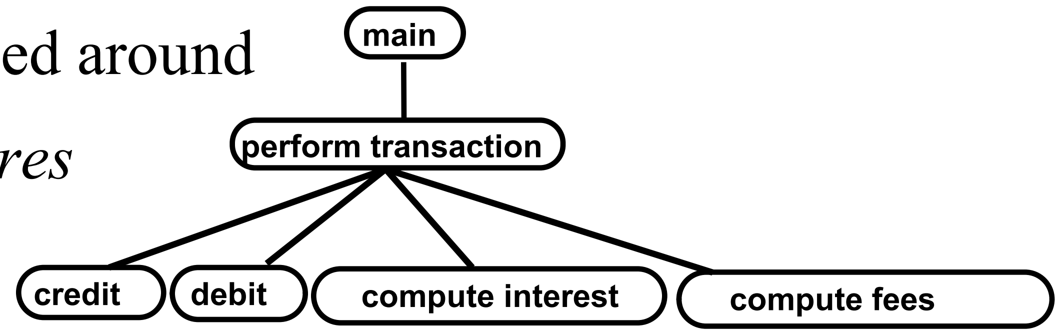
- Type:
  - An abstraction in the context of programming languages
    - Name: int,
    - Purpose: integer number,
    - Members: `0, -1, 1, 2, -2, . . .`
- Instance:
  - Member of a specific type
- The type of a variable represents all possible instances the variable can take

The following relationships are similar:

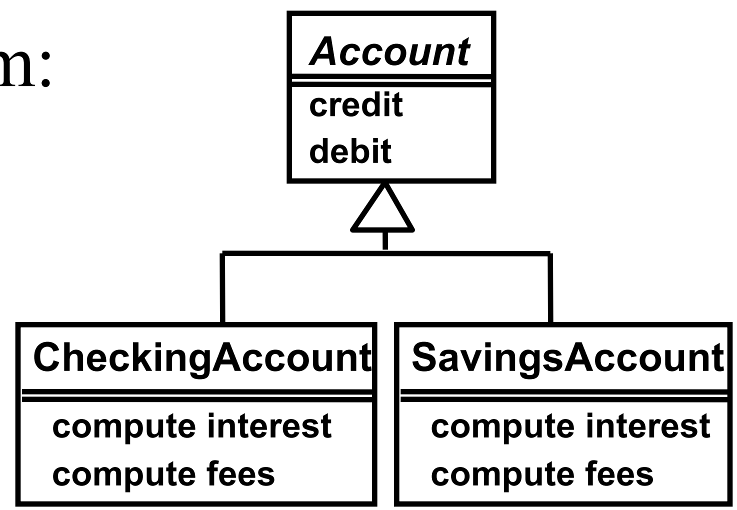
- "type" <–> "instance"
- "concept" <–> "phenomenon"

Software modeling paradigms

- Procedural paradigm:
  - Software is organized around the notion of *procedures*



- Object oriented paradigm:
  - Organizing procedural abstractions in the context of data abstractions



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

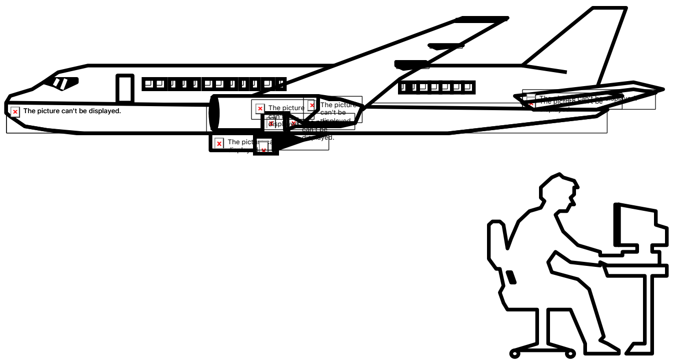
Objects, Classes, Inheritance

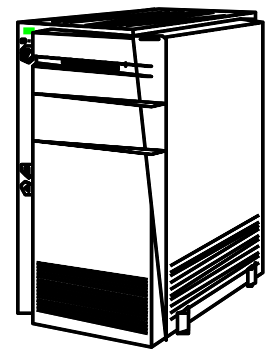
- Object
  - combines features of data and procedure
  - Has *properties* - Represent its state
  - Has *behavior* – set of methods how the object can be manipulated
- Class
  - A template from which instances of the class (objects) can be created
  - Carrier of properties common to the objects of the class
- Superclasses
  - Contain features common to a set of subclasses
- Inheritance hierarchies
  - Show the relationships among superclasses and subclasses
- Inheritance
  - The *implicit* possession by all subclasses of features defined in its superclasses

Abstract Data Types and Classes

- Abstract data type
  - Special type whose implementation is hidden from the rest of the system.
- Class:
  - An abstraction in the context of object-oriented languages
- Like an abstract data type, a class encapsulates both state (variables) and behavior (methods)
- Unlike abstract data types, classes can be defined in terms of other classes using inheritance
- Unlike of classes, abstract data types may have axioms/constraints on the values of their members
- Abstract data type is rather mathematical notion while class is a programming notion

Object-Oriented Modeling



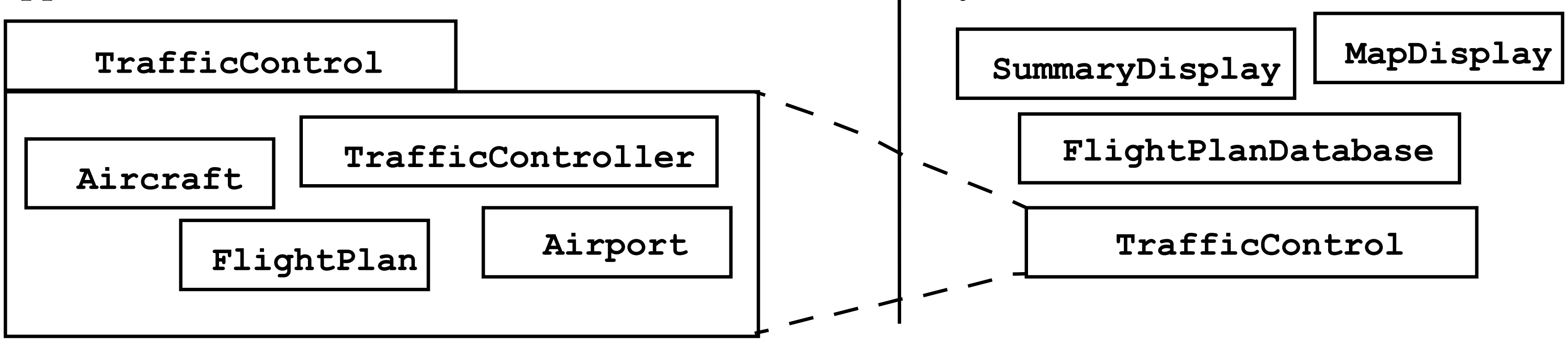


Application Domain

Solution Domain

Application Domain Model

System Model



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

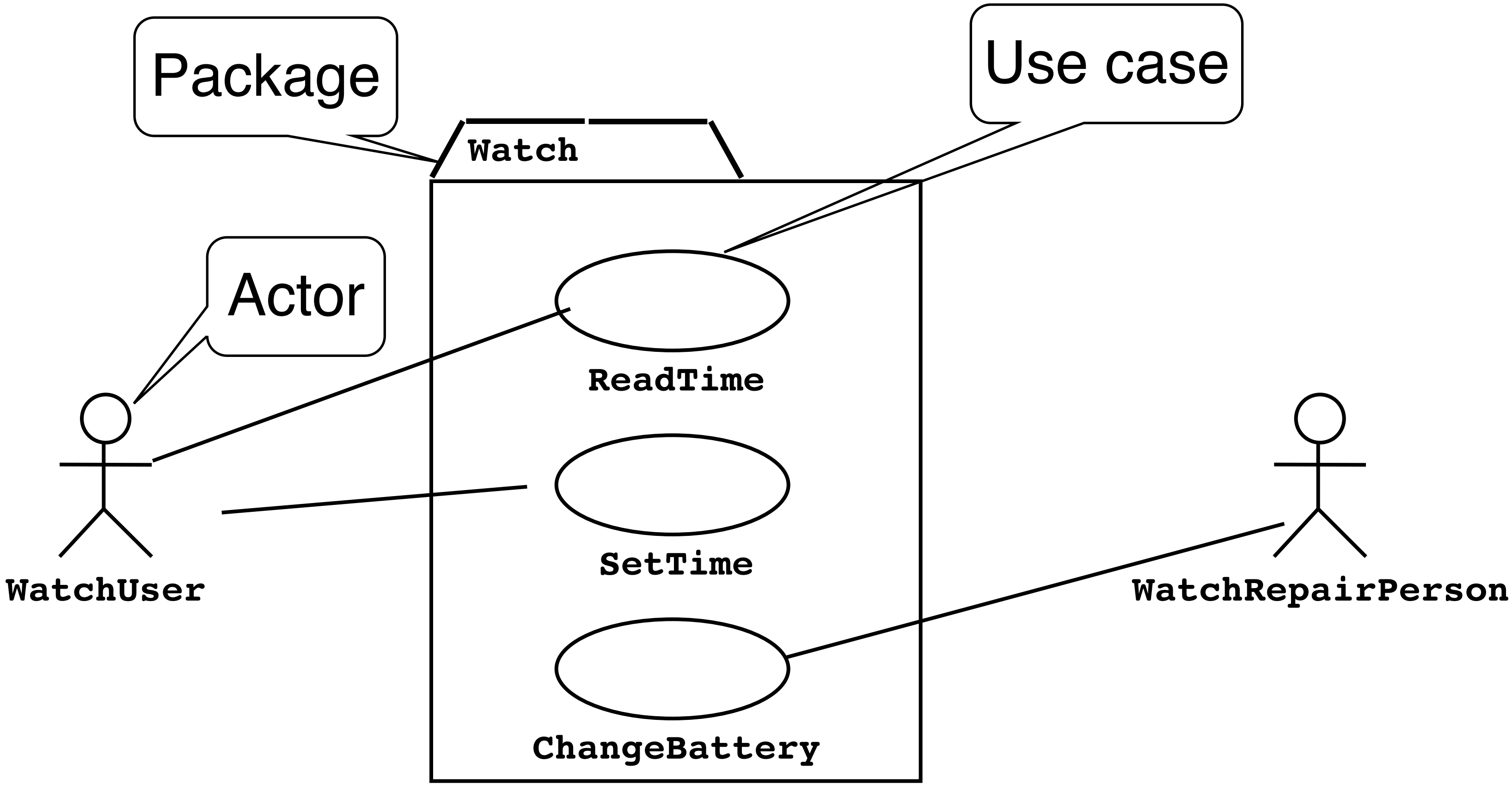
UML

- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: http://www.uml.org/
  - Quick reference: http://www.holub.com/goodies/uml/
- Supported by several CASE tools
  - IBM Rational ROSE
  - …

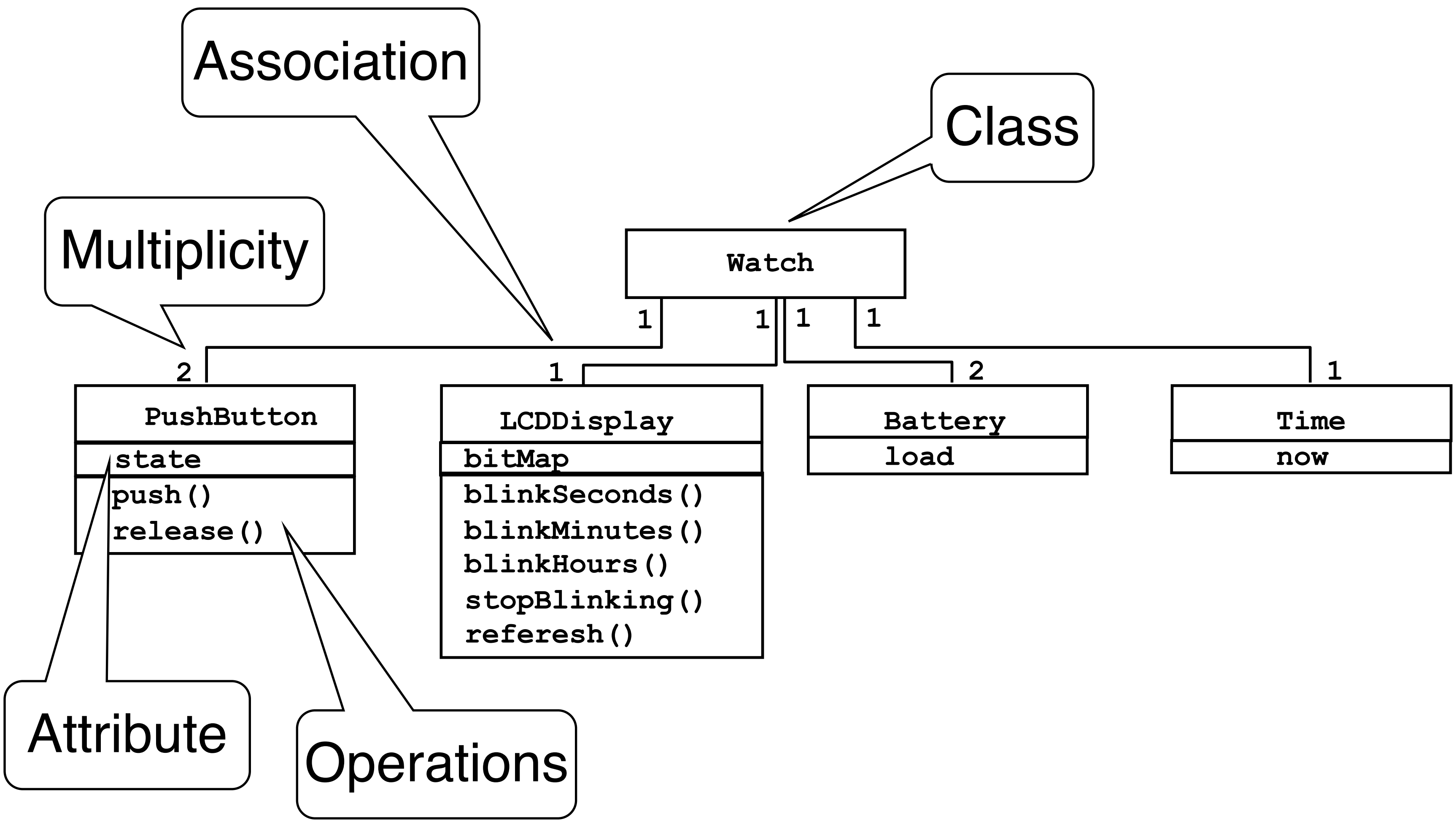
UML: Brief overview

- Use case Diagrams
  - Describe the functional behavior of the system as seen by the user.
- Organizational diagrams
  - Packaging and components
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, Associations
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects of the system
- Communication diagrams
  - Alternative presentation of sequence diagrams
- State diagrams
  - Describe the dynamic behavior of an individual object (essentially a finite state automaton)
- Activity Diagrams
  - Model the dynamic behavior of a system, in particular the workflow (essentially a flowchart)
- Component diagrams
  - Show structure of components
- Deployment diagrams
  - Used to model the hardware that will be used to implement the system

Use Case Diagrams

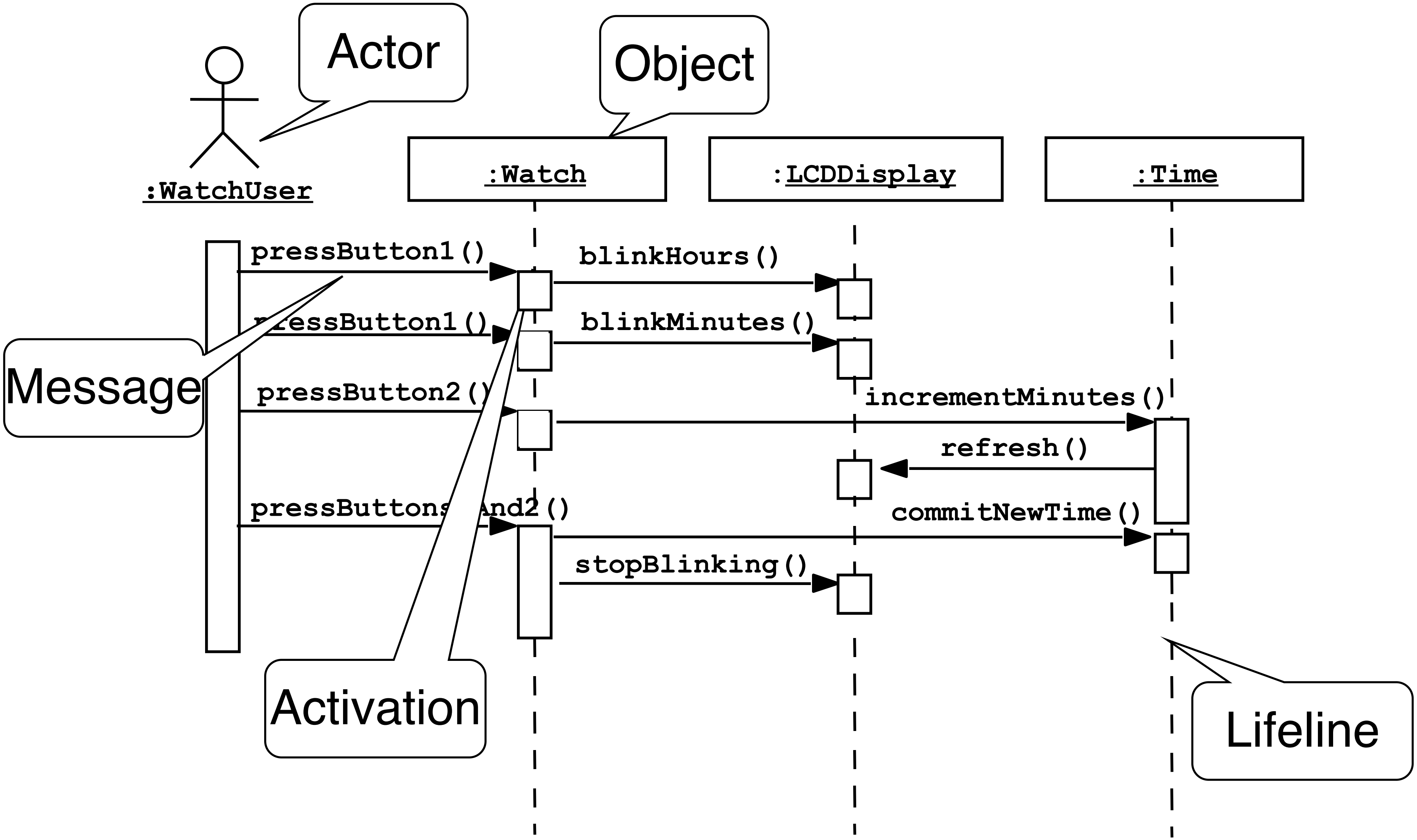


Class diagrams



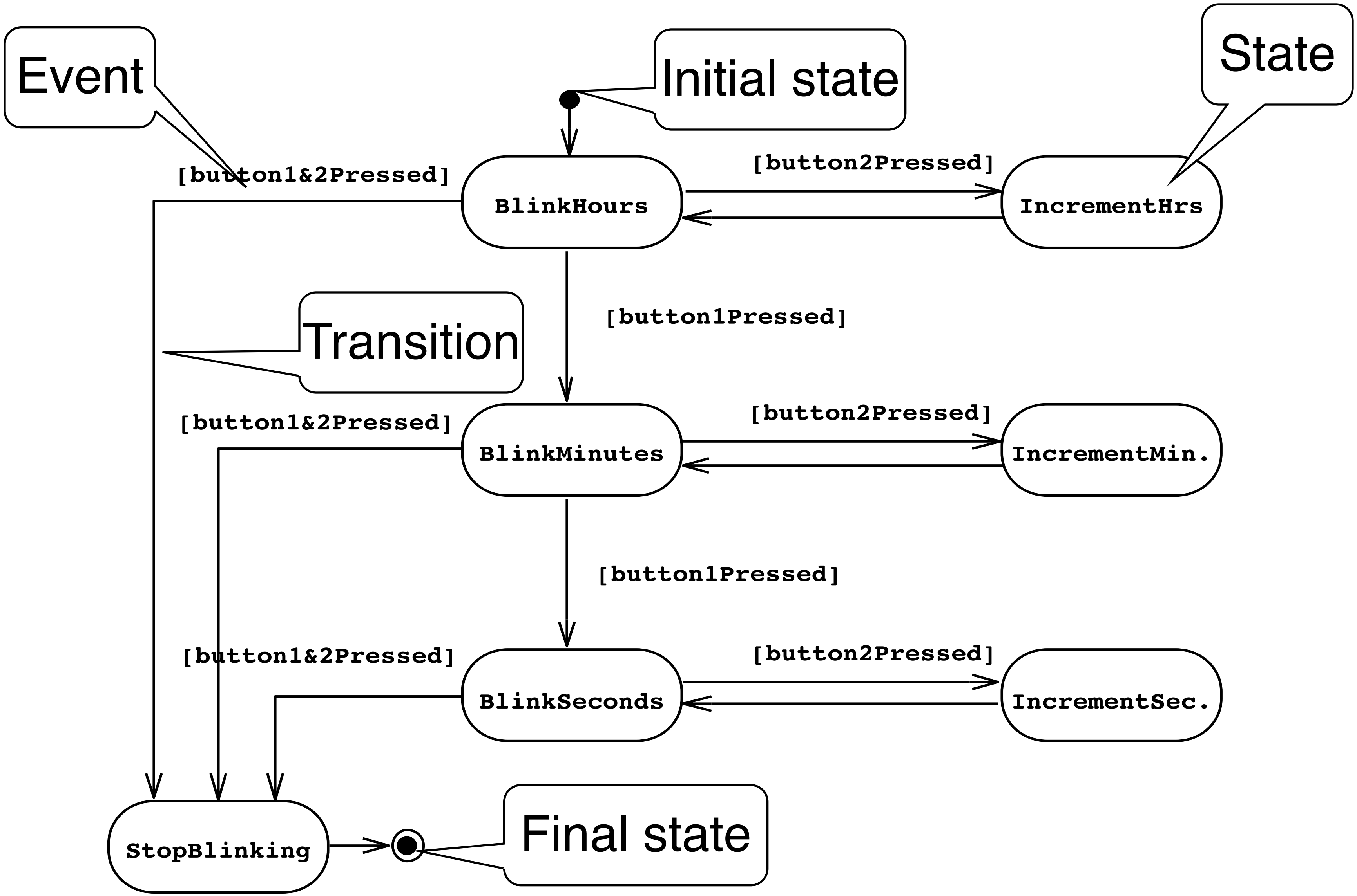
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Sequence diagrams



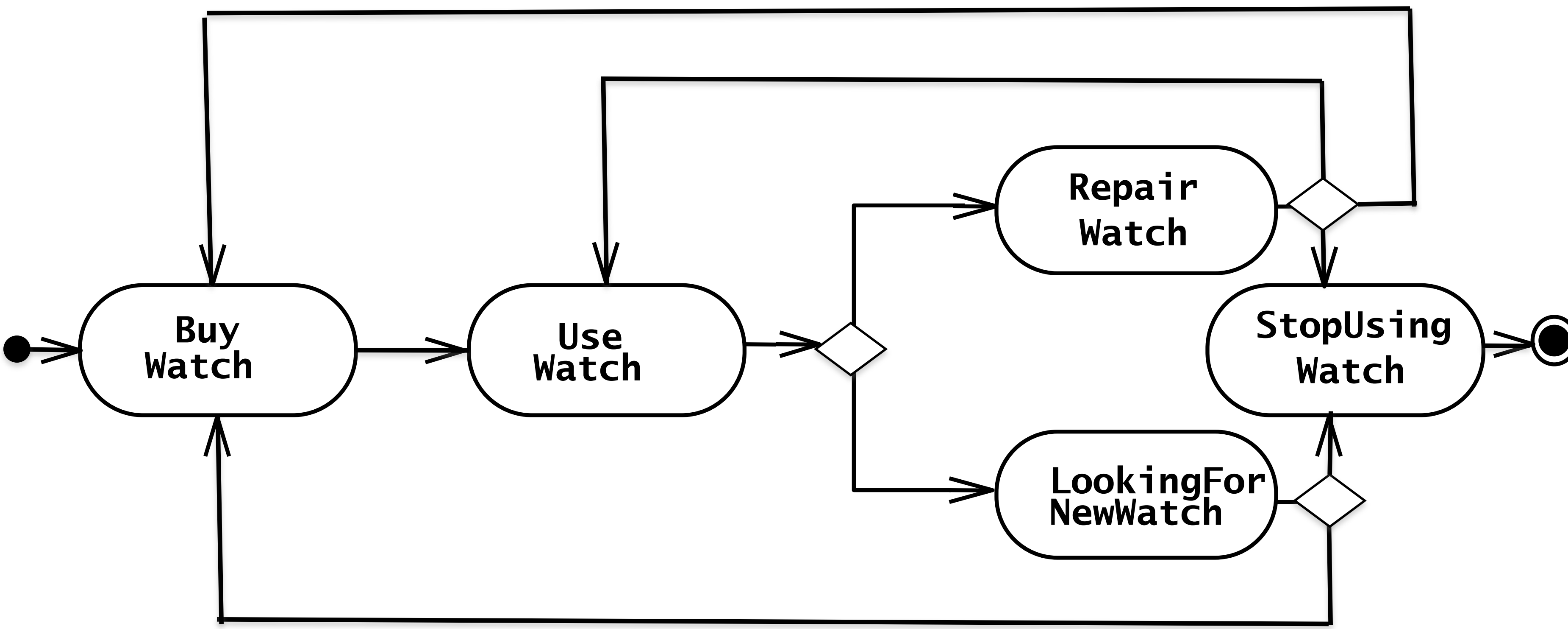
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Statechart Diagrams



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Activity diagrams



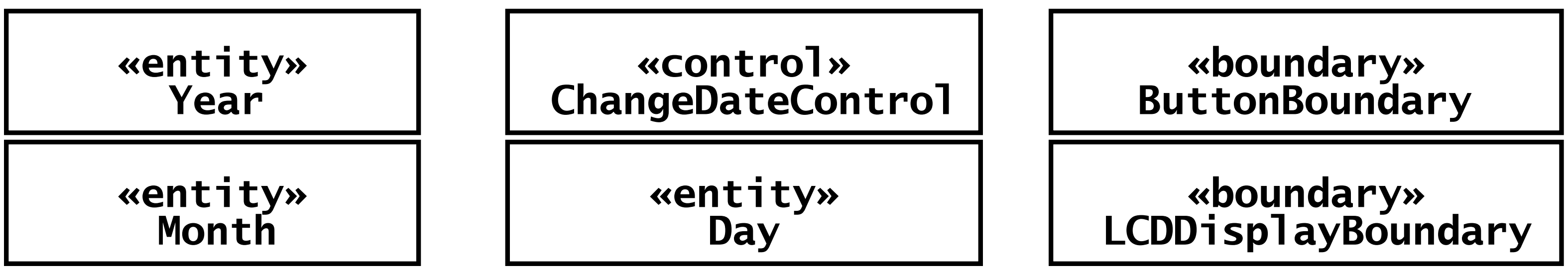
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

UML conventions

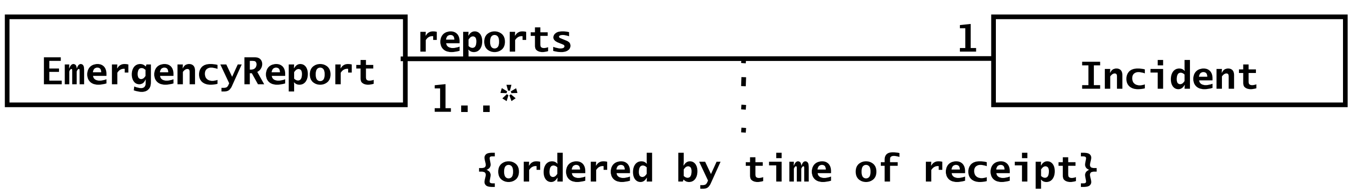
- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - <u>**`myWatch:SimpleWatch`**</u>
  - <u>**`:Firefighter`**</u>
- Types/concepts are denoted with non underlined names
  - **`SimpleWatch`**
  - **`Firefighter`**
- Diagrams are graphs
  - Nodes are entities
  - Arcs are relationships between entities

Extensions

- Stereotypes



- Constraint



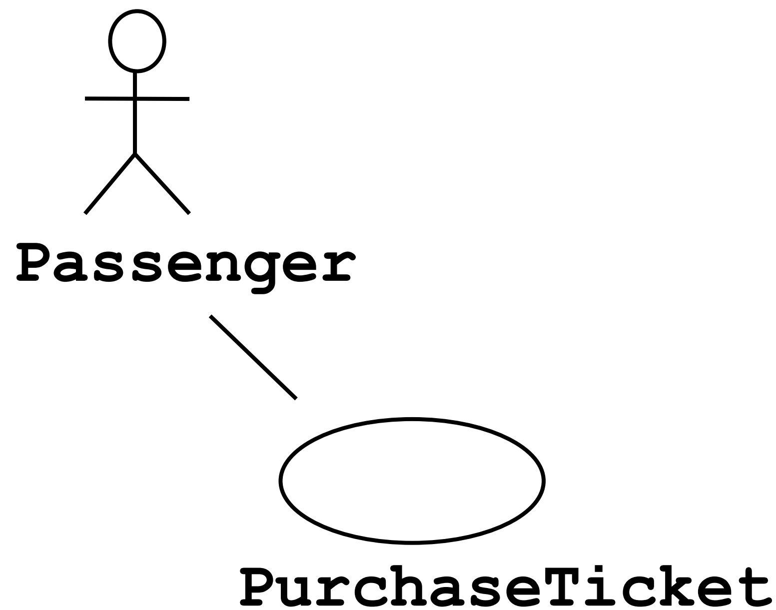
UML Diagrams

- Use case Diagrams
- Organizational diagrams
- Class diagrams
- Sequence diagrams
- Communication diagrams
- Statechart diagrams
- Activity Diagrams
- Component and Deployment diagrams

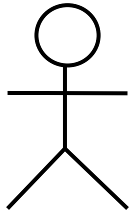
Use Case Diagrams (more details)

- Used during requirements elicitation to represent external behavior

- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

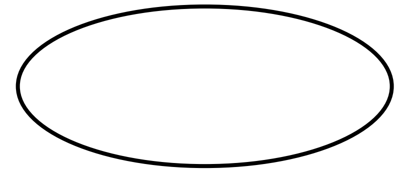


Actors and Use cases



Passenger

- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description.
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates



PurchaseTicket

A use case represents a class of functionality provided by the system as an event flow.

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case description

Name:

`Purchase ticket`

Participating actor(s):

`Passenger`

Entry condition:

- **`Passenger`** standing in front of ticket distributor.
- **`Passenger`** has sufficient money to purchase ticket.

Exit condition:

- **`Passenger`** has ticket.

Quality conditions:

- Time constraints

Event flow:

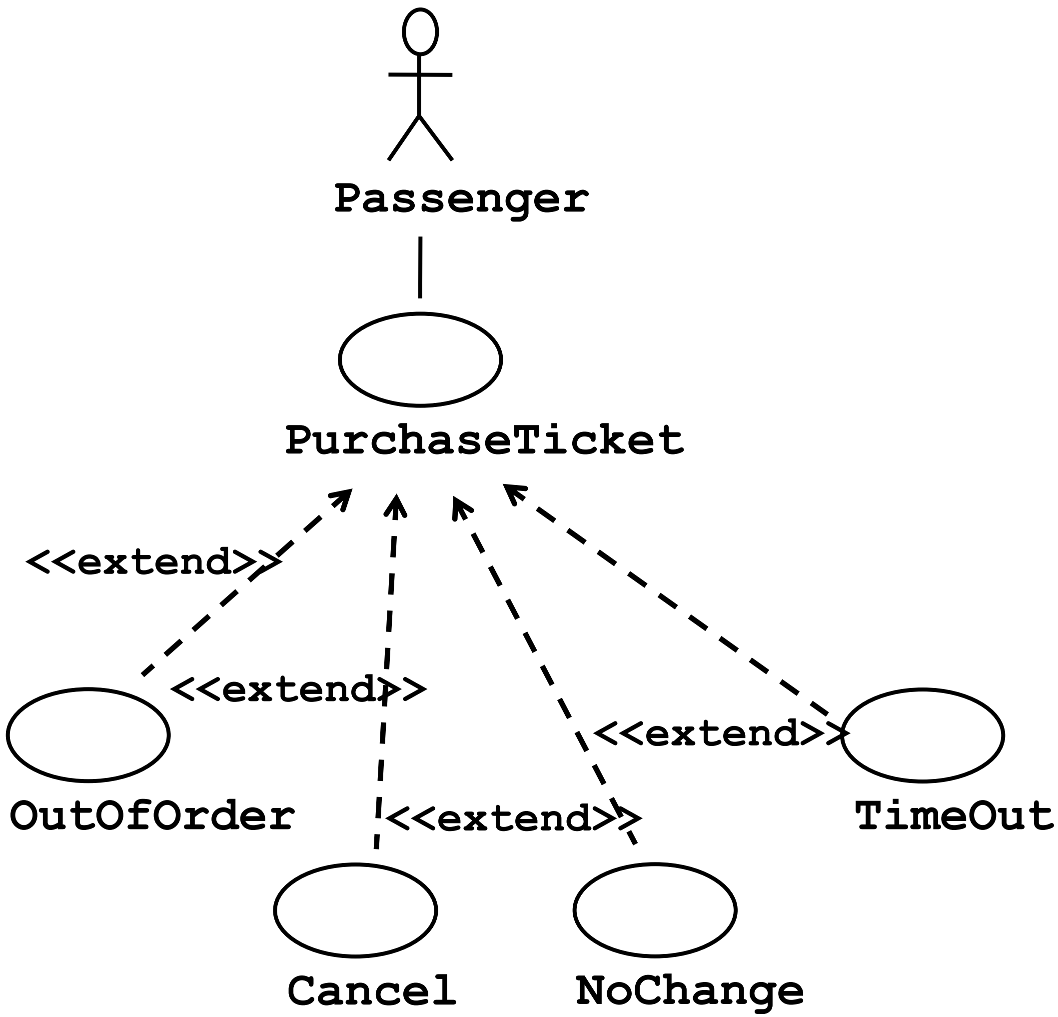
1. **`Passenger`** selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. **`Passenger`** inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

Anything missing?

Exceptional cases!

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

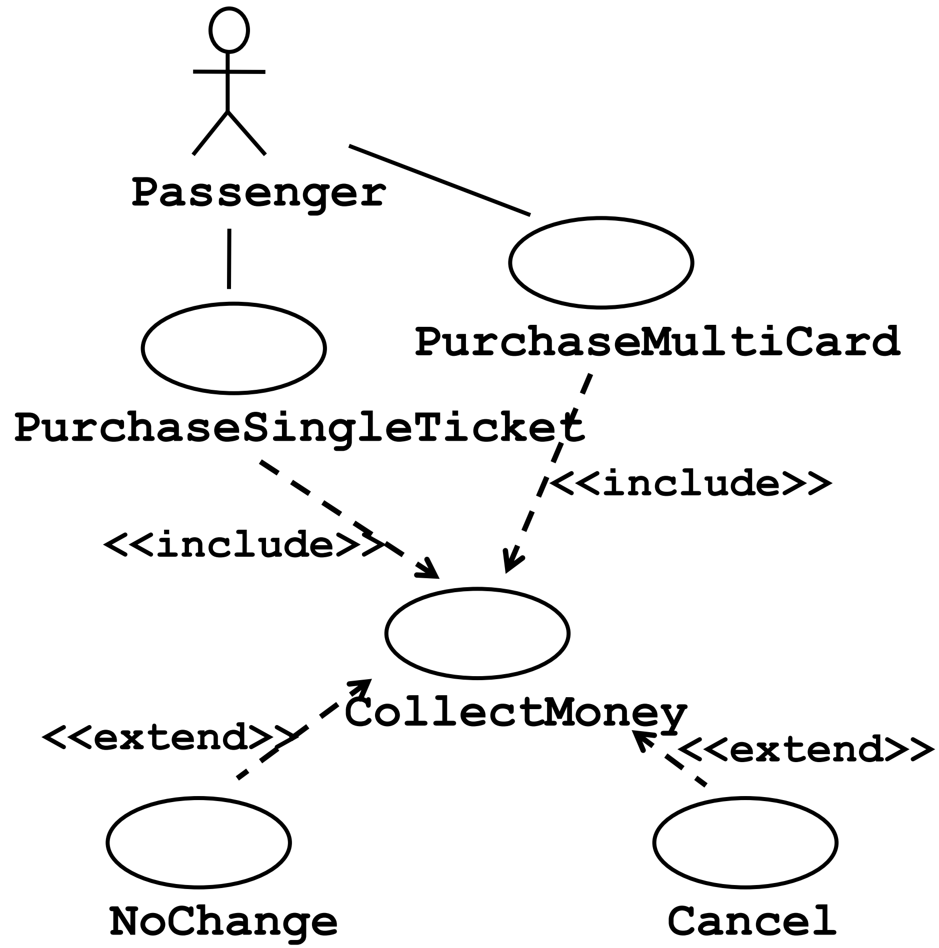
The <<extend>> Relationship



- **<<extend>>** relationships typically represent exceptional or seldom invoked cases.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a **<<extend>>** relationship is to the extended use case

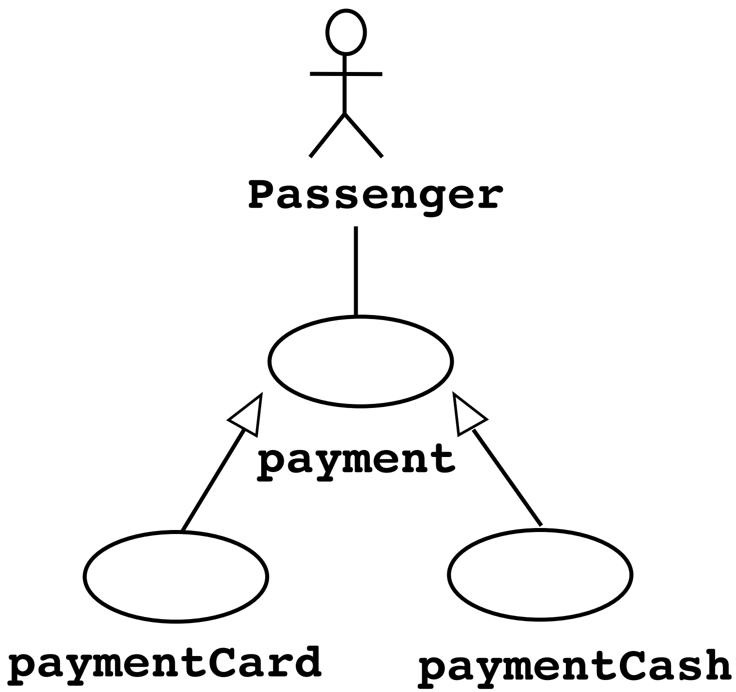
The <<**`include`**>> relationship

- **`<<include>>`** relationship represents behavior that is factored out of the use case.
- **`<<include>>`** behavior is factored out for reuse, not because it is an exception.
- The direction of a **`<<include>>`** relationship is to the using use case (unlike **`<<extend>>`** relationships).

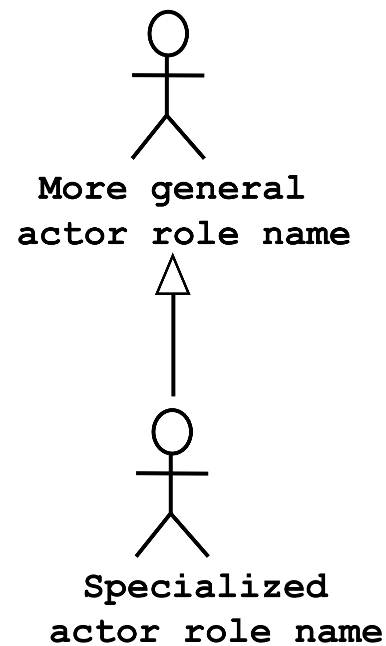


Generalization (Inheritance relationships)

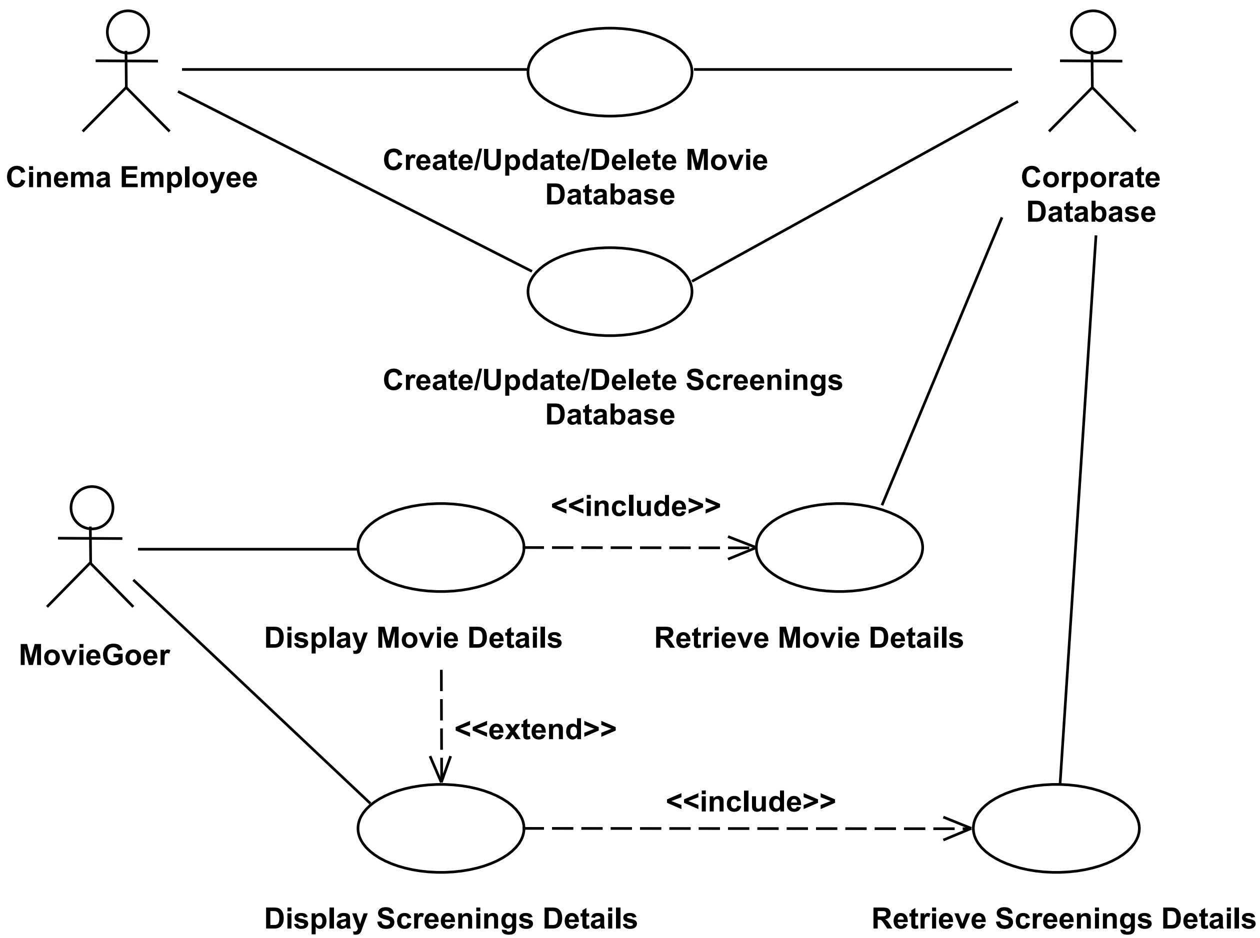
- Generalization between Use Cases



- Generalization between Actors



Another Use Case Diagram



Adopted from Maciaszek

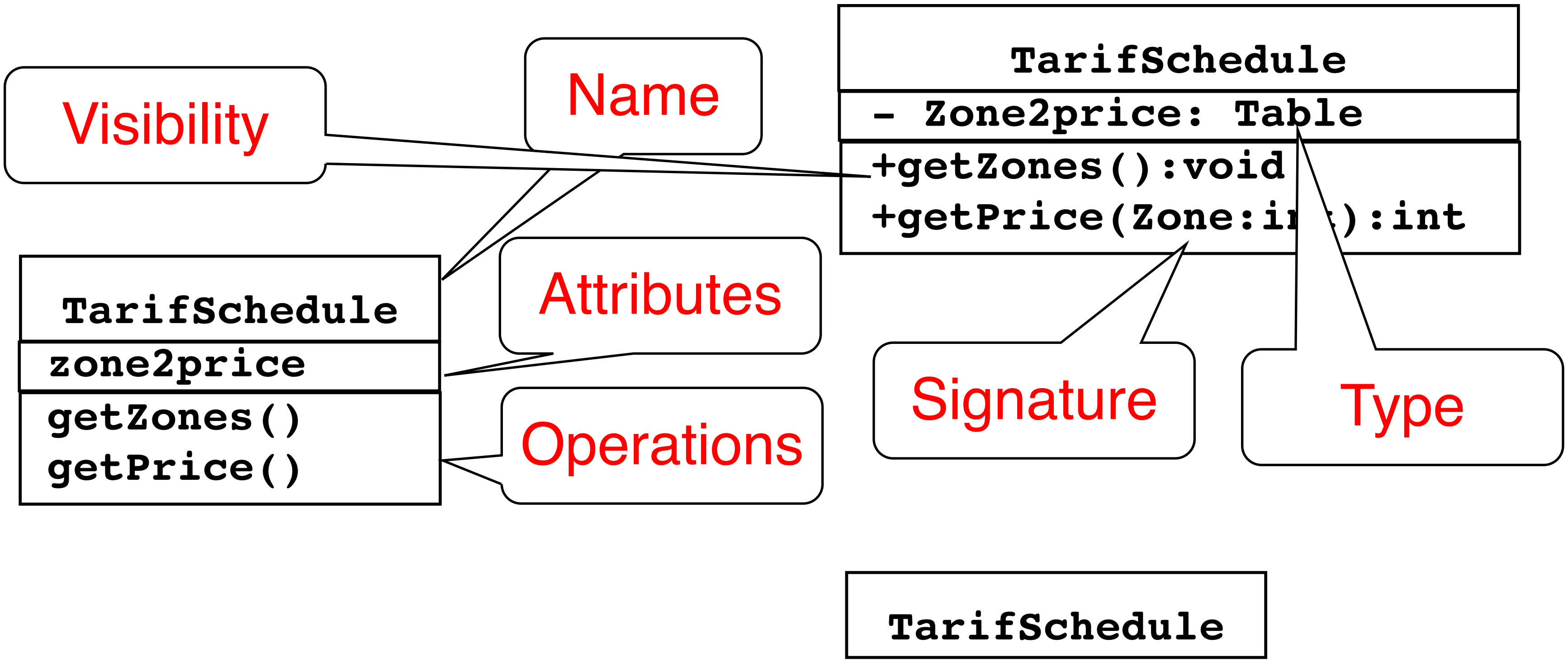
Use Case Diagrams (summary)

- Use case diagrams represent behavior from external view
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams

- Class diagrams represent the structure of the system.
- Used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

Class Diagrams (Classes)



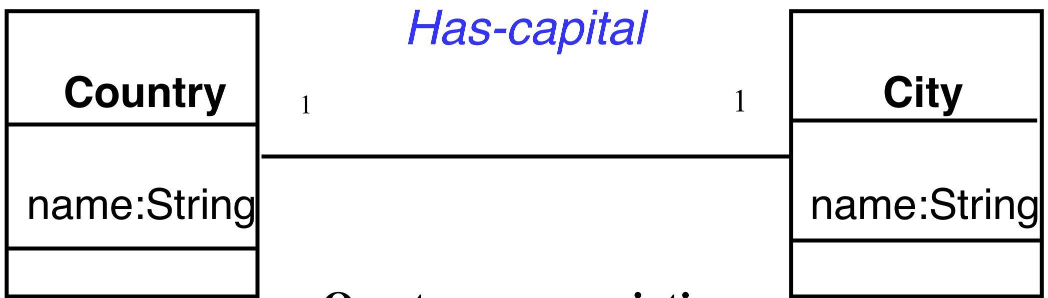
- A *class* represents a concept and it has a name
- A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- Each attribute has a ***type***.
- Each operation has a ***signature***.
- The class name is the only mandatory information.

Class Diagrams (Instances)

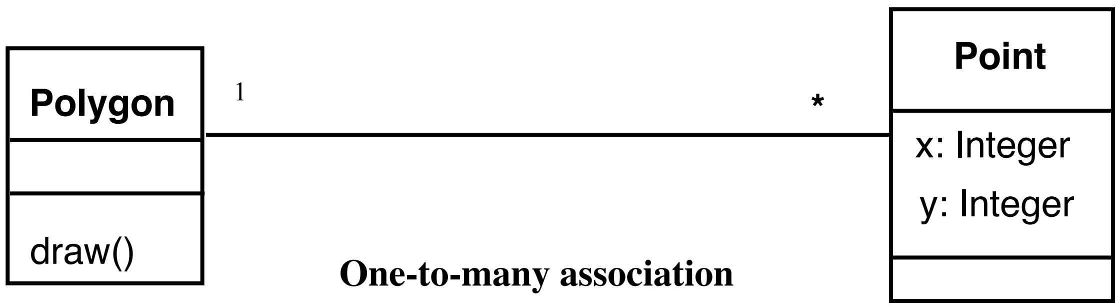
```
tarif_1974:TarifSchedule
--------------------------------
zone2price = {{'1', .20},
{'2',.40},{'3', .60}}
LastUpdate:String="2004/12/15"
```

- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Class Diagrams (Associations)



One-to-one association



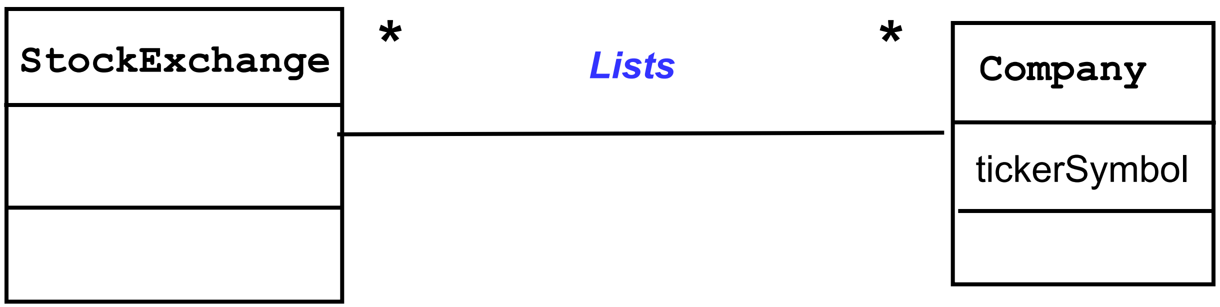
One-to-many association

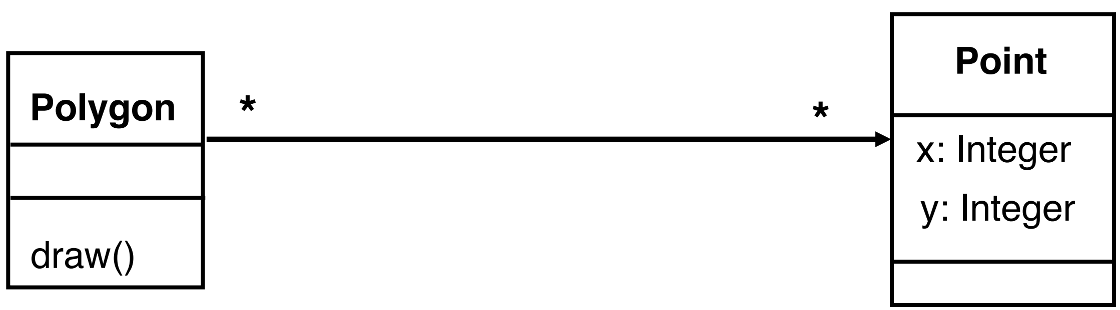
One-to-many association

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Class Diagrams (Associations)

Many-to-Many Associations

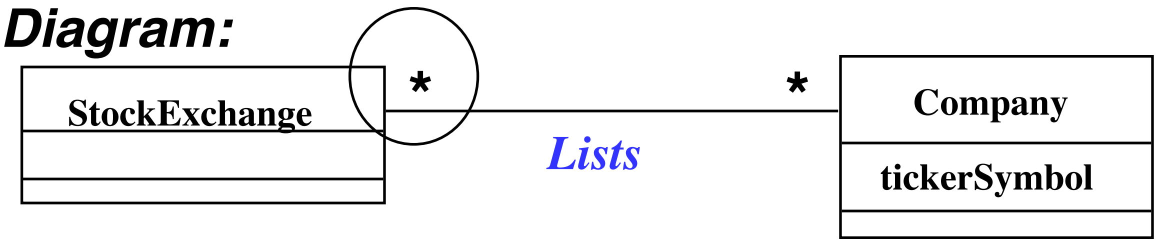




From Problem Statement To Object Model

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

Class Diagram:

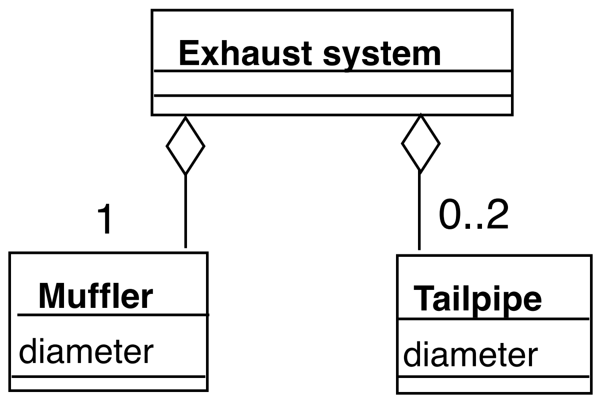


Java Code (incomplete)

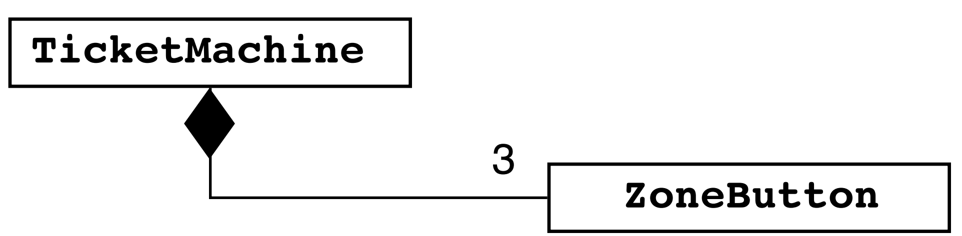
```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

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

Class Diagrams (Aggregation)

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

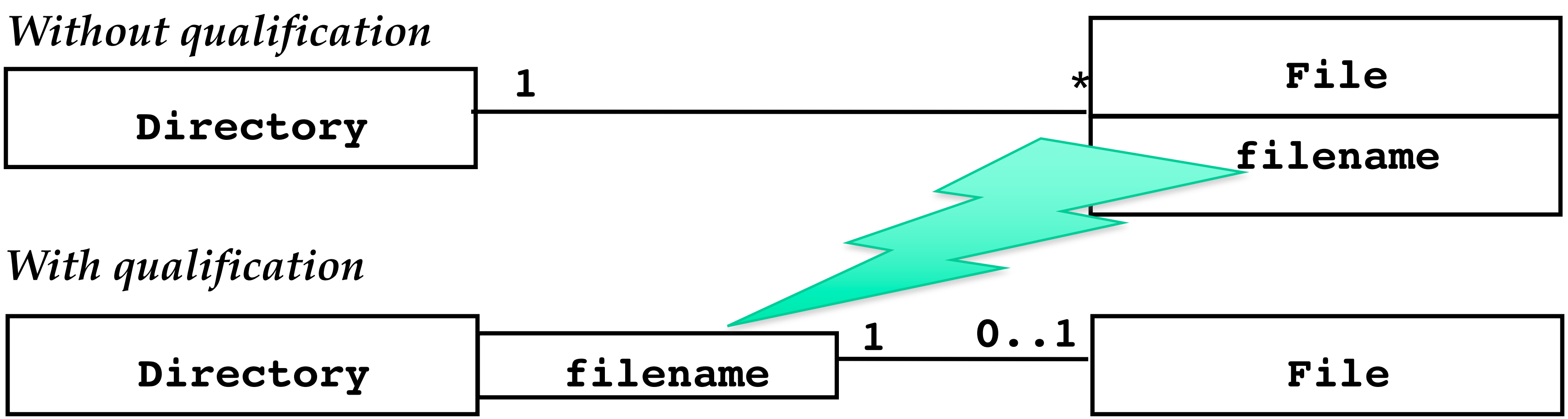


- A solid diamond denotes ***composition***, a strong form of aggregation where components cannot exist without the aggregate.



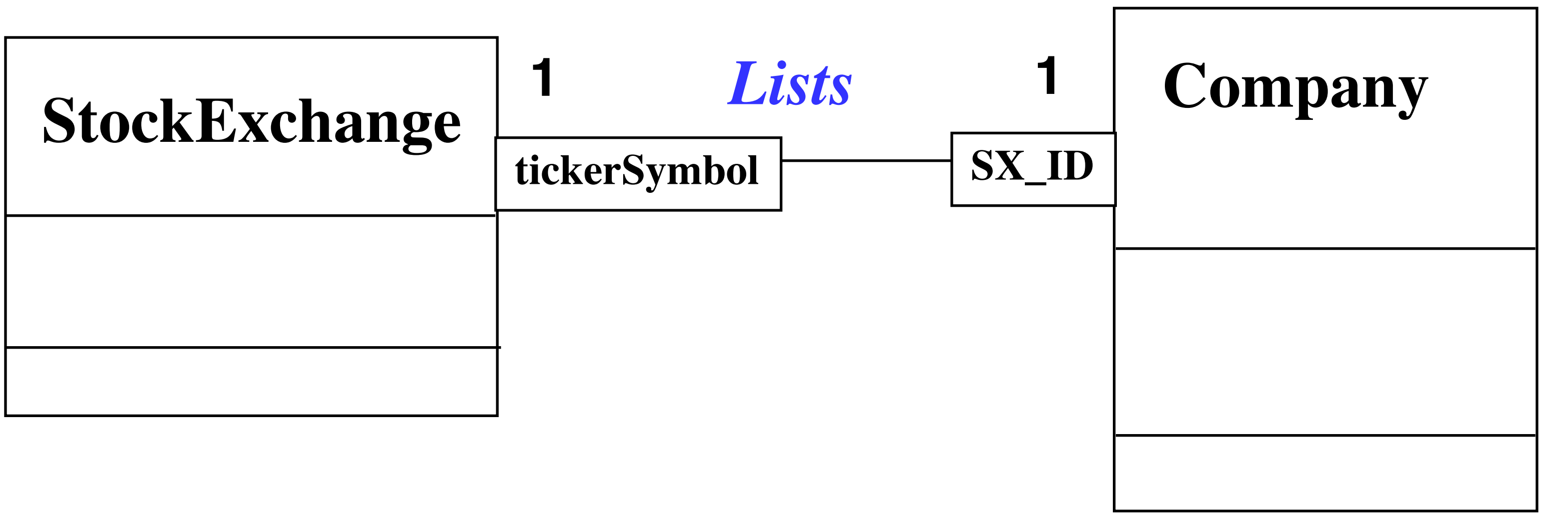
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Class Diagrams (Qualifiers)

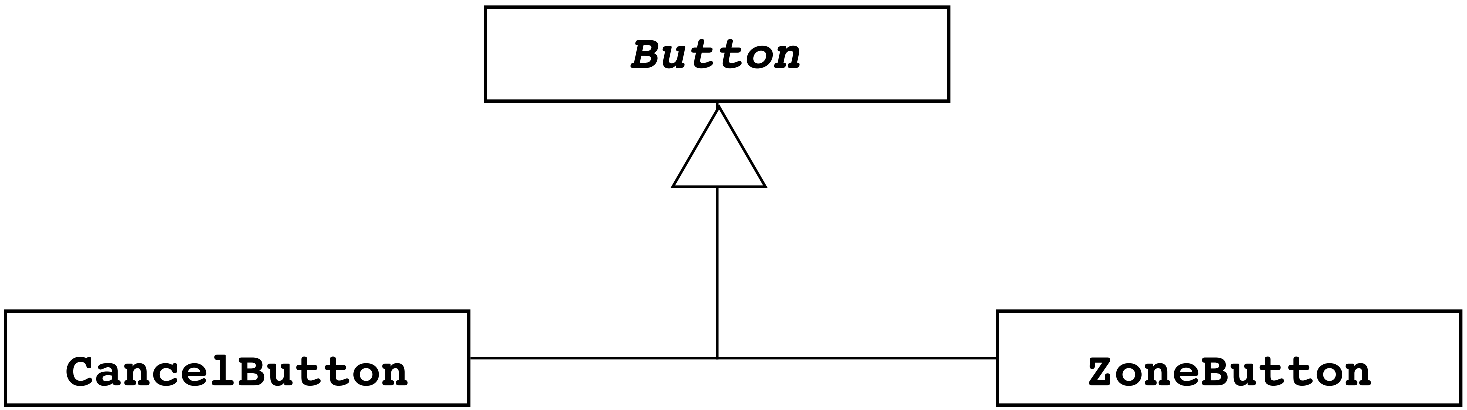


- Qualifiers can be used to reduce the multiplicity of an association.

Class Diagrams (Qualifiers)

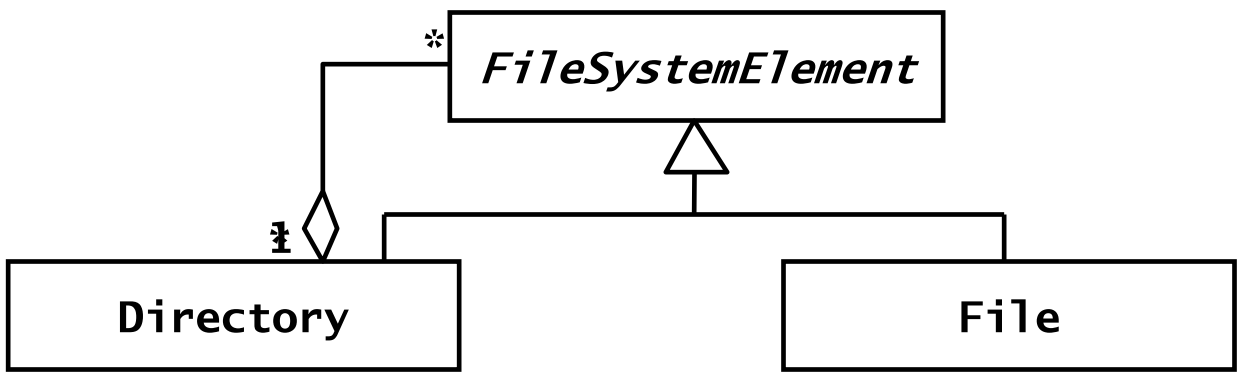


Class Diagrams (Inheritance)



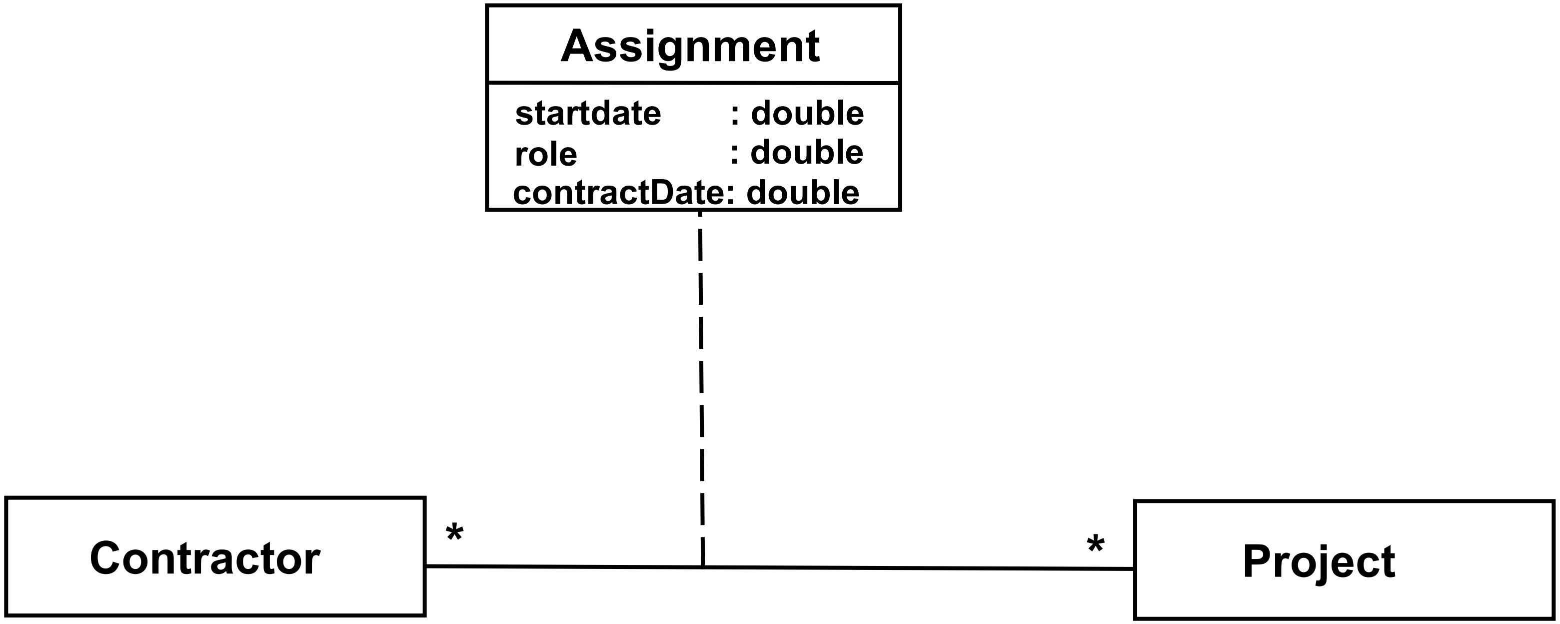
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

More examples

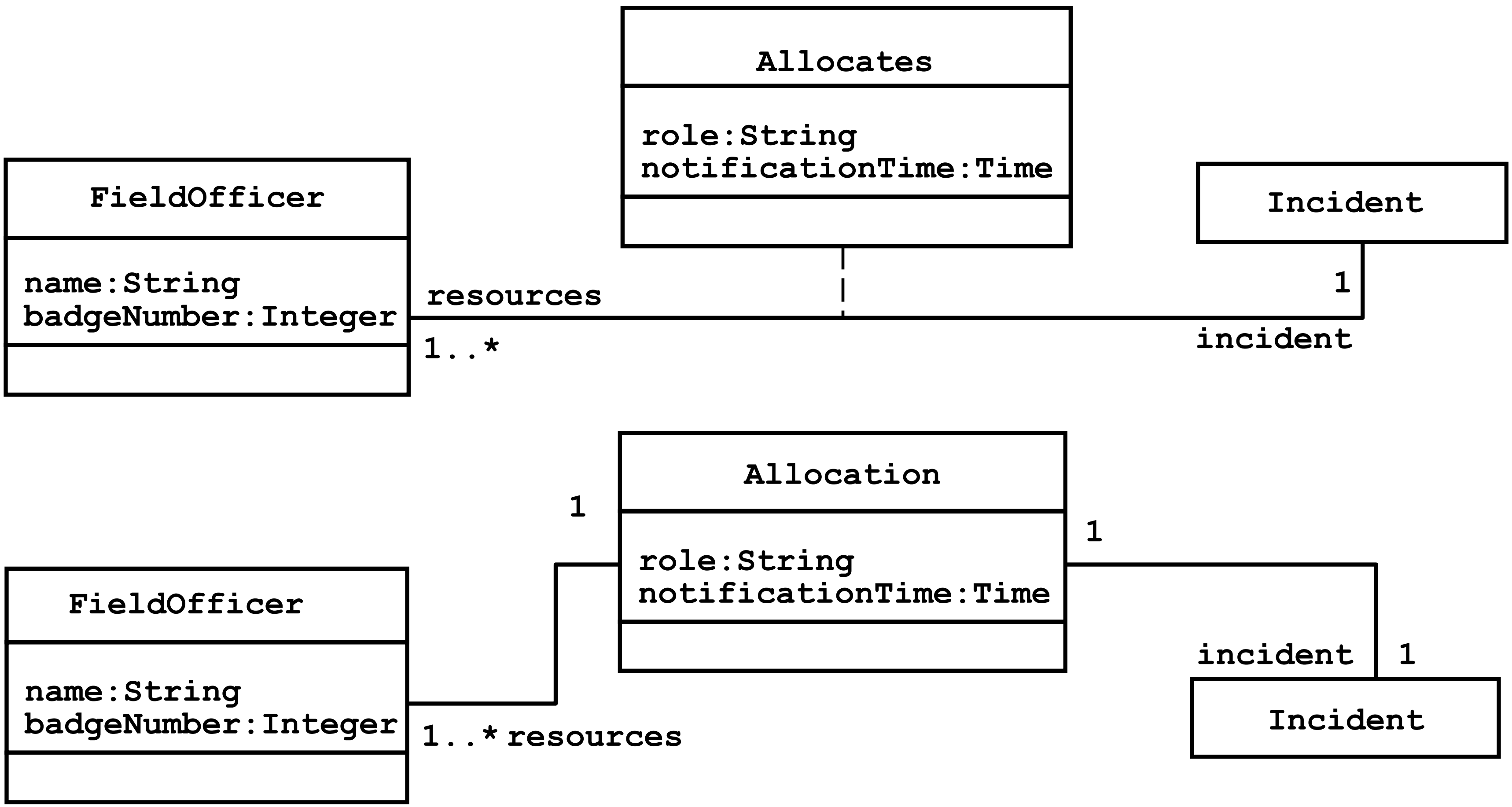


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Class Diagrams (Association class)



Class Diagrams (End Notations and Association)



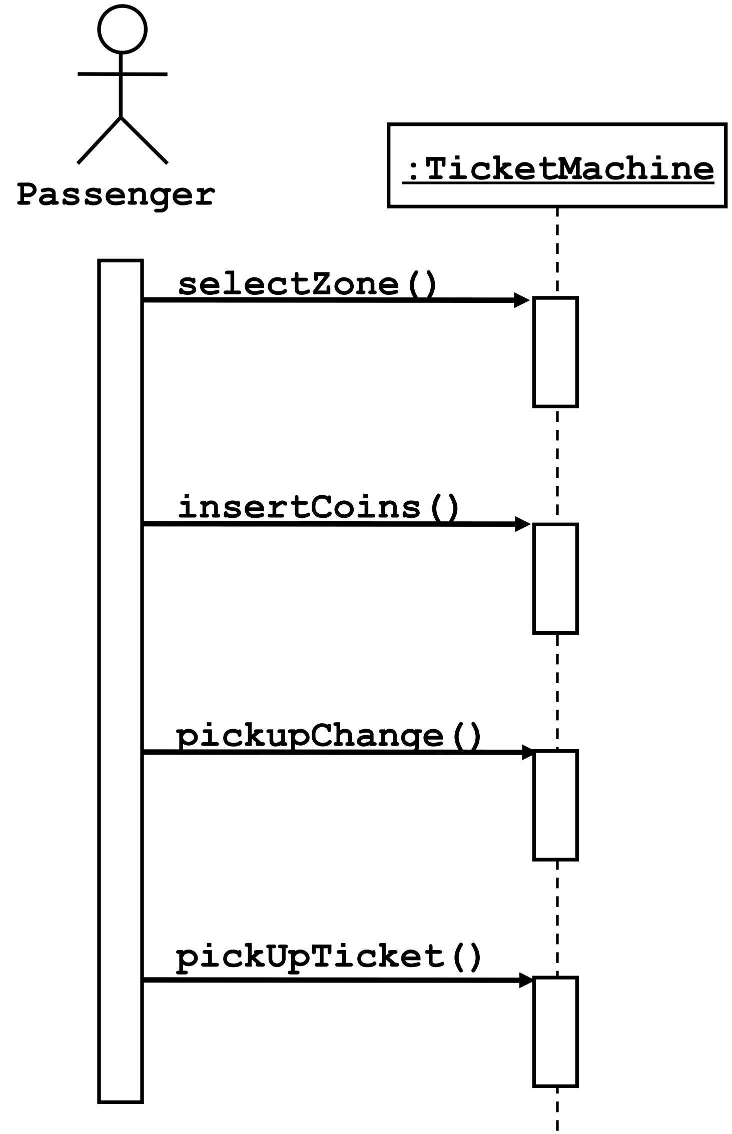
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Interaction Diagrams

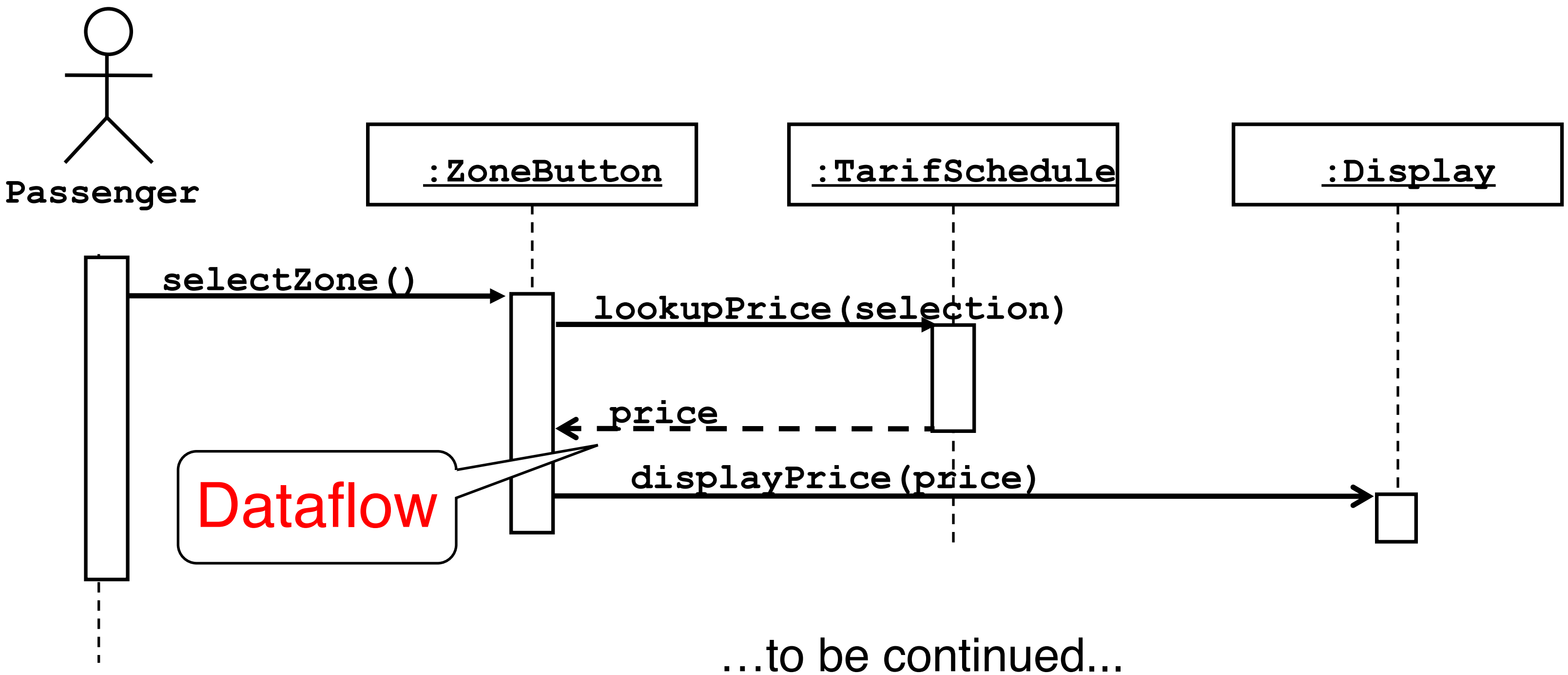
- **Interaction diagrams** are the main design-level behavior modeling technique in UML
  - Sequence diagrams
  - Communication diagrams
- **Interactions diagram** is a graphical visualization of sequences of messages between objects
- Object receiving a message activates the relevant method.
- The time when the flow of control is focused in an object is called **activation**

Sequence Diagrams

- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Objects*** participating in the interaction are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines



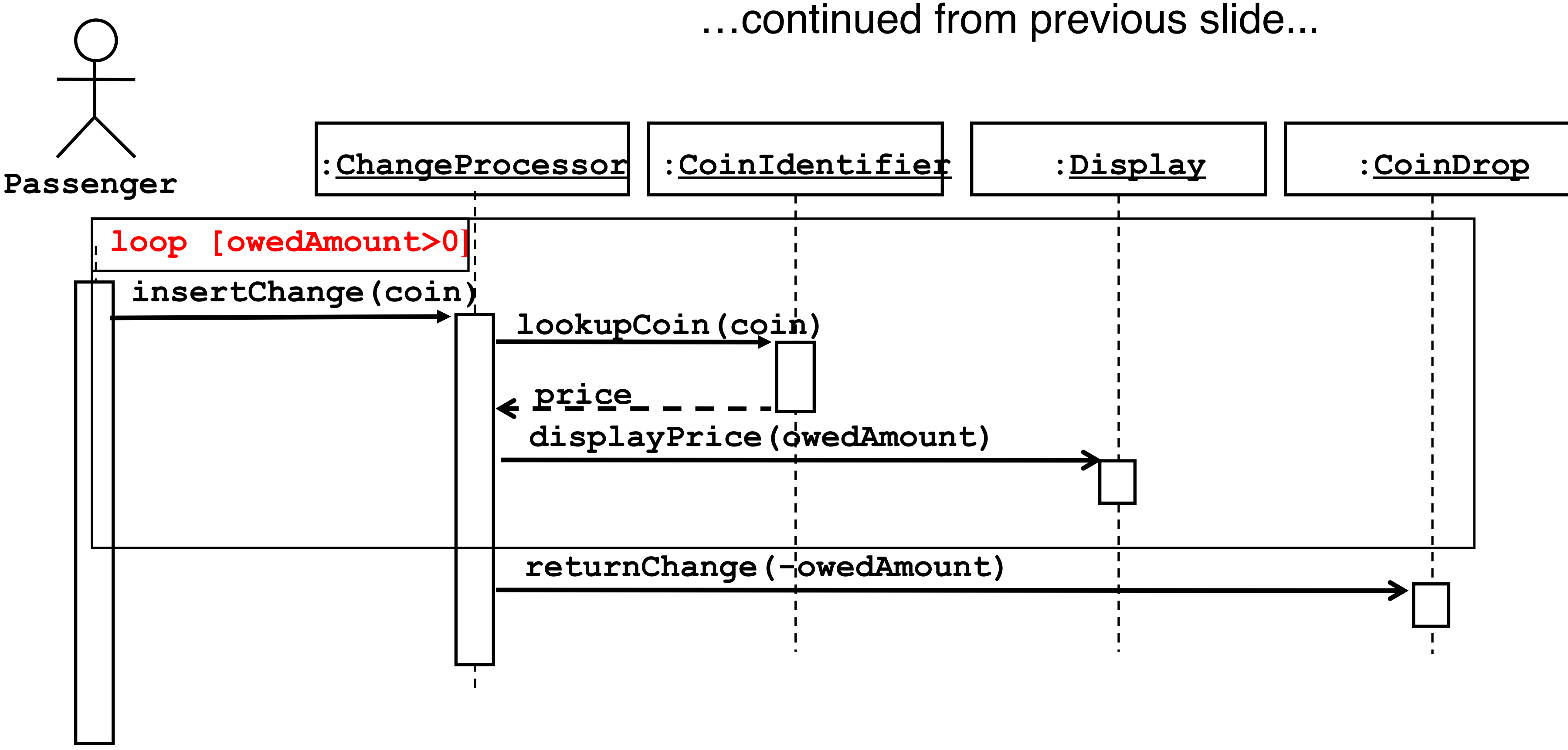
Sequence Diagrams



- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations
- Horizontal arrows indicate data flow
- Vertical dashed lines indicate lifelines

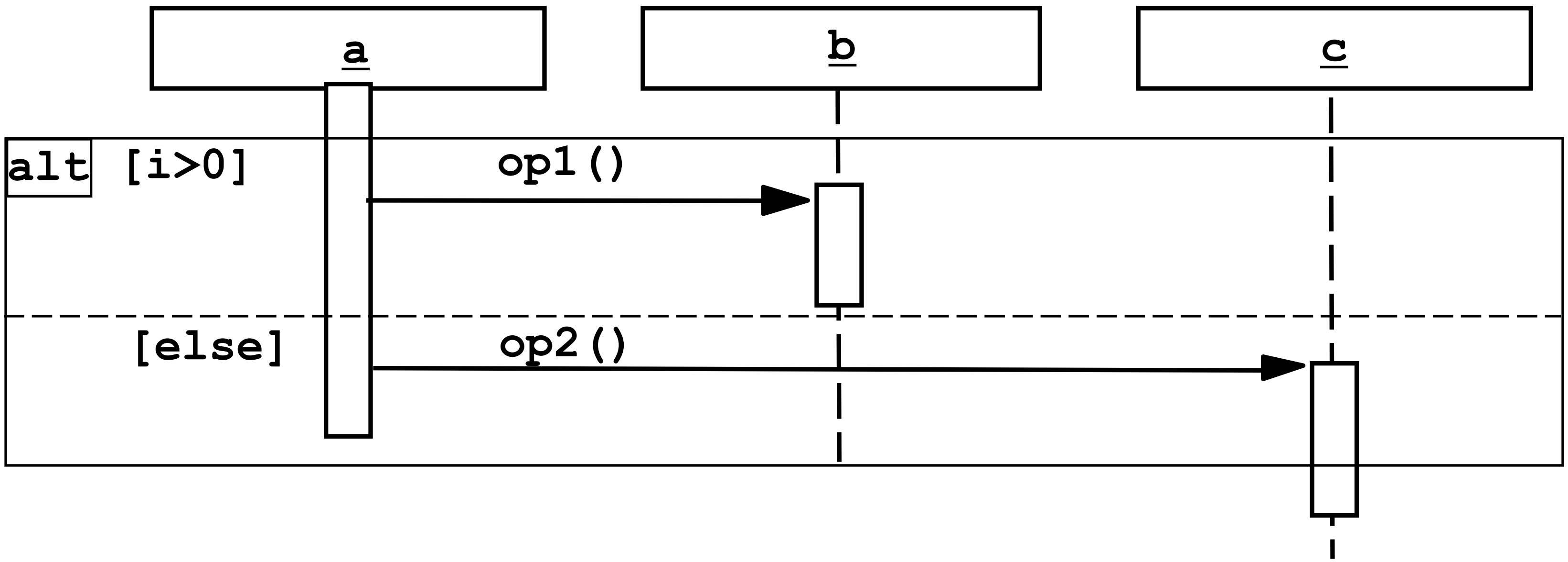
Sequence Diagrams (Iteration & condition)

…continued from previous slide...

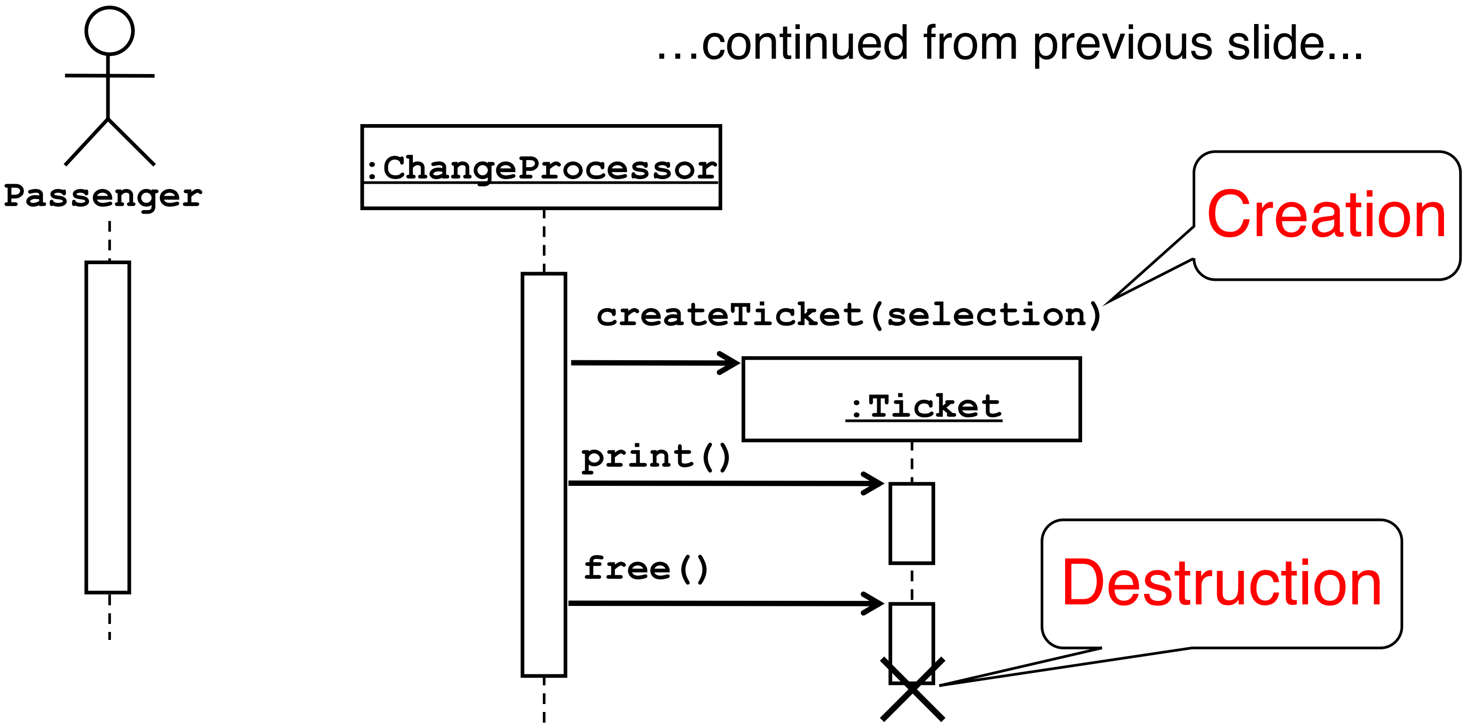


…to be continued...

Sequence Diagrams (branches)

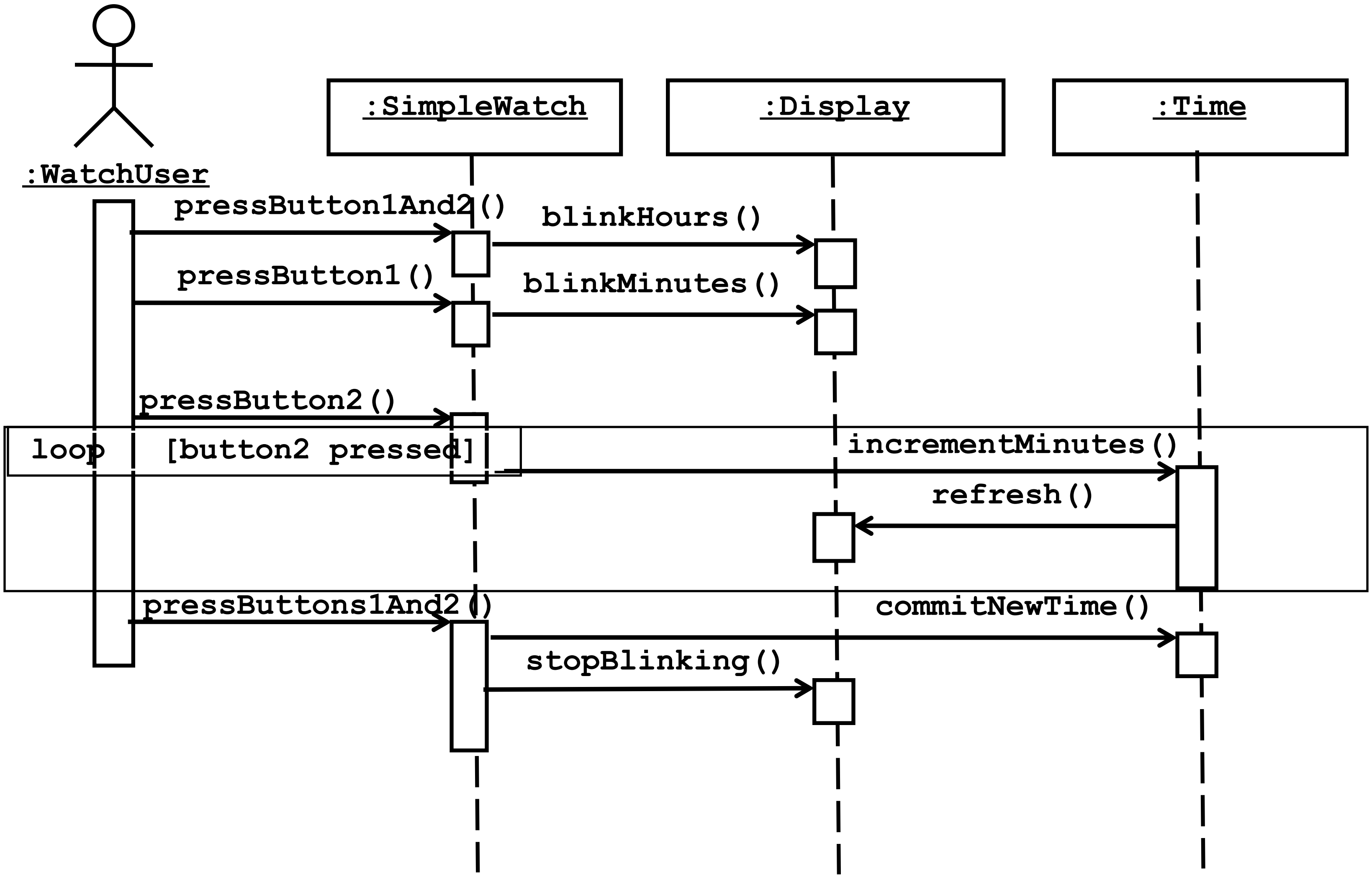


Sequence Diagrams (Creation and destruction)



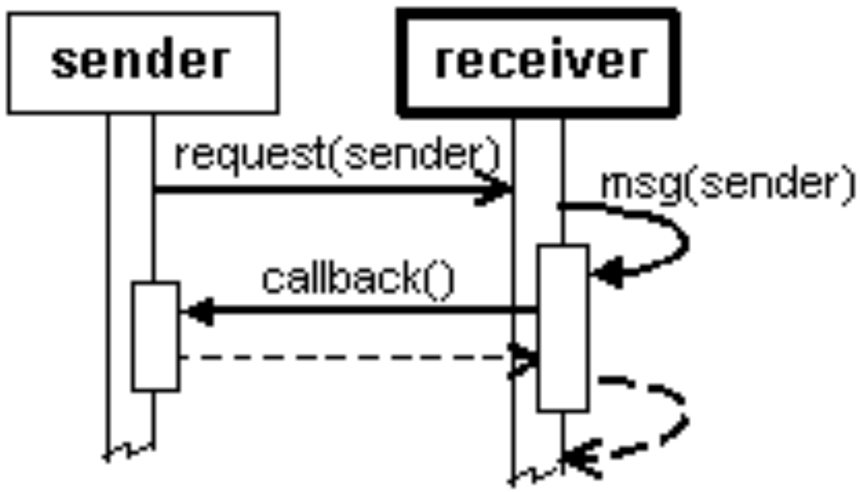
- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Example



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

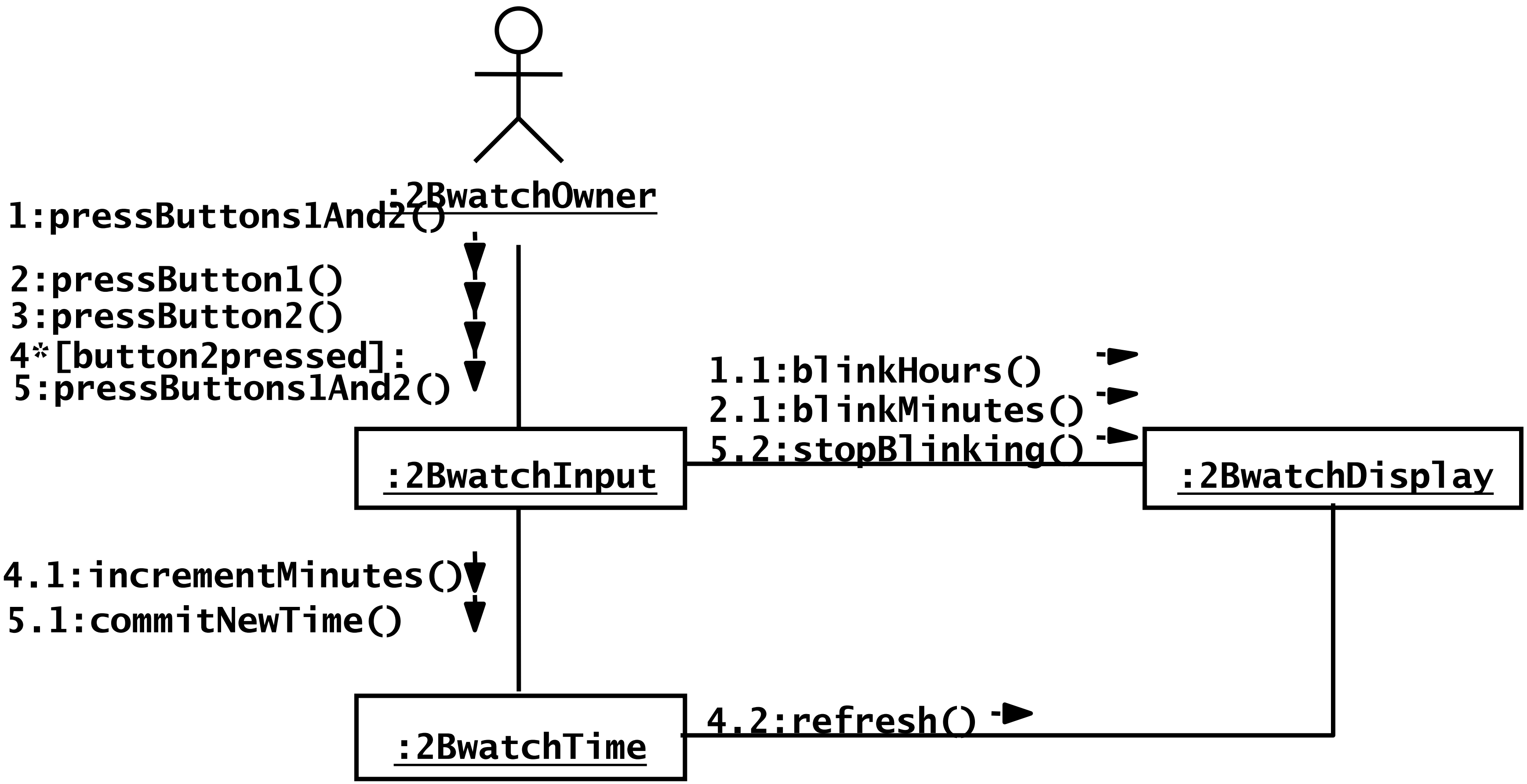
Some other example



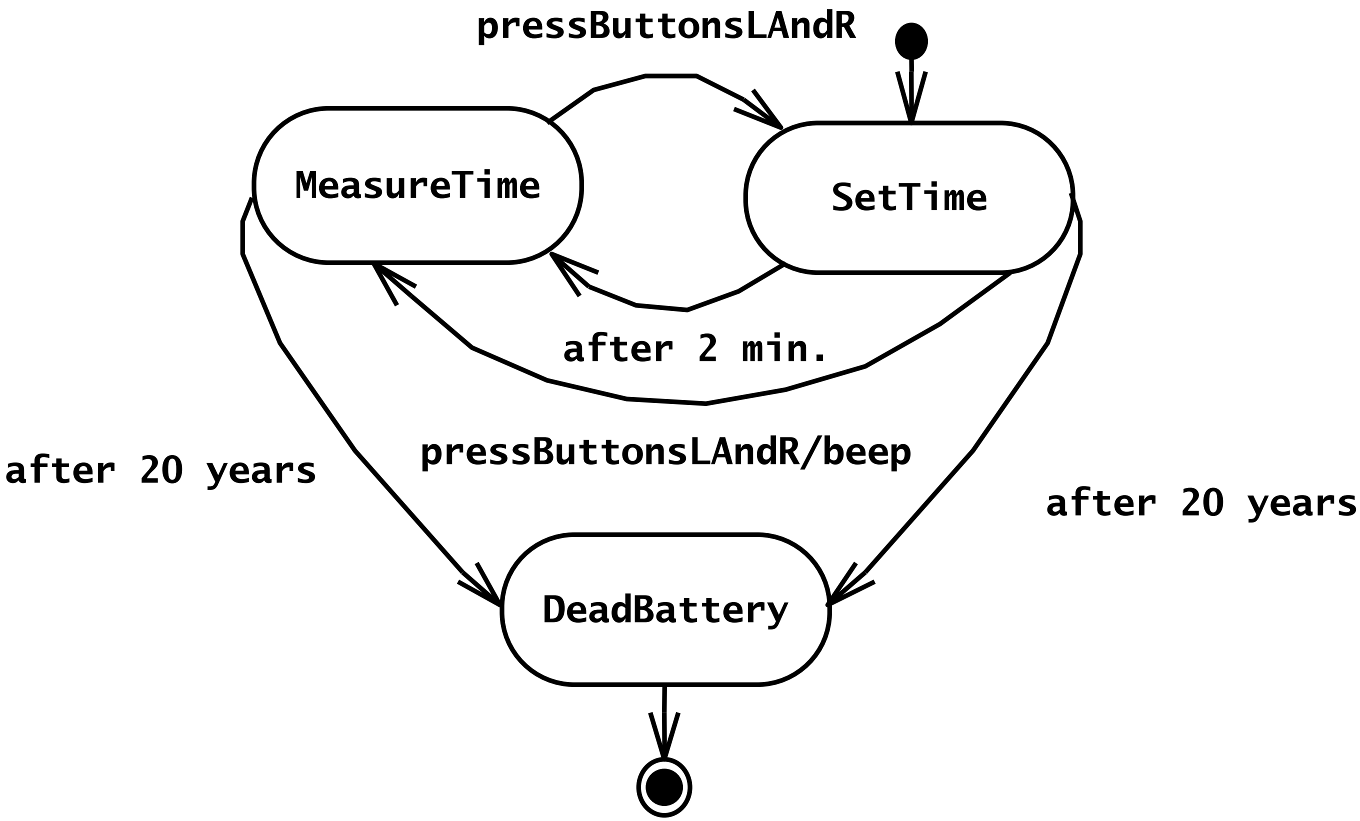
Sequence Diagrams (Summary)

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

Communication diagrams

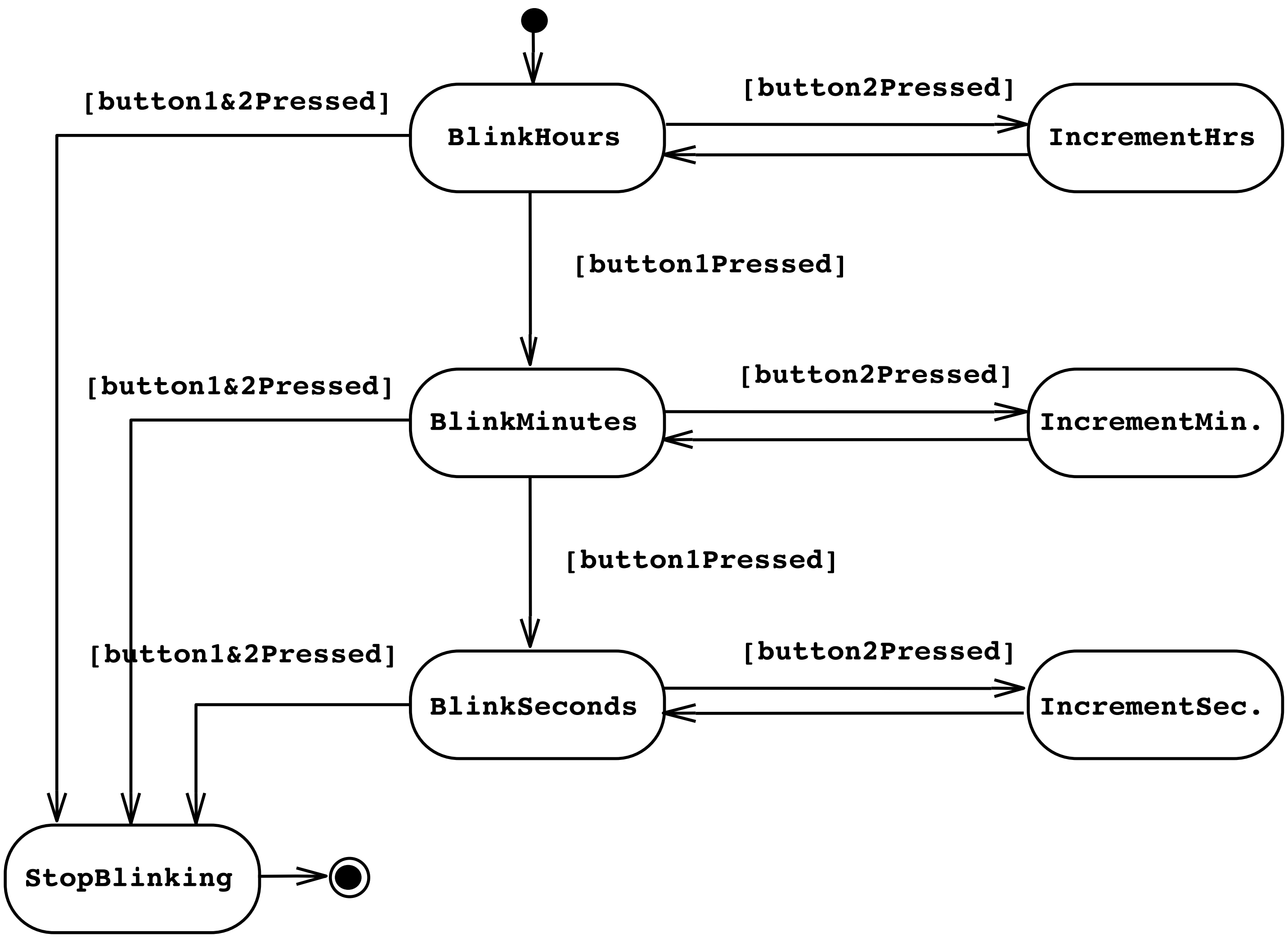


State Charts



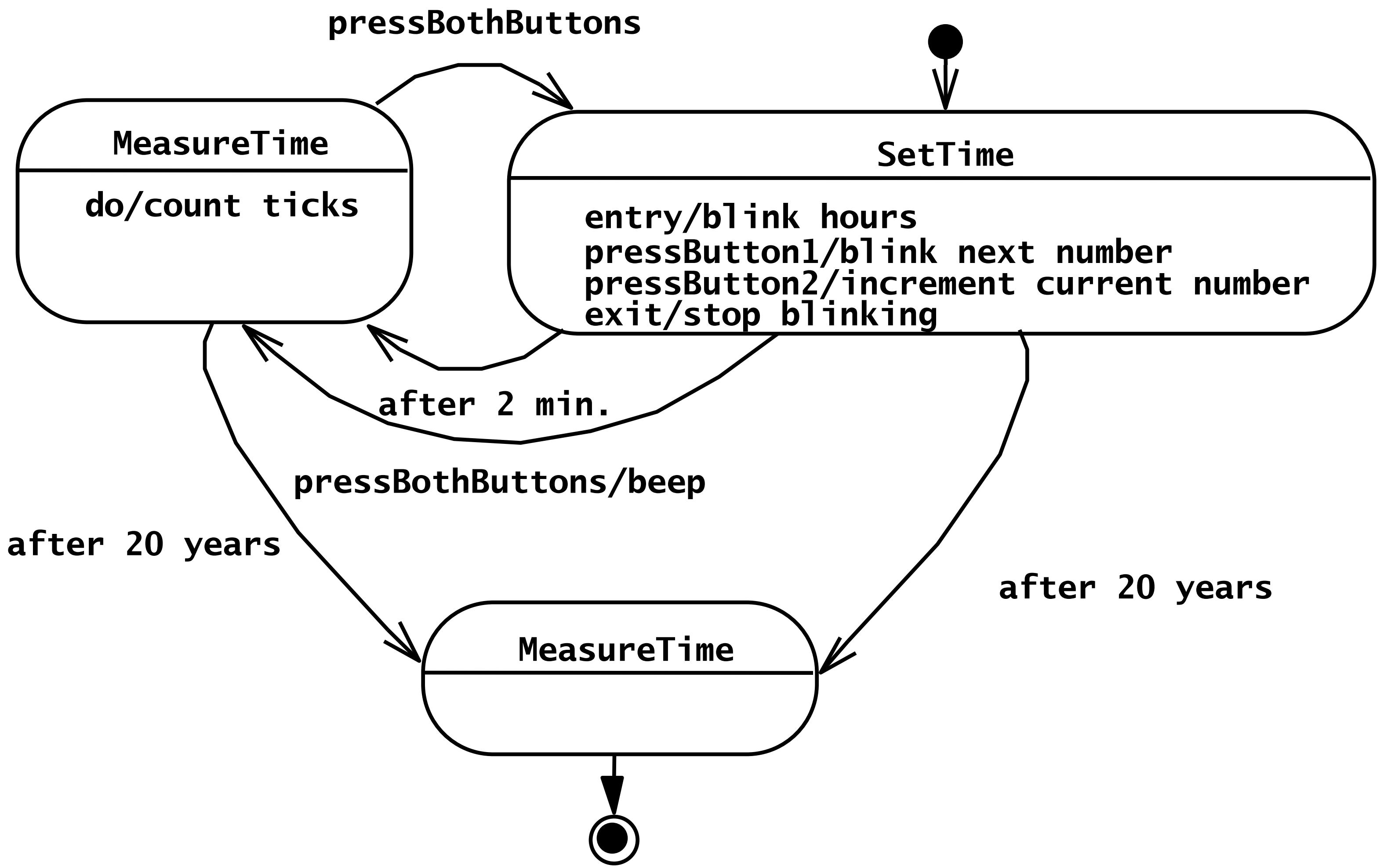
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

State Chart Diagrams

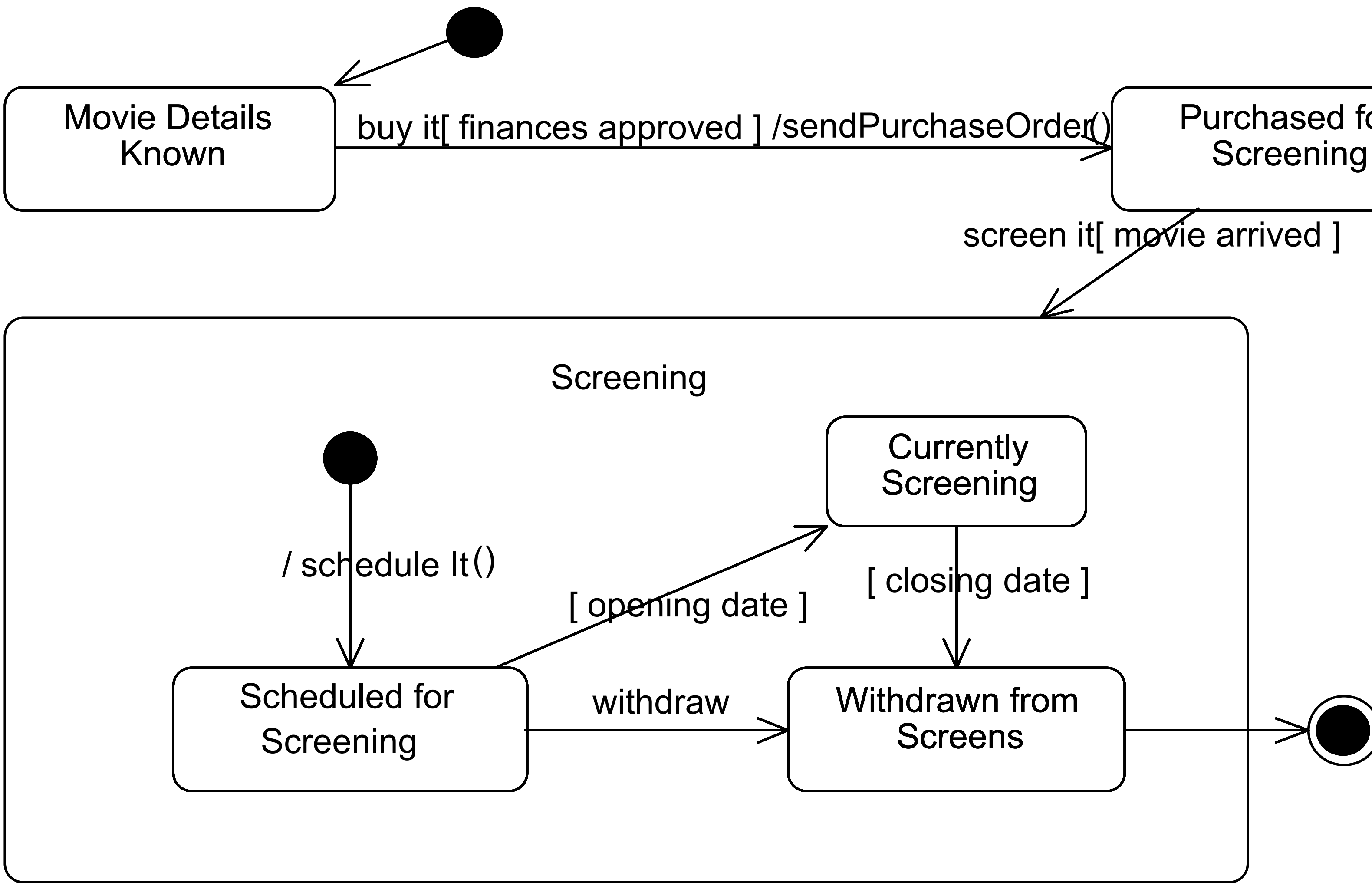


Represent behavior as states and transitions

Internal transitions



State Chart Diagrams (nested charts)



Adopted from Maciazcek

Activity Diagrams

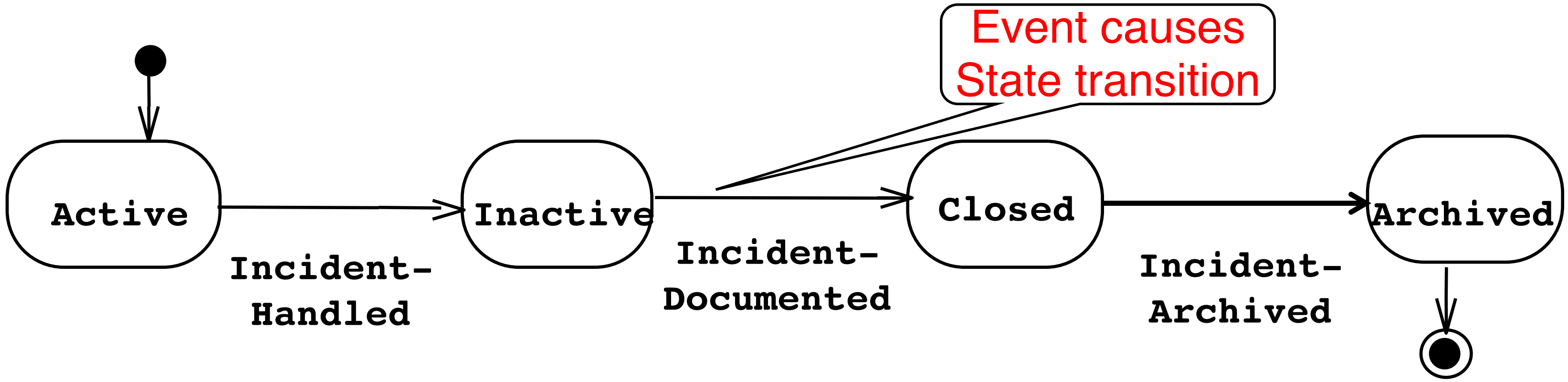
- An activity diagram shows flow control within a system

- An activity diagram is a special case of a state chart diagram in which states are activities ("functions")
- Two types of states:
  - *Action state*:
    - Cannot be decomposed any further
    - Happens "instantaneously" with respect to the level of abstraction used in the model
  - *Activity state*:
    - Can be decomposed further
    - The activity is modeled by another activity diagram

Statechart Diagram vs. Activity Diagram

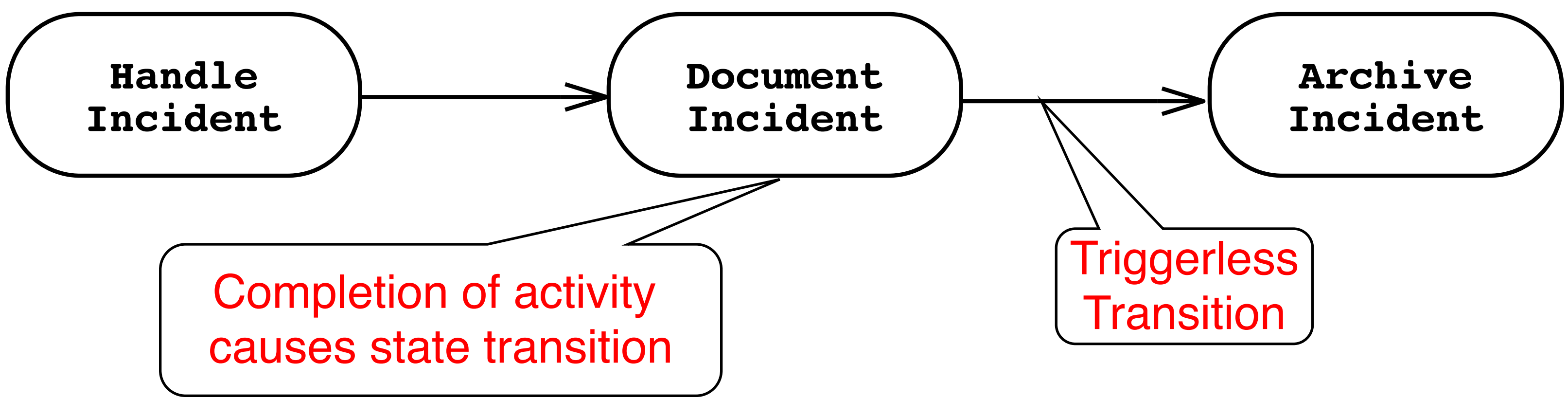
Statechart Diagram for Incident

(State: Attribute or Collection of Attributes of object of type Incident)

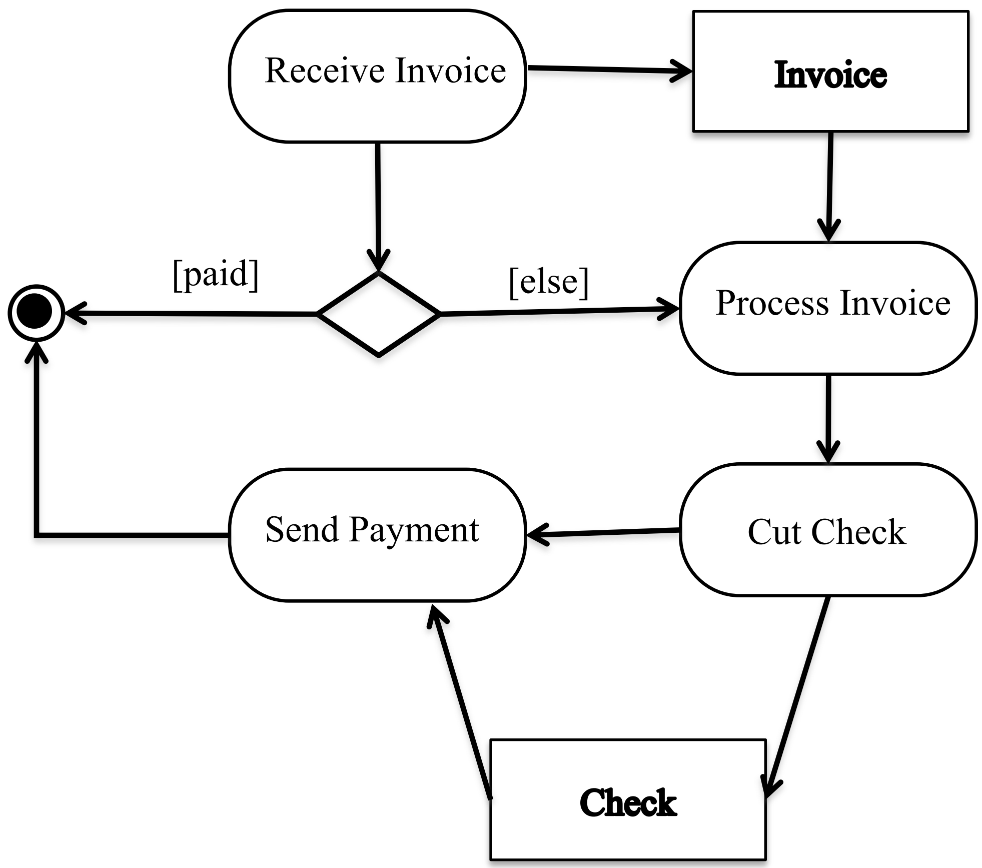


Activity Diagram for Incident

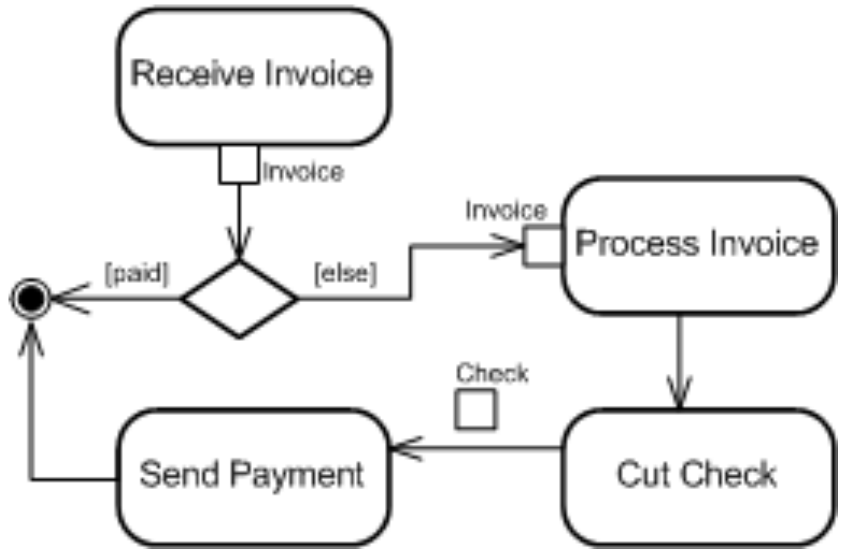
(State: Operation or Collection of Operations)



Object flow

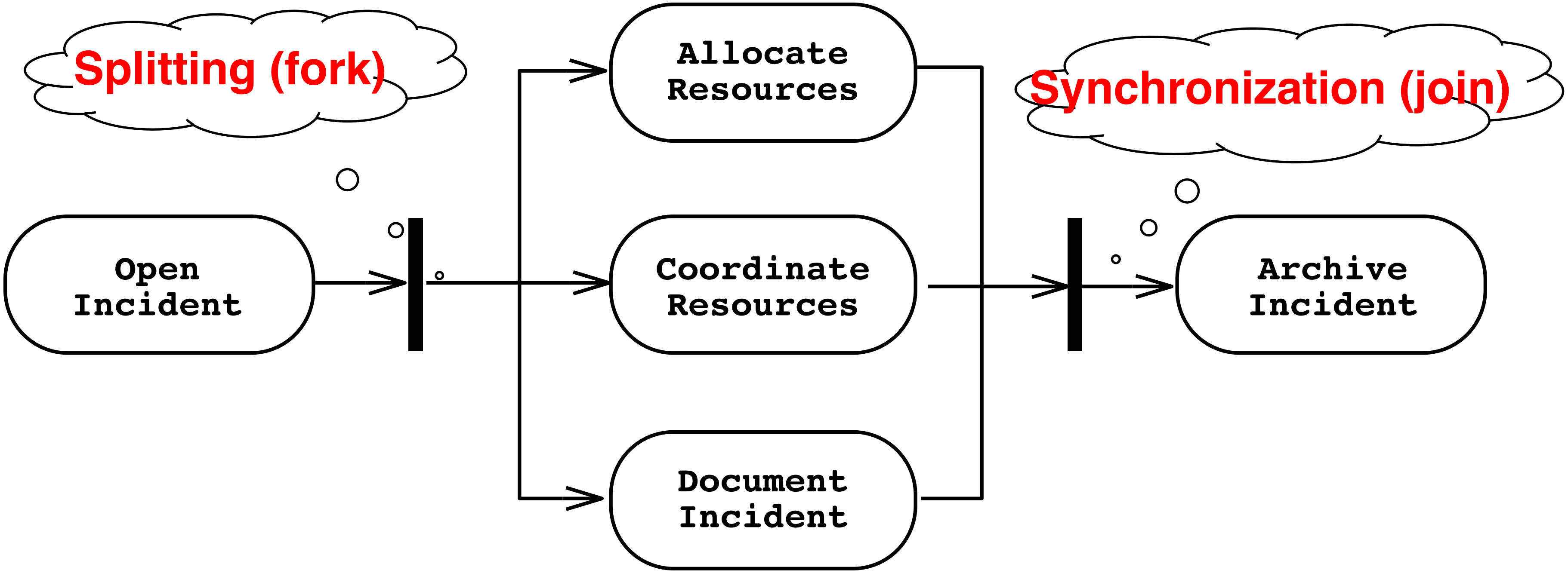

Receive Invoice
Invoice
[paid]
[else]
Process Invoice
Send Payment
Cut Check
Check

Decisions and Object flow



Activity Diagrams: Modeling Concurrency

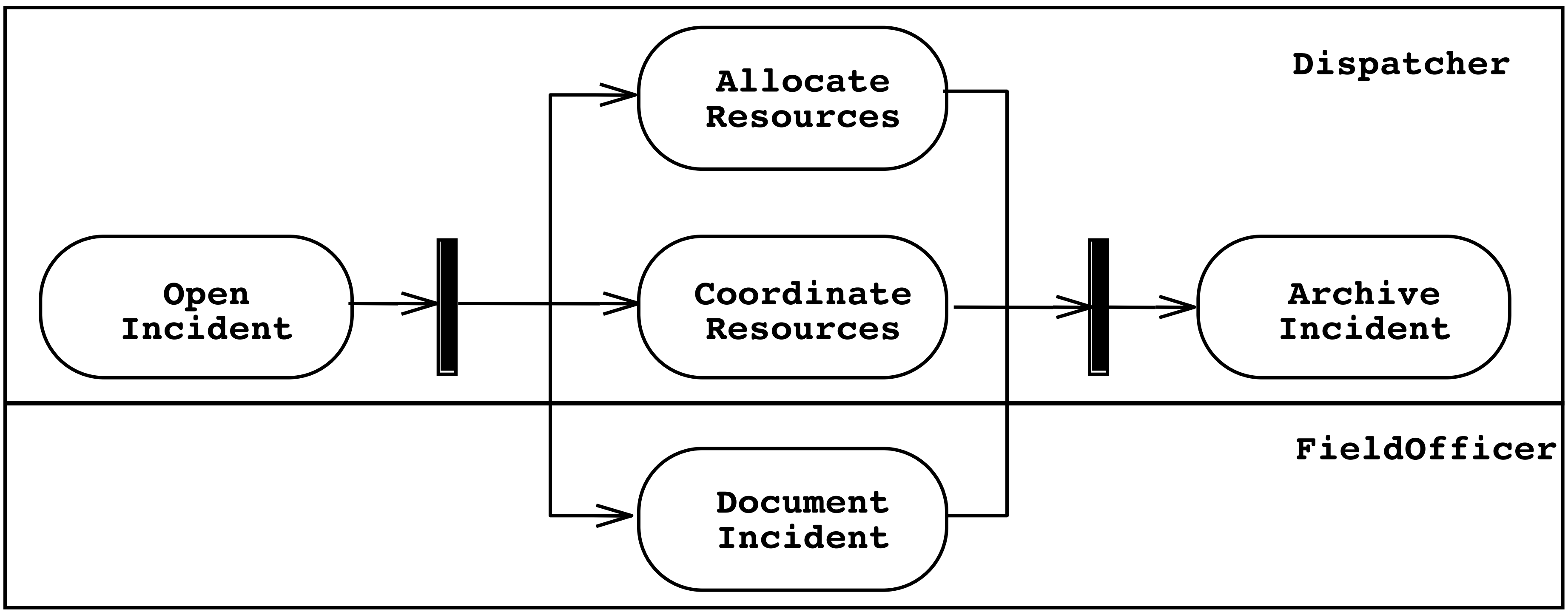
- Synchronization of multiple activities
- Splitting the flow of control into multiple threads



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Activity Diagrams: Swimlanes

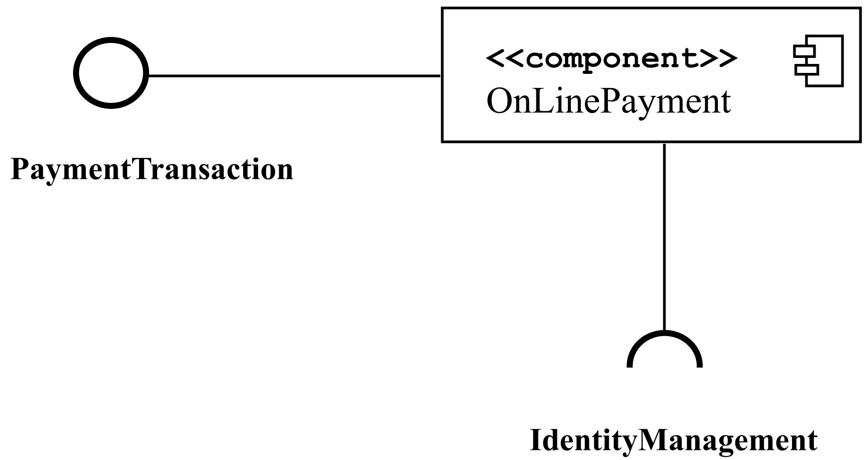
- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.

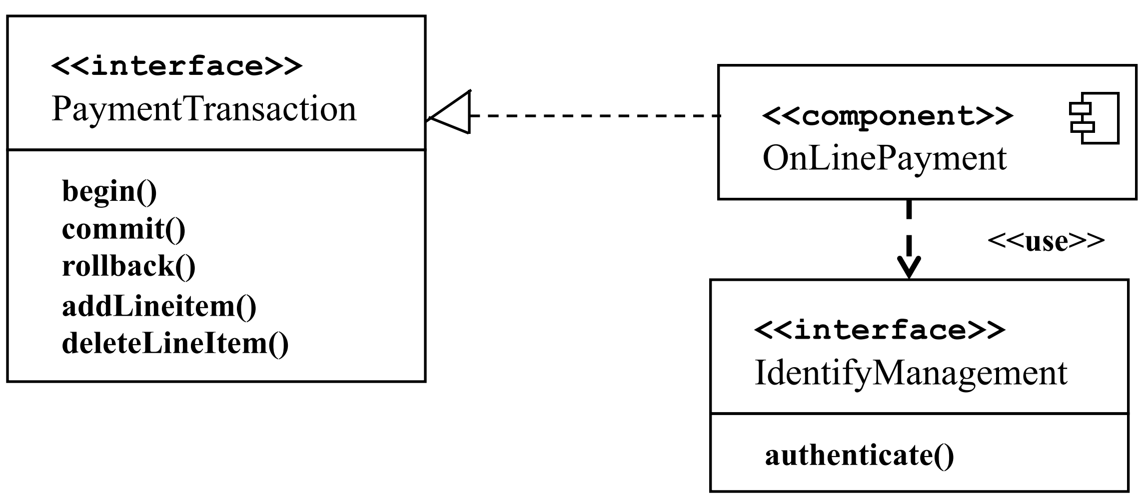


Component and Deployment diagrams

- Models for physical implementation of the system
- Show system components, their structure and dependencies and how they are deployed on computer nodes
- Two kinds of diagrams:
  - component diagrams
  - deployment diagrams
- Component diagrams show structure of components, including their interface and implementation dependencies
- Deployment diagrams show the runtime deployment of the system on computer nodes

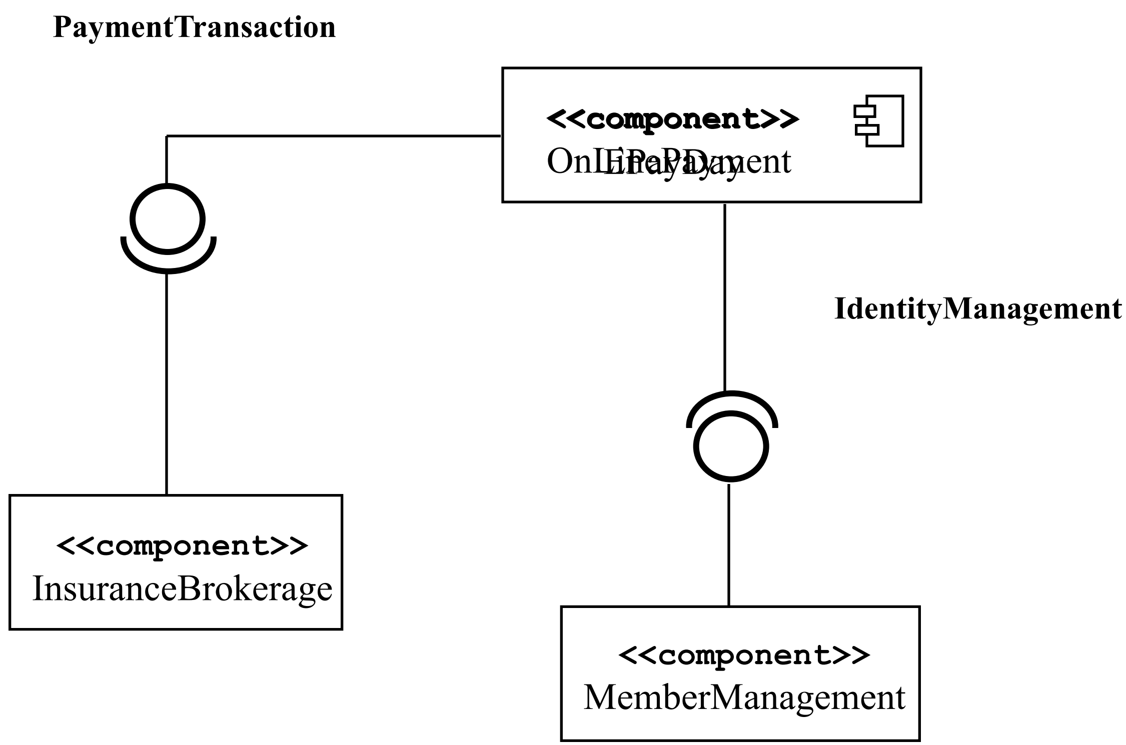
Component Diagrams (component interface)





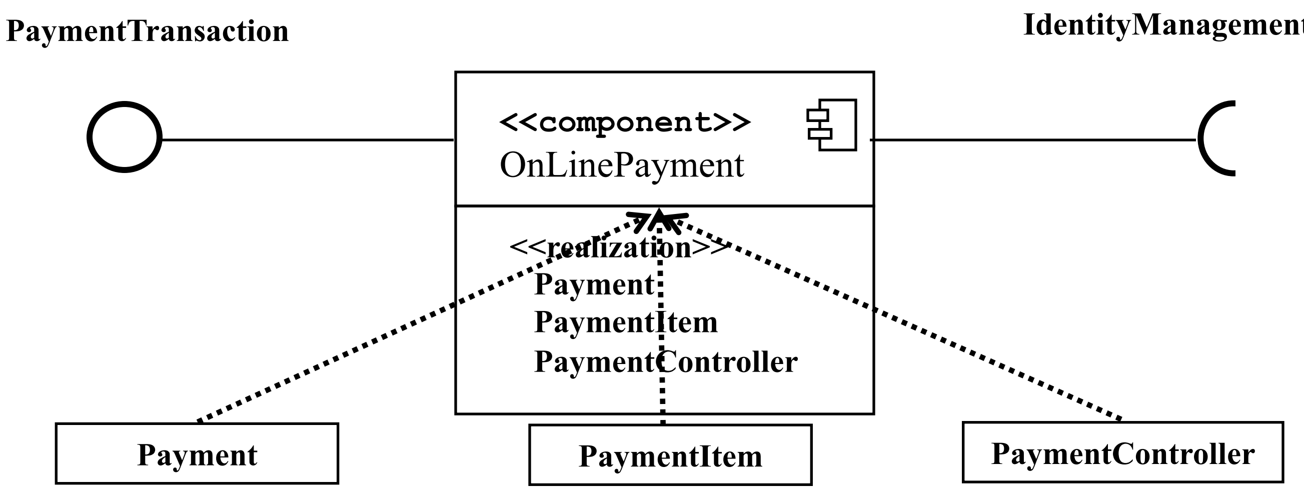
Adopted from Shaum

Component Diagram



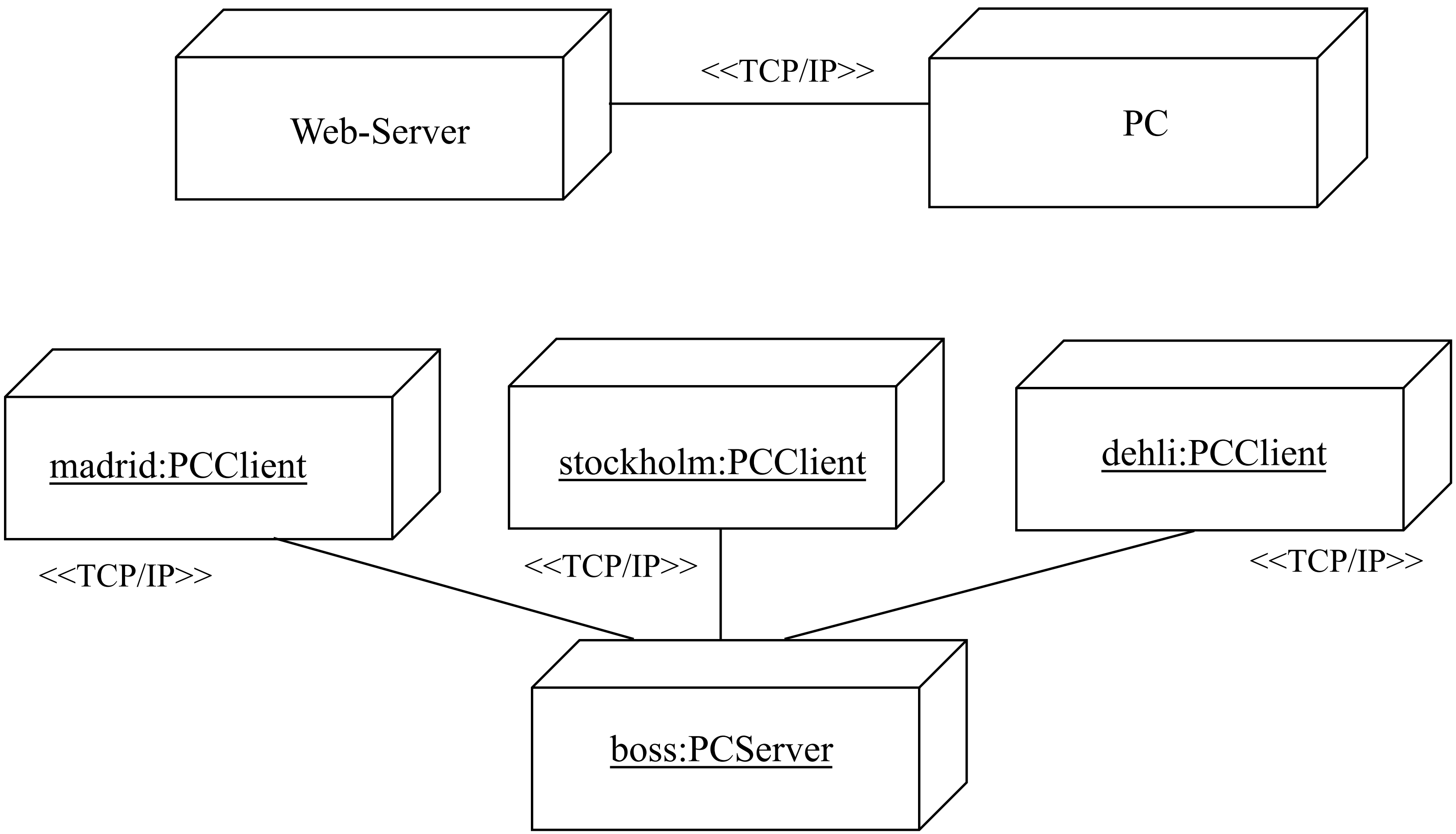
Adopted from Shaum

Component Diagram (component realization)



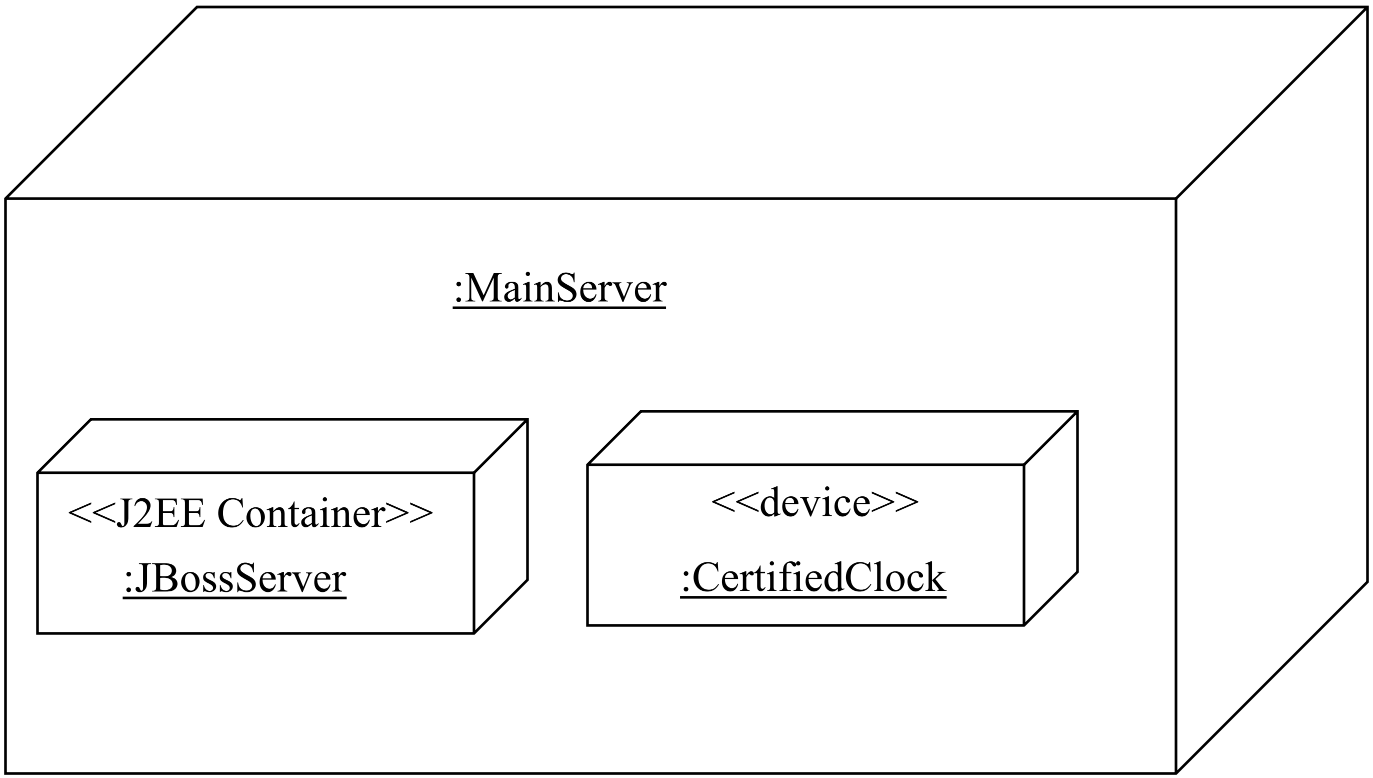
Adopted from Shaum

Deployment diagrams



Adopted from Shaum

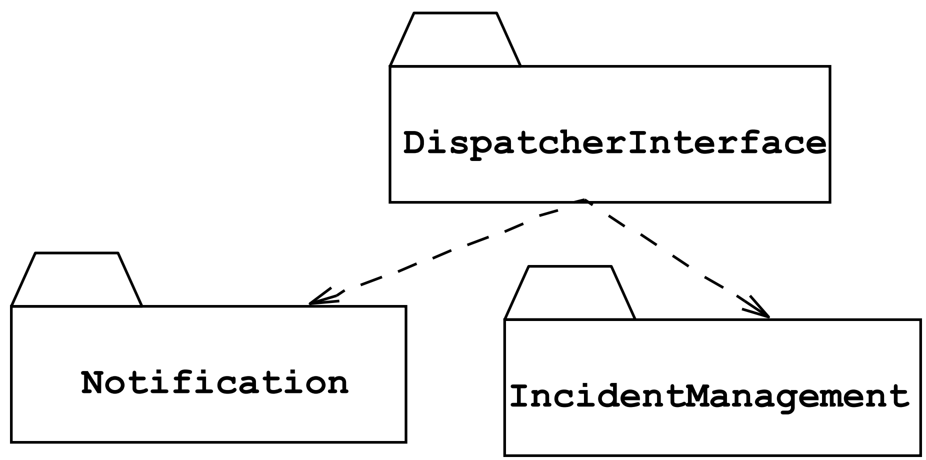
Deployment diagrams



Adopted from Shaum

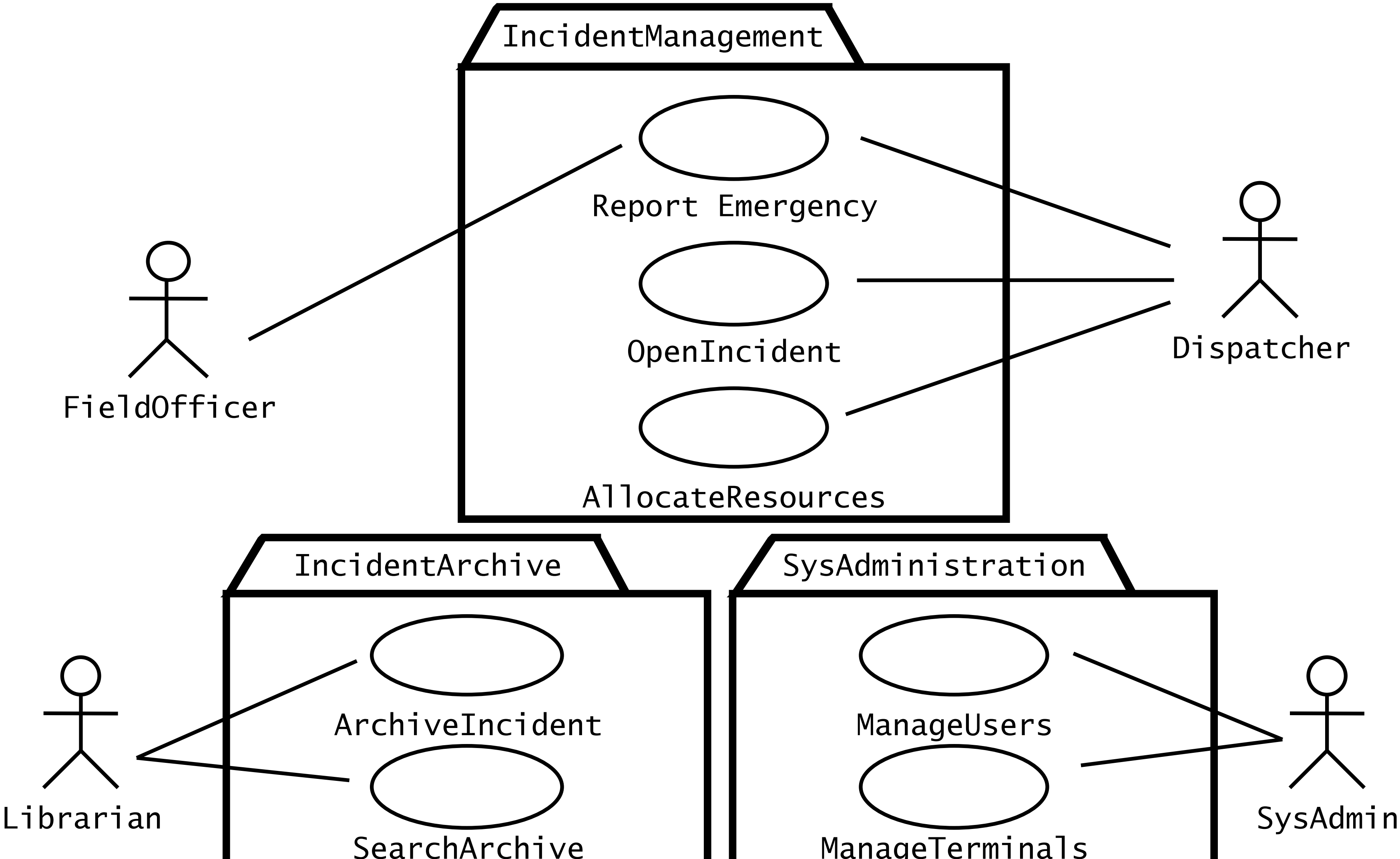
Packages

- A package is a UML mechanism for organizing elements into groups (usually not an application domain concept)
- Packages are the basic grouping construct with which you may organize UML models to increase their readability.

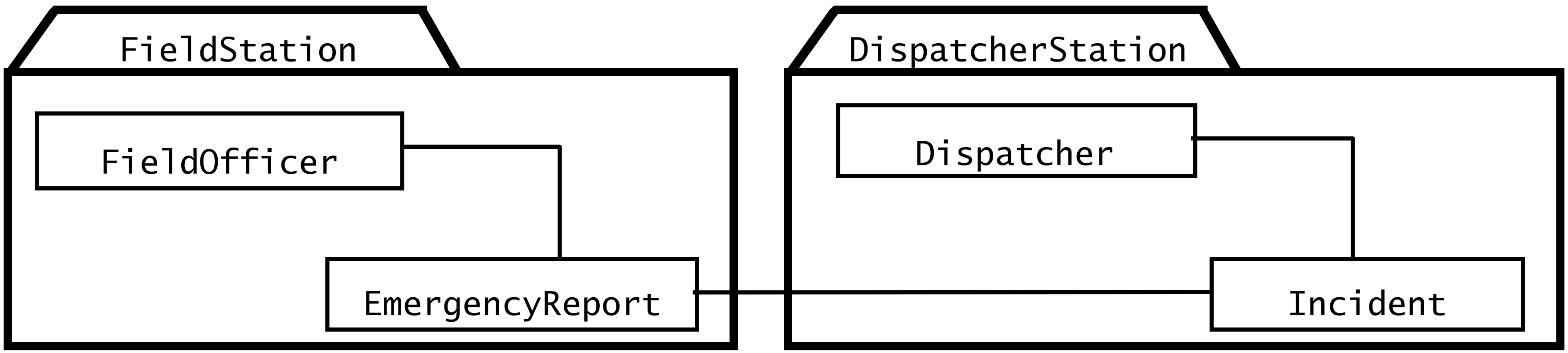


- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

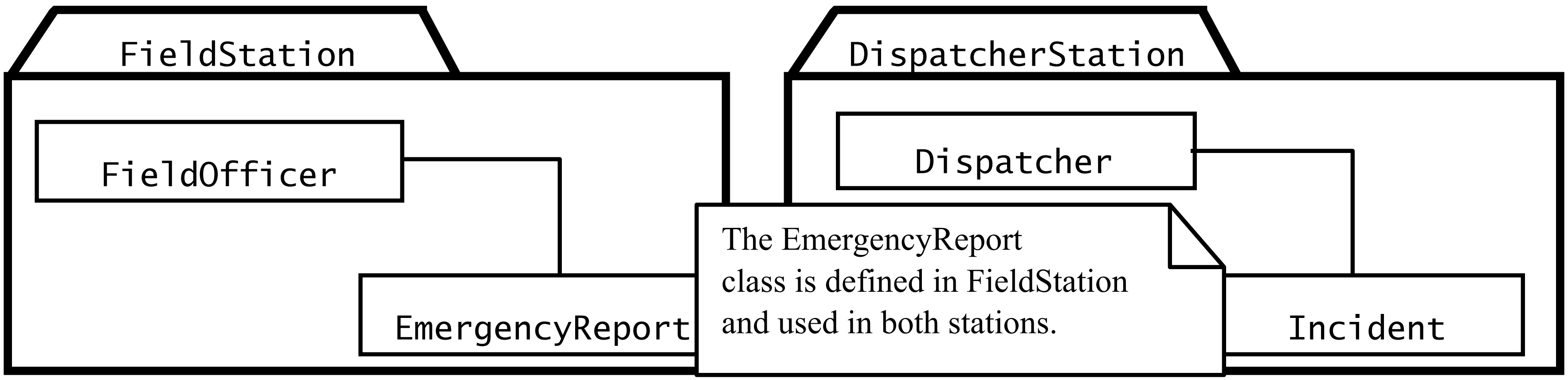
Packages


IncidentManagement
Report Emergency
OpenIncident
AllocateResources
FieldOfficer
Dispatcher
IncidentArchive
ArchiveIncident
SearchArchive
SysAdministration
ManageUsers
ManageTerminals
Librarian
SysAdmin

Packages


FieldStation
FieldOfficer
EmergencyReport
DispatcherStation
Dispatcher
Incident

Notes


FieldStation
FieldOfficer
EmergencyReport
DispatcherStation
Dispatcher
Incident
The EmergencyReport
class is defined in FieldStation
and used in both stations.

UML Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - Powerful, but complex language
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features

- For now we concentrate on a few notations:
  - Functional model: Use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, statechart and activity diagrams

Next lecture

- Requirements elicitation

Chapter 4 in the text-book