GE Java Sessions

Week 2 -
The Object-Oriented Paradigm and UML: An Overview

Presentation for distance students

- Two options
  - Come to class in person
  - http://http://univofmich.adobeconnect.com/xu
    - Login as a guest.
    - Follow screen instruction to install add-on
    - Make sure your microphone and speaker are working

Outline

- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

Important

- This is not an OO or UML course
  - Not a complete overview of OO and UML
  - Focus on the concepts we will use the most in this course
- More detailed description
  - Object-Oriented Design with Applications (Second Edition) by Grady Booch
  - UML Distilled – Third Edition by Martine Fowler, Addison Wesley

Classes and Objects

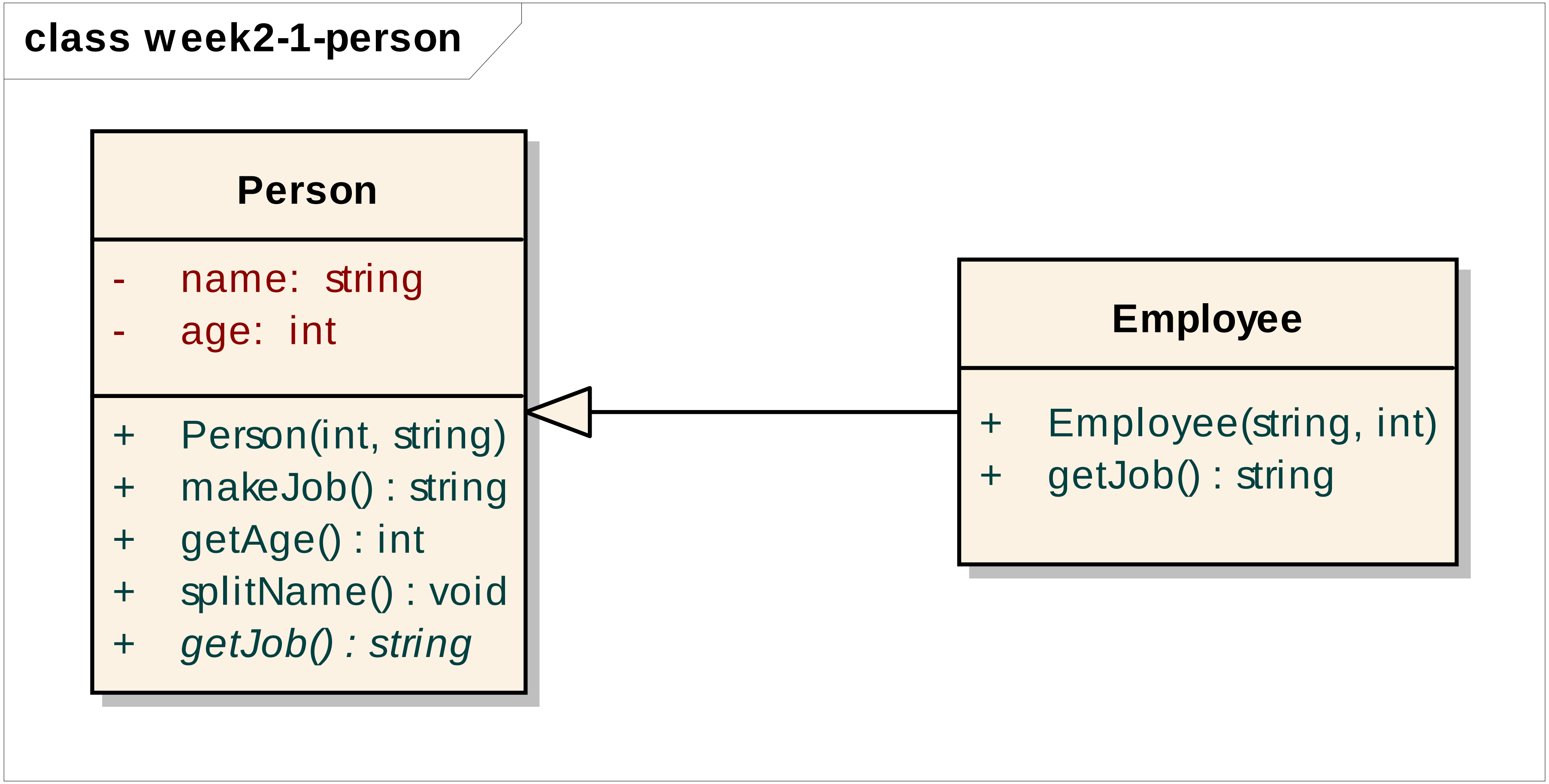
- Class: A class is a collection of things or concepts that have the same characteristics

- An Object that belongs to a particular class is often referred to as an *Instance* of that class

Inheritance

- A class inherits from another class if it receives some or all of the qualities (e.g., methods) of that class
  - Starting class: base, super, parent, generalized class
  - Inheriting class: derived, sub, child, specialized class

Inheritance Example



Inheritance Example…

- C#

```
public class Person {
        private string name;
        private int age;
        public Person(int ag, string nm){
                name=nm;
                age=ag;
        }
        public string makeJob(){
                return "hired";
        }
        public int getAge(){
                return age;
        }

        public void splitName(){
        }
        public abstract string getJob();

}
```

```
public class Employee : Person{
        public Employee(string nm, int ag){
        }
        public override string getJob(){
                return "Worker";
        }

}
```

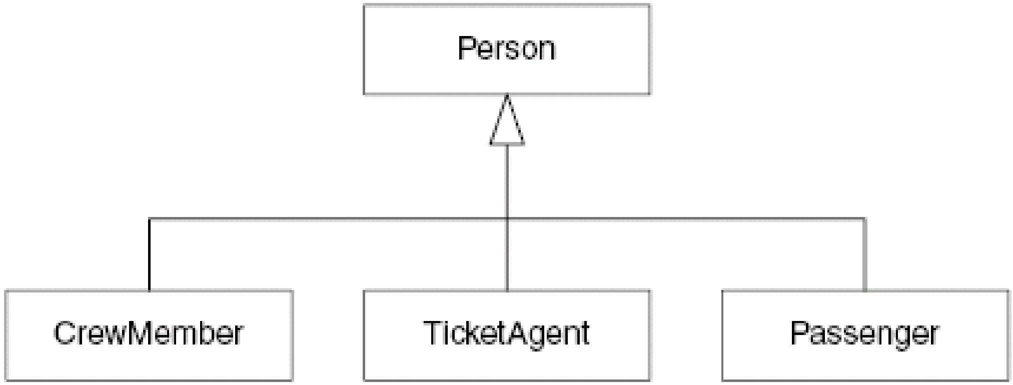
Delegation

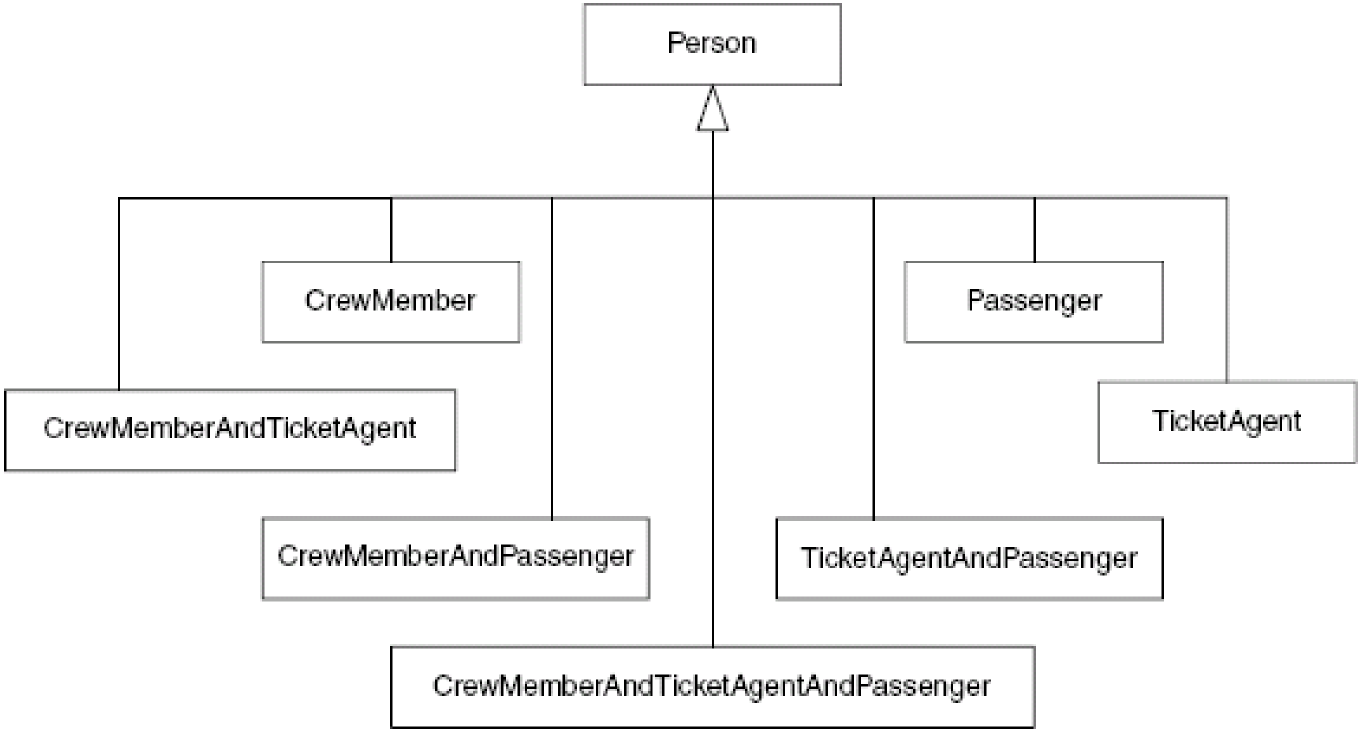
- Inheritance is a common way to extend and reuse the functionality
- Delegation is a more general way for extending a class's behavior that involves a class calling another class's methods rather than inheriting them.

Delegation…

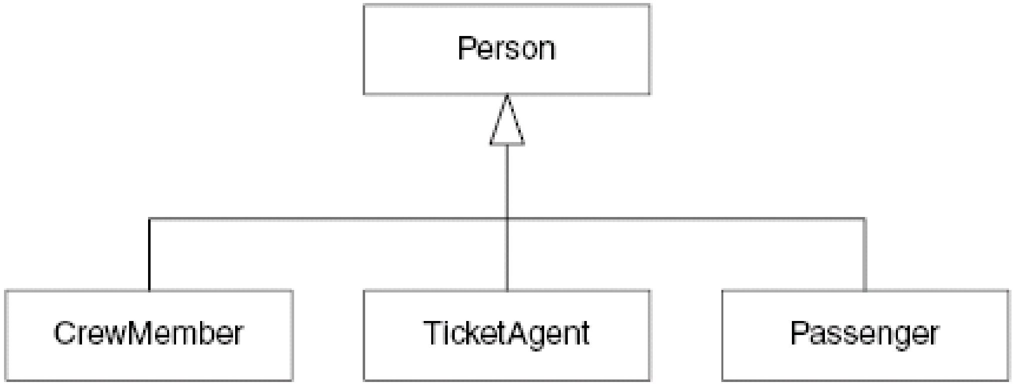
- Inheritance is useful for capturing “is-a-kind-of” relationships
- Delegation is useful for capturing “use” relationships

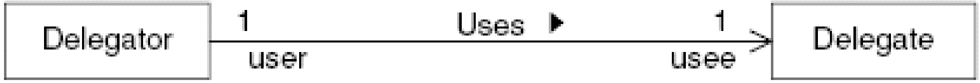
Delegation: Example

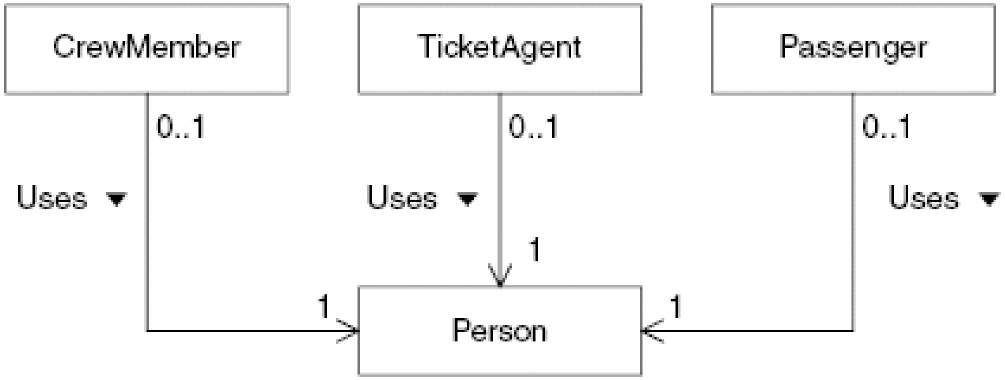




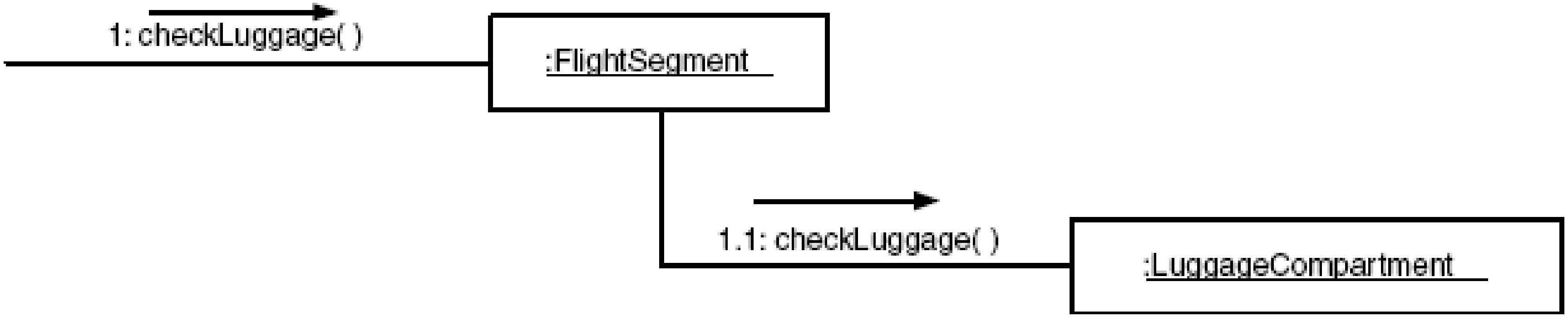
Delegation: Example







Delegation: Example



FlightSegment Class delegates the CheckLuggage operation to LuggageComparment Class

```
class FlightSegment {
            LuggageCompartment luggage;
            void checkLuggage(Luggage piece) {
                        luggage.checkLuggage(piece);
            } // checkLuggage(Luggage)
}
```

Visibility

- For data and methods: the principle of data hiding.
- Public: anything can see it
- Protected: only objects of this class and derived classes can see it
- Private: only object from this class can see it

Public and Private Derivation

Member Access in Parent: Private ‘-’	Member Access in Parent: Protected ‘#’	Member Access in Parent: Public ‘+’
•Always private regardless of derivation access	•Private in derived class if you use private derivation •Protected in derived class if you use protected derivation •Protected in derived class if you use public derivation	•Private in derived class if you use private derivation •Protected in derived class if you use protected derivation •Public in derived class if you use public derivation

Abstract and Concrete Class

- Abstract Class: defines what a set of related classes can do
  - Act as a placeholder for other classes
- Concrete Class: a class that implements a particular type of behavior for an abstract class
  - Derived from abstract classes

Signatures

- In any programming language, a signature is what distinguishes one function or method from another
- In C, every function has to have a different name
- In Java, two methods have to differ in their *names* or in the *number* or *types* of their parameters
  - `foo(int i)` and `foo(int i, int j)` are different
  - `foo(int i)` and `foo(int k)` are the same
  - `foo(int i, double d)` and `foo(double d, int i)` are different
- In C++, the signature also includes the *return type*
  - But not in Java!

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- Ability to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to

- There are two kinds of polymorphism:
  - Overloading
    - Two or more methods with different signatures
  - Overriding
    - Replacing an inherited method with another having the same signature

Overloading

In Java

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) { // same name, different parameters
        System.out.println("double d = " + d);
    }
}
```

```
int i = 5
double d = 5.0
```

Inheritance Override

- C#

```
public class Person {
        private string name;
        private int age;
        public Person(int ag, string nm){
                name=nm;
                age=ag;
        }
        public string makeJob(){
                return "hired";
        }
        public int getAge(){
                return age;
        }

        public void splitName(){
        }
        public abstract string getJob();

}
```

```
public class Employee:Person{
        public Employee(string nm, int ae){
        }
        public override string getJob(){
                return "Worker";
        }

}
```

Software Model Overview

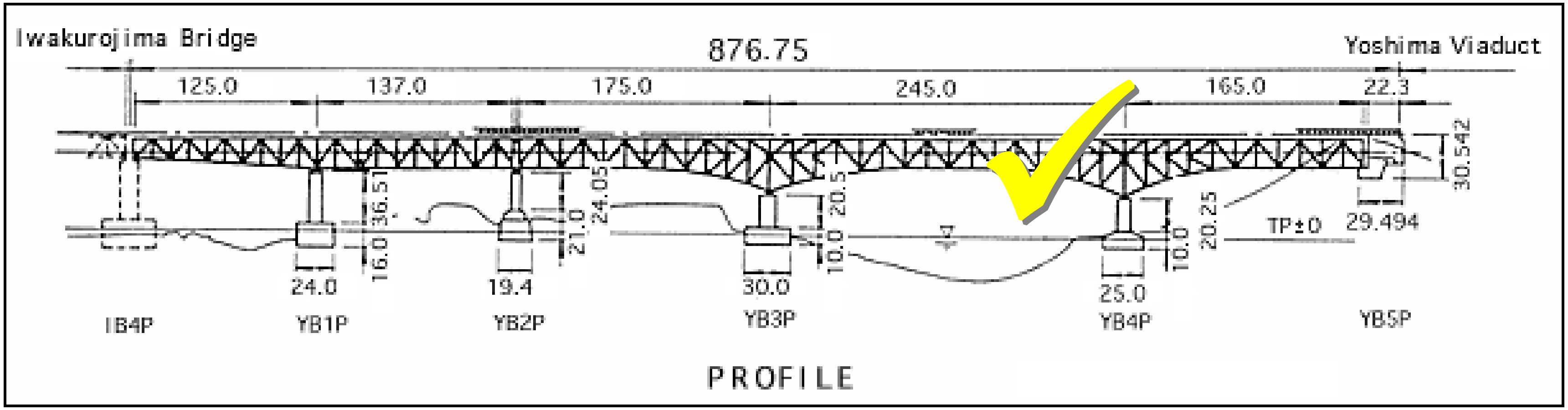
- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

What is modeling?

- Before they build the real thing...

…they first build models

...and then learn from them



What is a Software Model?

- A description (specification) of the software which
  - Abstracts out irrelevant detail
  - Presents the software using higher-level abstractions

Why do we *model*?

- It is cheaper to mess up a large project model than it is to mess up the actual product.
- To comprehend complex systems
  - In their entirety
- Models allow big issues to be caught early.
  - Really big issues, the costly ones.
    - Are lives at stake?
    - Recurring maintenance costs after product delivery
      - Are many times the big cost of a project
      - Support
      - Bug fixing
      - Downtime

How Models are Used?

- To detect errors and omissions in designs before committing full resources to full implementation
  - Through (formal) analysis and experimentation
  - Investigate and compare alternative solutions
- To communicate with stakeholders
  - Clients, users, implementers, testers, documenters, etc.
- To drive implementation

Characteristics of “Good” Software Models

- Abstract: Emphasize important aspects while removing irrelevant ones
- Understandable: Expressed in a form that is readily understood by observers
- Accurate: Faithfully represents the modeled system
- Predictive: Can be used to derive correct conclusions about the modeled system
- Inexpensive: Much cheaper to construct and study than the modeled system

UML Overview

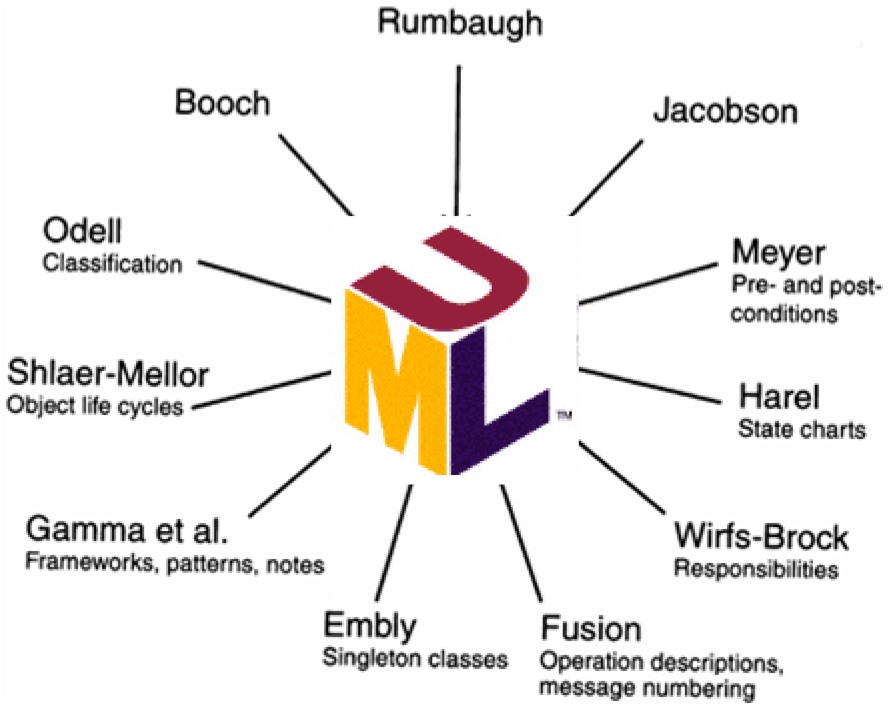
- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

What is UML?

- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.

- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- Supported by several CASE tools
  - Rational ROSE
  - Telelogic Tau
  - Enterprise Architect

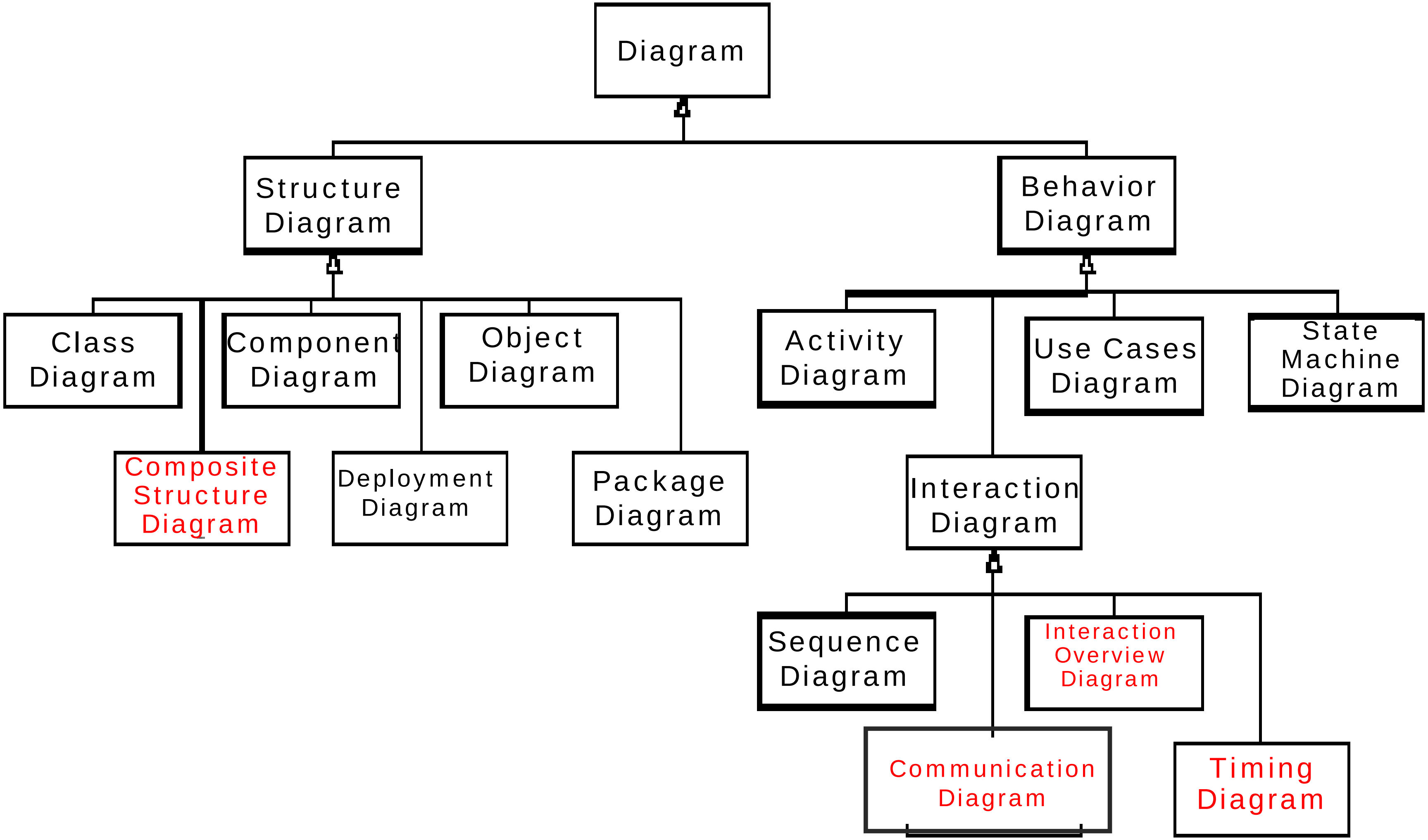
The Method Wars



UML Today

- UML notation won the Method Wars
  - An alliance of popular technologies
- UML is published and approved by the ISO as an international standard
  - ISO/IEC 19501 (UML v1.4.2)
- Current publicly available version is v2.x
- Benefits from standardization
  - Common interoperability
  - Collaboration among competitors!

UML 2.0 Diagram types



Why so many diagrams?

- A system typically has many different stakeholders
- Your objective is to communicate *clearly* with every type of stakeholder.
- Each stakeholder has his own particular interests
  - ...in specific aspects of the same system
  - ...in specific aspects of the same diagram!
- Each diagram is a different ”view” of the same model.
- Conscientious system design involves all possible viewpoints
- The more diagrams you can provide
  - the more clear the system becomes
  - the more you will be able to better defend resulting conclusions

Different Stakeholders

•Customers
•Domain Experts
•Business Analyst
•Designers
•Marketing Team
•Sales Team
•Product manager
•Programmers
•Database Administrators
•...etc.

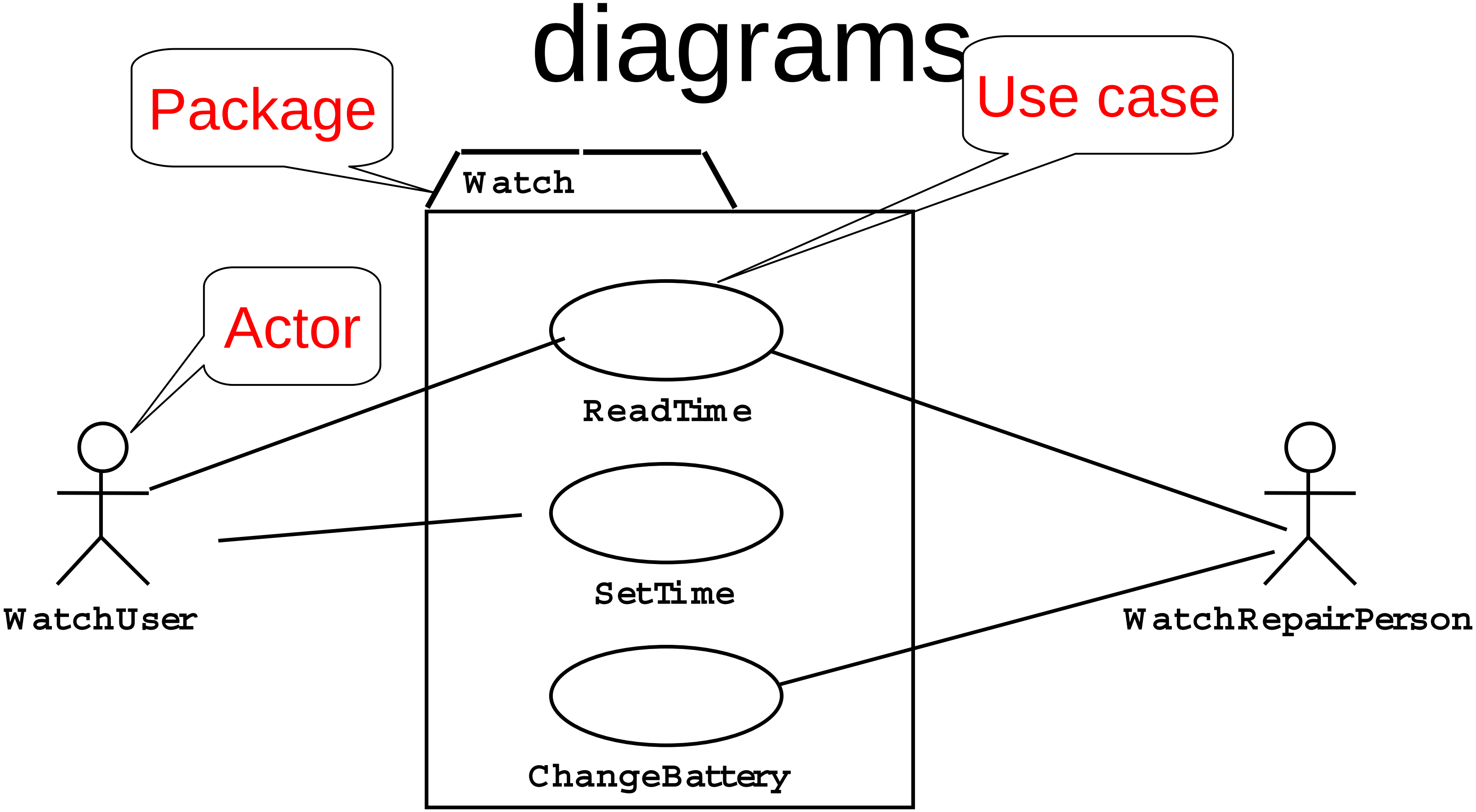
UML Diagram Classifications

- Static Diagrams (structure)
  - Class Diagram
  - Package Diagram
  - Component Diagram
  - Structure Diagram
  - Deployment Diagram
- Dynamic Diagrams (behavior)
  - Use Case Diagram
  - Interaction Diagram
  - Communication Diagram (also Collaboration Diagram in UMLv1.4)
  - Statechart Diagram
  - Activity Diagram

UML First Pass

- Use case Diagrams
  - Describe the functional behavior of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, Associations
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects of the system
- Statechart 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)

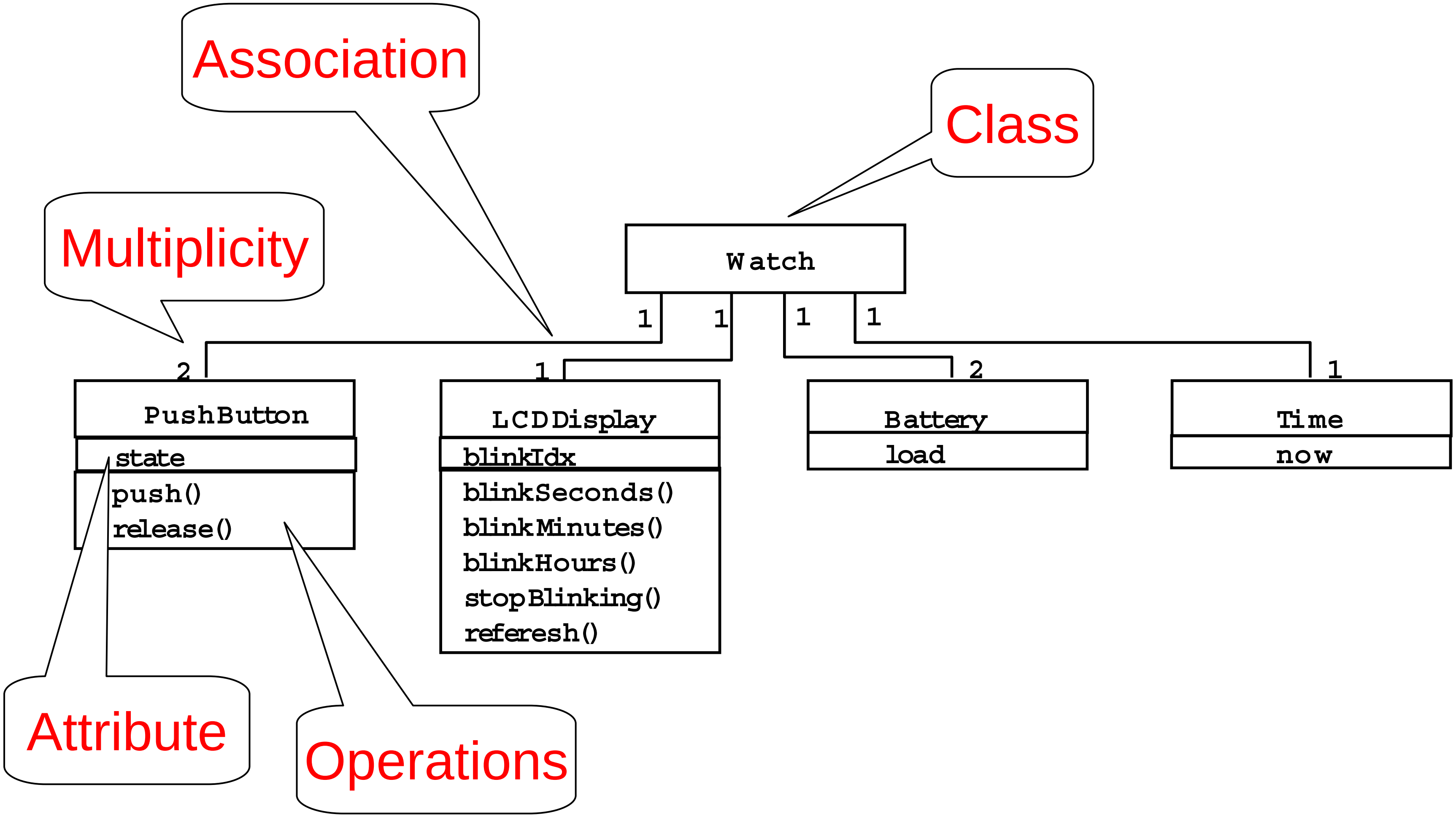
UML first pass: Use case diagrams



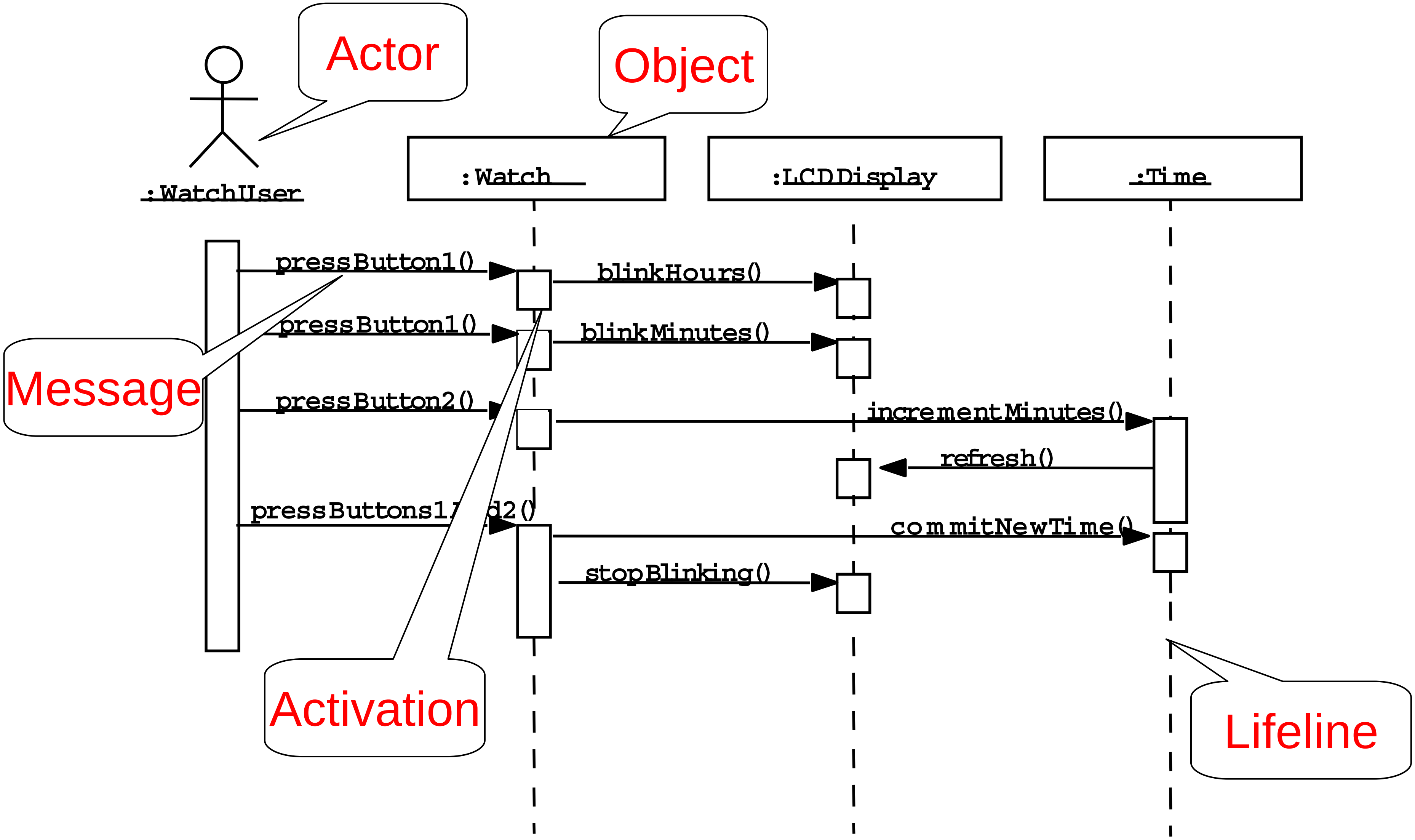
Use case diagrams represent the functionality of the system from user’s point of view

UML first pass: Class diagrams

Class diagrams represent the structure of the system

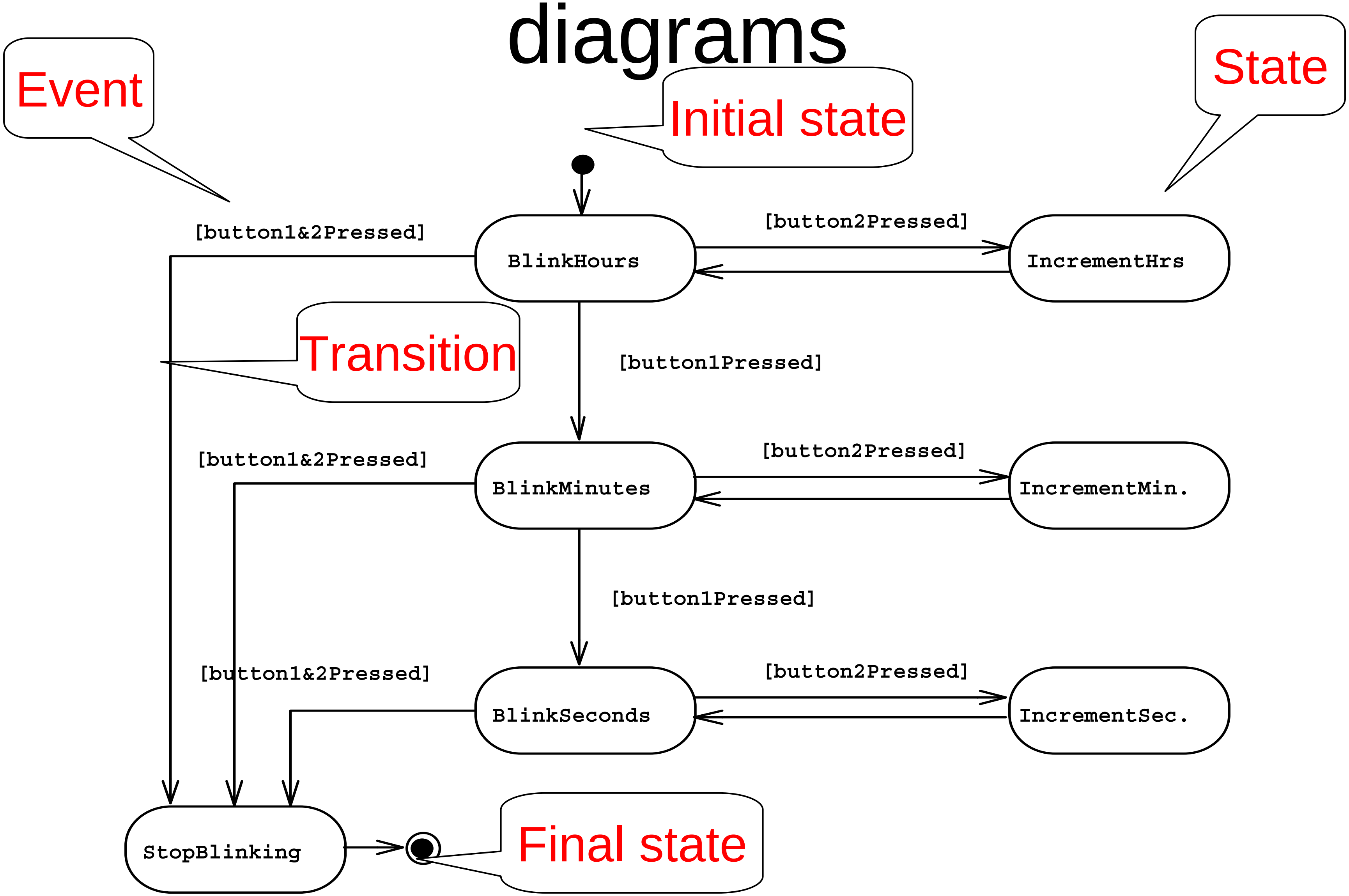


UML first pass: Sequence diagram



Sequence diagrams represent the behavior as interactions

UML first pass: Statechart diagrams

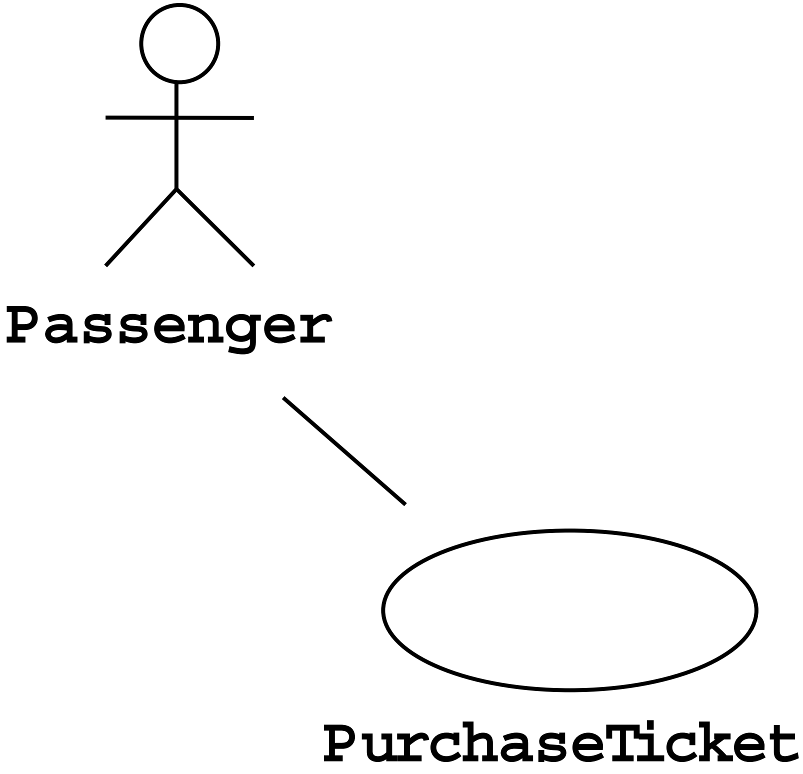


Represent behavior as states and transitions

Use Case Diagrams

- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

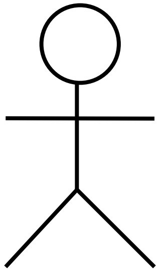
Use Case Diagrams



- 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

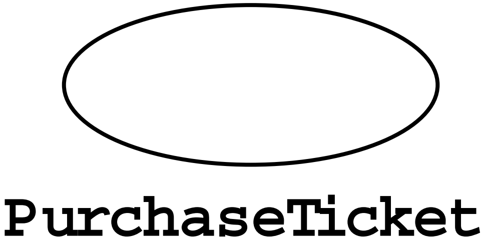
Actors



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

Use Case



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 Diagram: Example

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

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

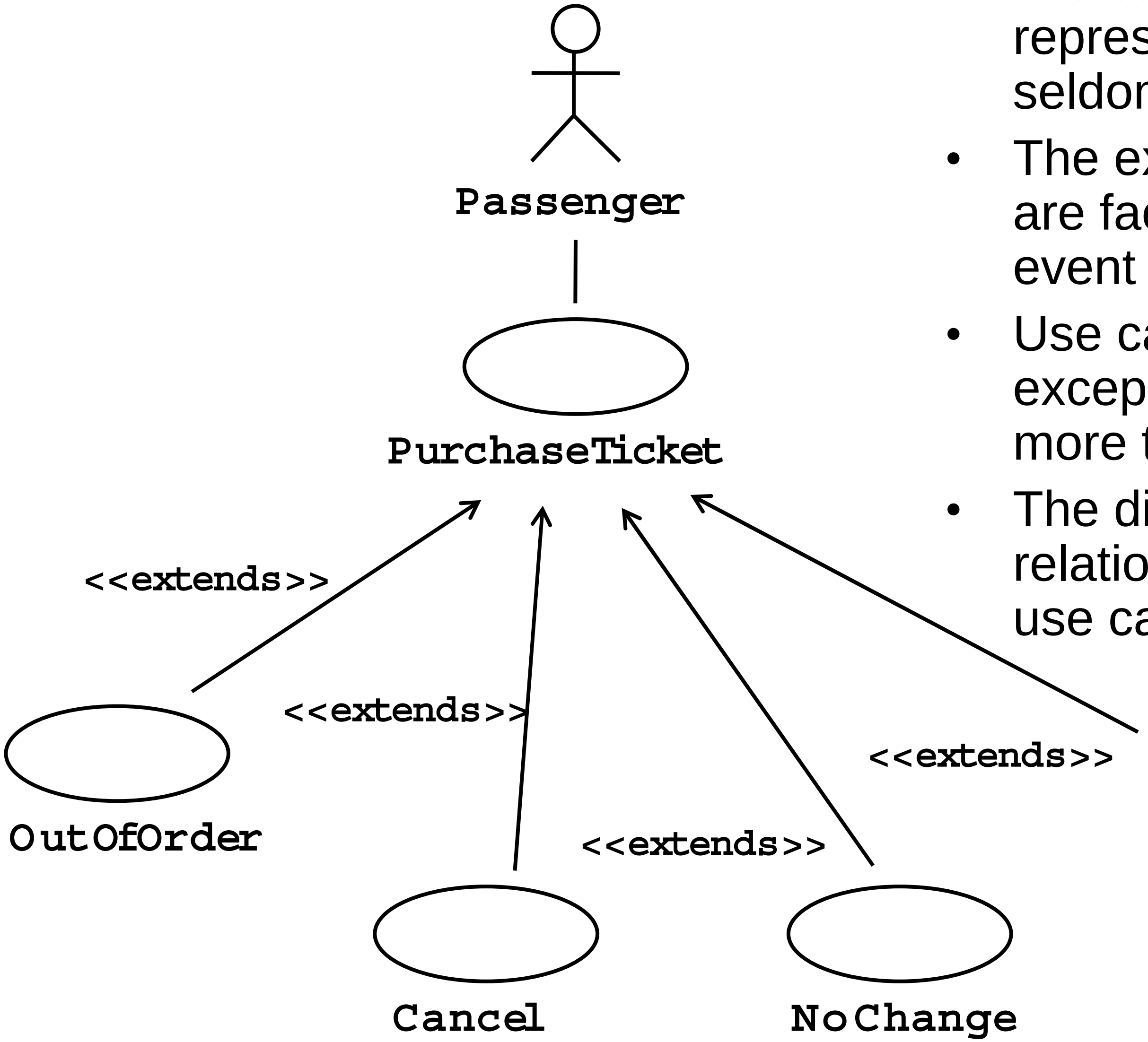
Exit condition:

- `Passenger` has ticket.

Event flow:

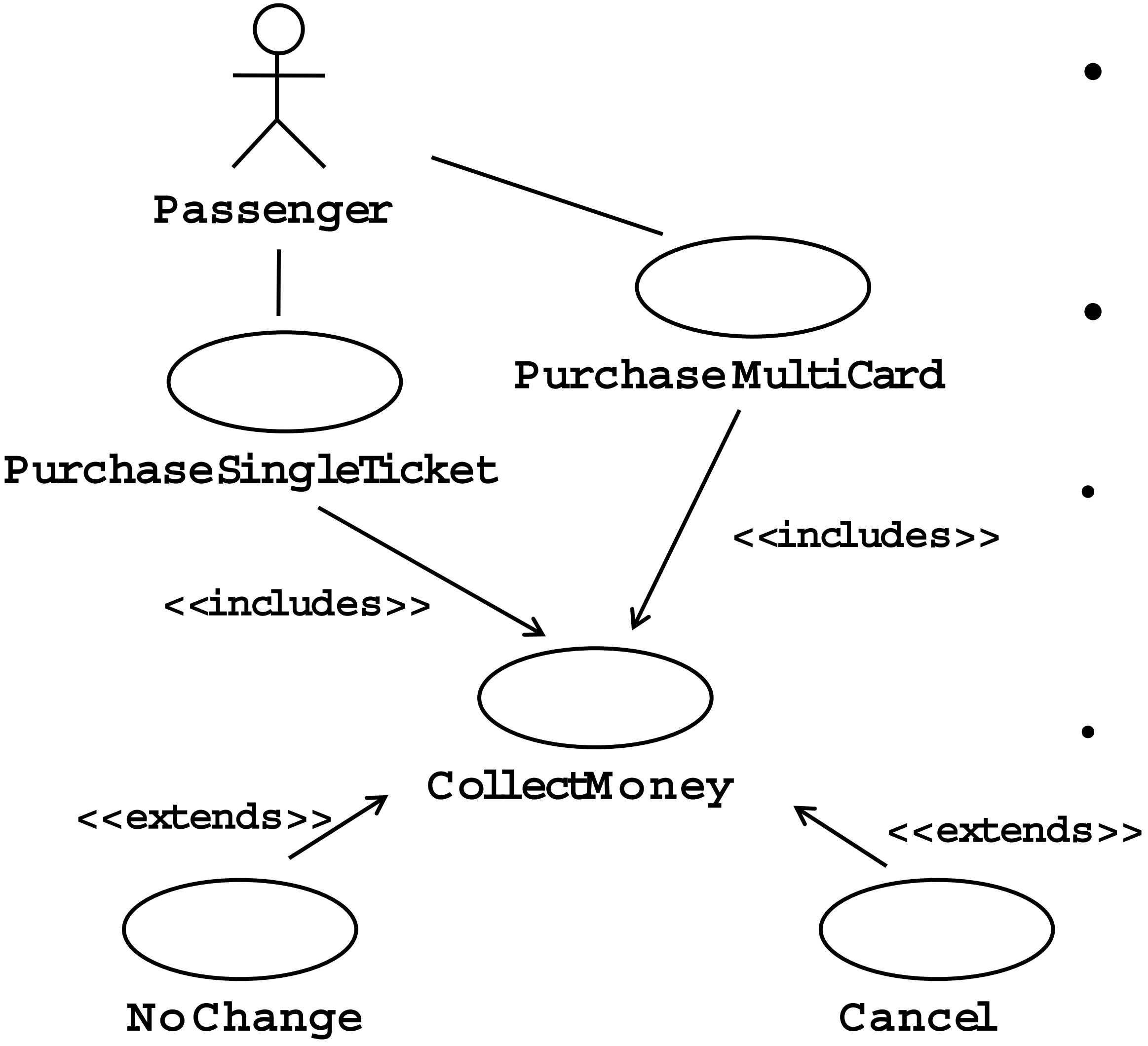
1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

The <<*extends*>> Relationship



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

The <<*includes*>> Relationship



- <<includes>> relationship represents that one use case explicitly includes the behavior of another use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).
- The Included use case doesn’t stand alone

Use Case Generalization

- Generalization Refers to a relationship between a general use case and a more specific version of that use case

- Sub use case as being a “kind of” the super use case

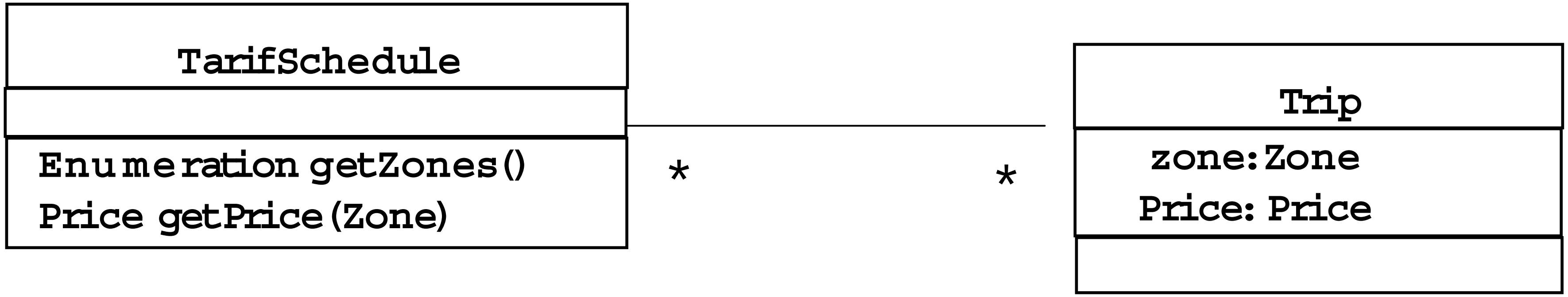
Use Case Diagrams: Summary

- Use case diagrams represent external behavior
- 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

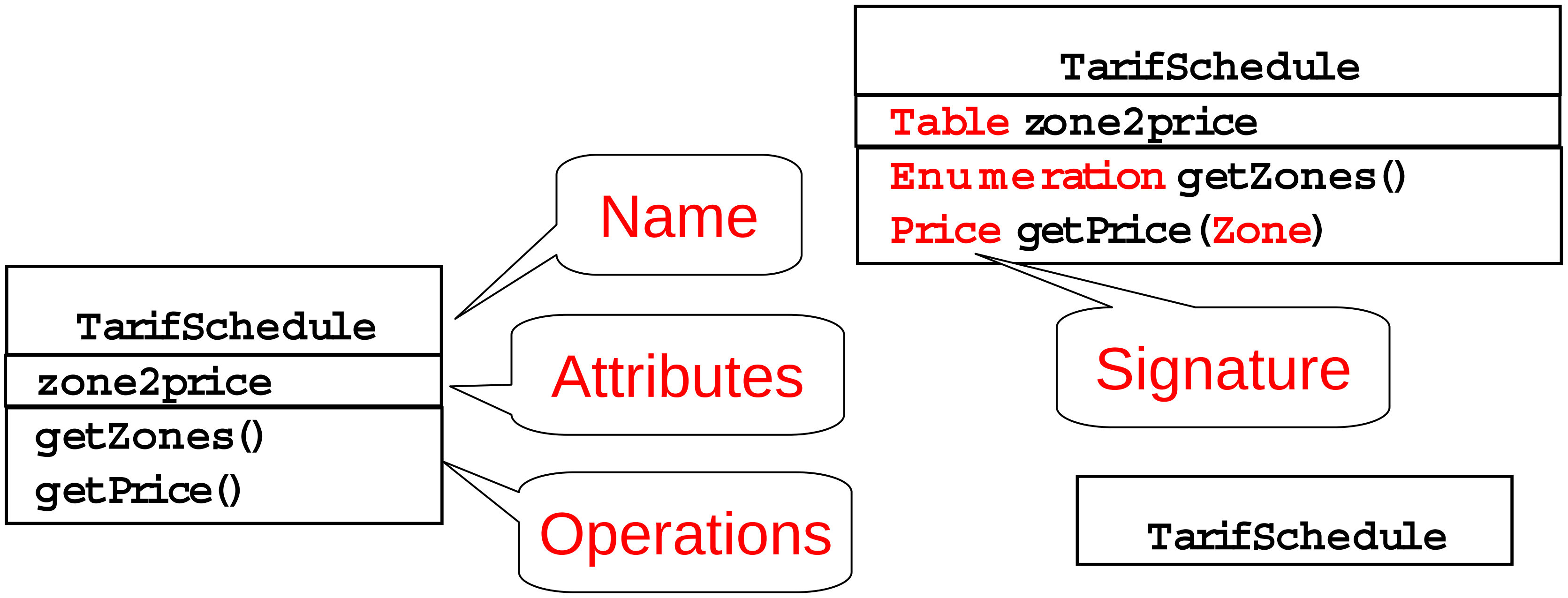
- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

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.

Classes



TarifSchedule

zone2price

getZones()

getPrice()

TarifSchedule

Table zone2price

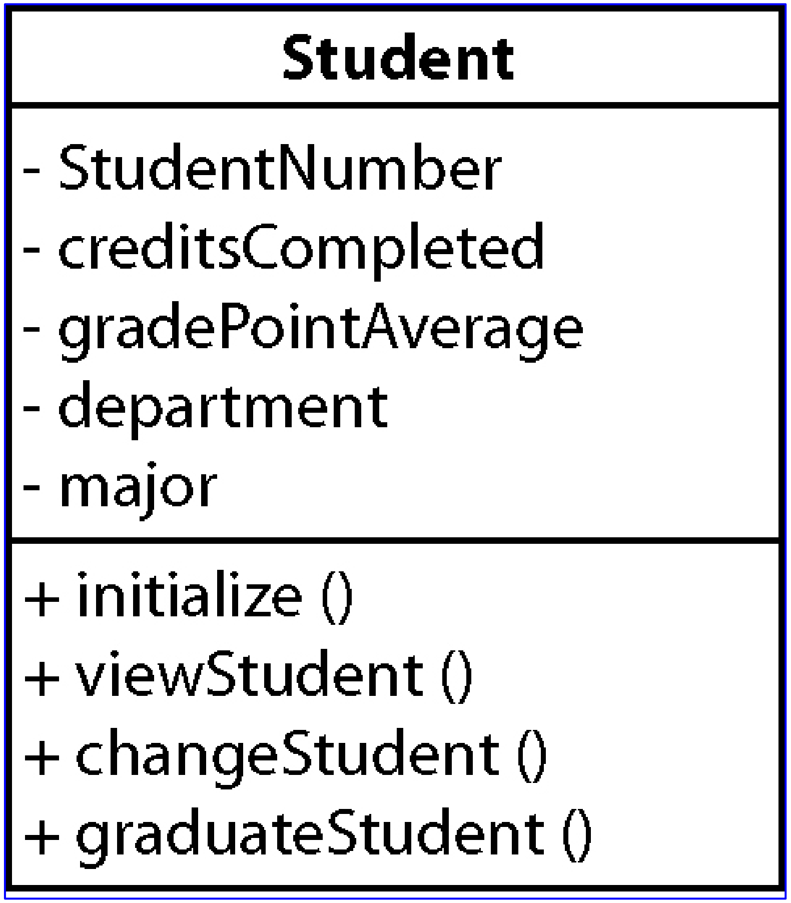
Enumeration getZones()

Price getPrice(Zone)

TarifSchedule

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

Class Visibility



Student

- StudentNumber
- creditsCompleted
- gradePointAverage
- department
- major

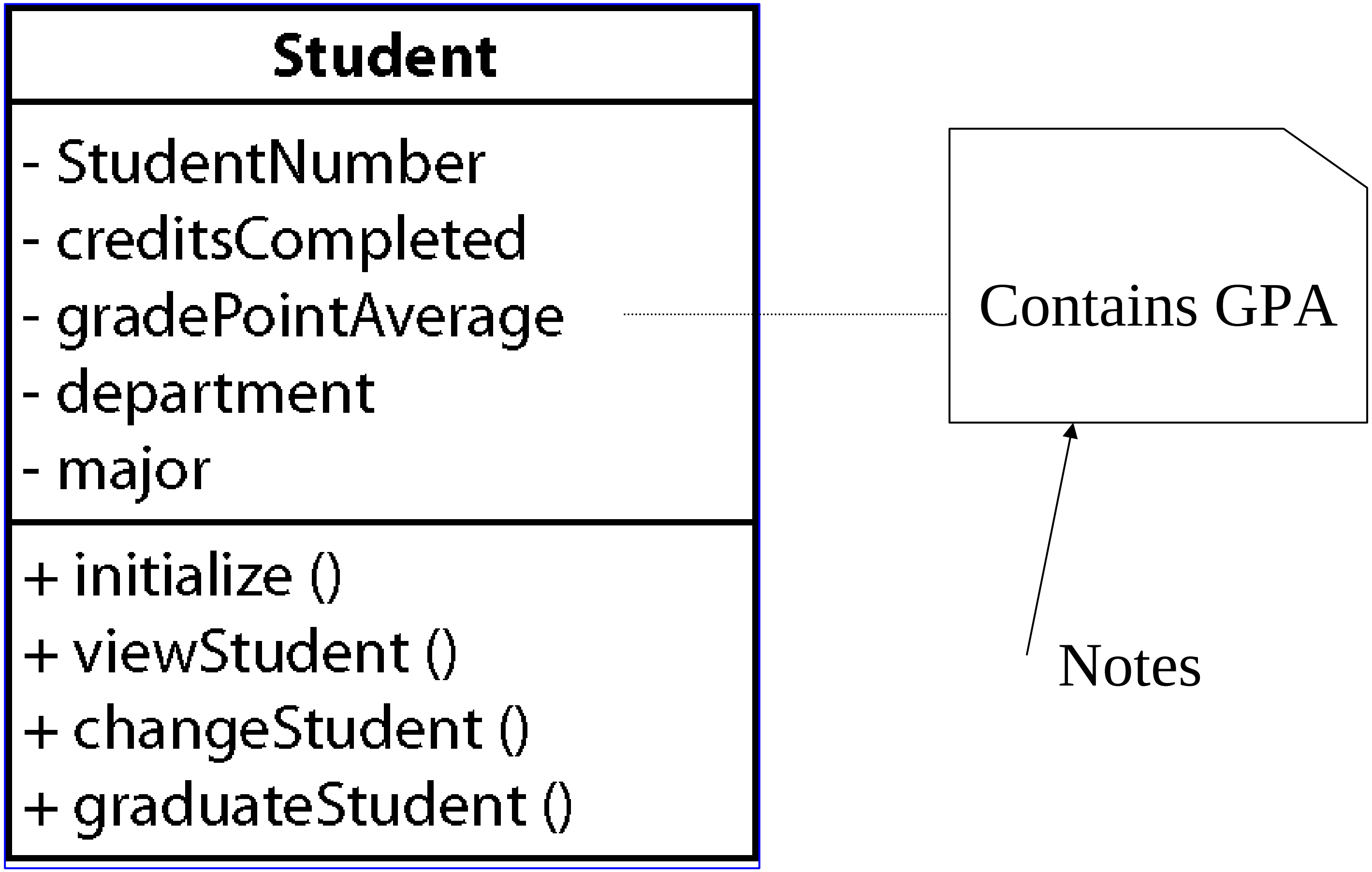
+ initialize ()
+ viewStudent ()
+ changeStudent ()
+ graduateStudent ()

+: Public

-: Private

#: Protected

Adding Notes to a UML Class



Student

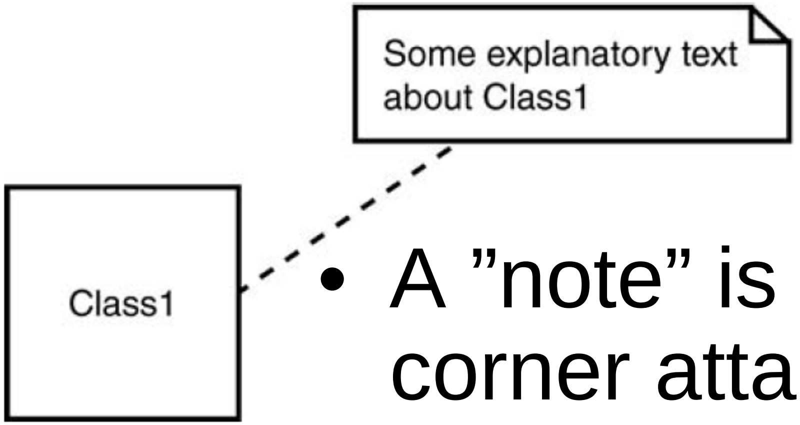
- StudentNumber
- creditsCompleted
- gradePointAverage
- department
- major

+ initialize ()
+ viewStudent ()
+ changeStudent ()
+ graduateStudent ()

Contains GPA

Notes

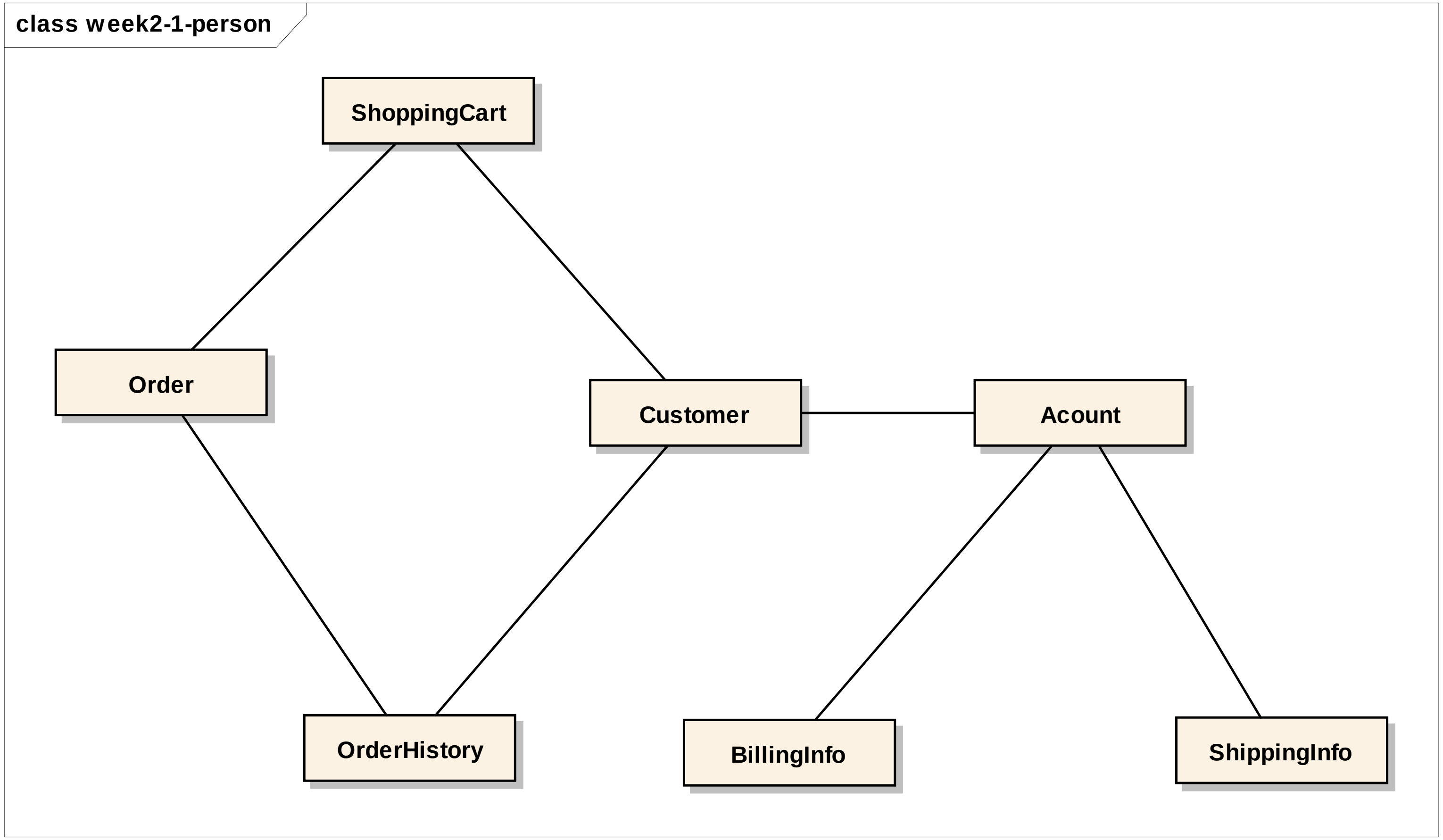
Notes



- A ”note” is a little box with a bent corner attached to some other component by a dashed line
- Notes represent additional clarification about ***any*** diagram
- Notes can be used on all diagrams
- If you want to capture a business rule, and don’t where to put it right then, make it a Note.

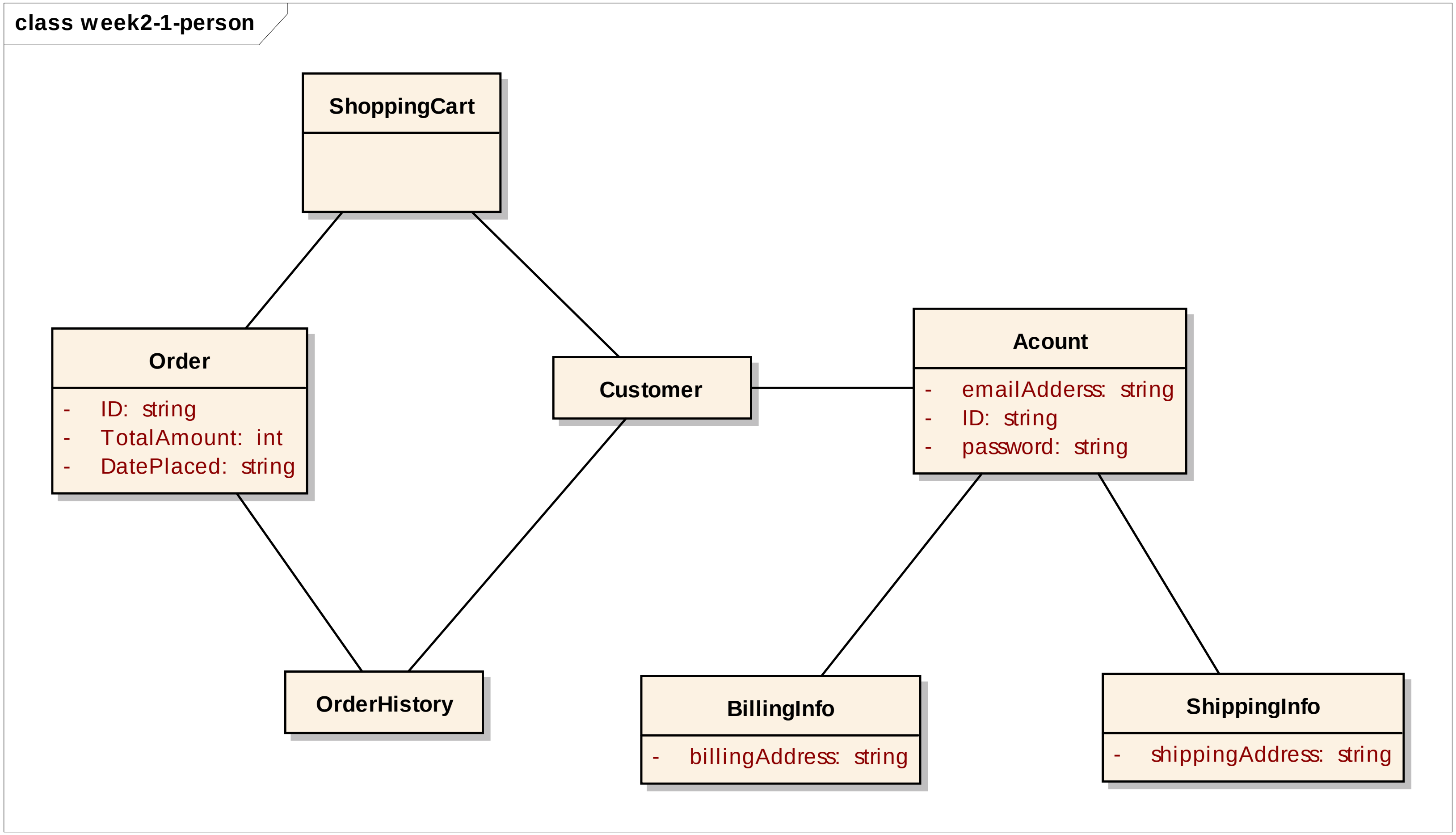
Domain-Level Class Diagrams

- Just show the names of classes
- Show part of the initial core vocabulary with which system modeling can proceed



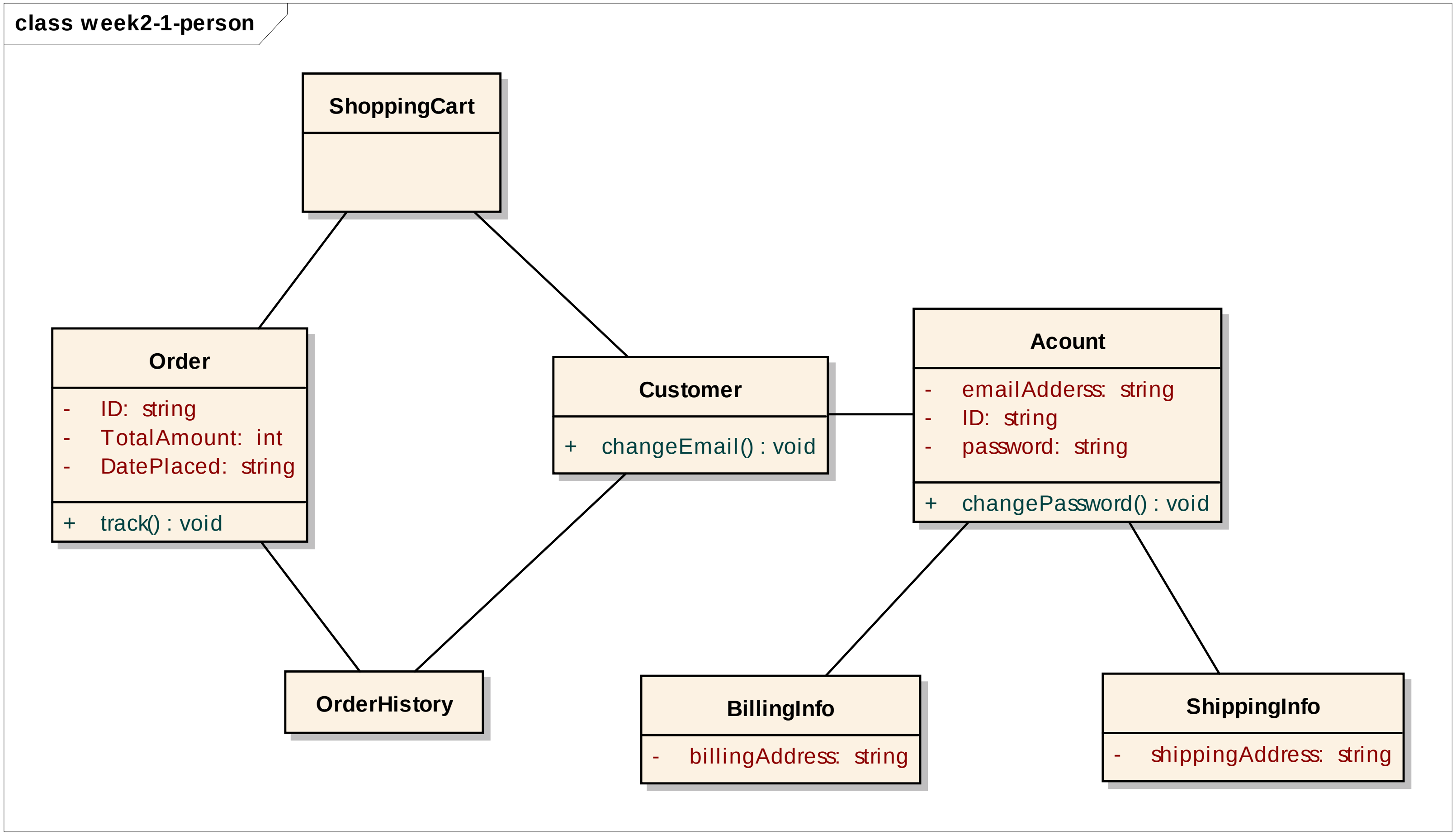
Analysis-Level Class Diagrams

- Show attributes, but not operations
- Show part of the initial core vocabulary with which system modeling can proceed



Design-Level Class Diagrams

- Contains detail info.



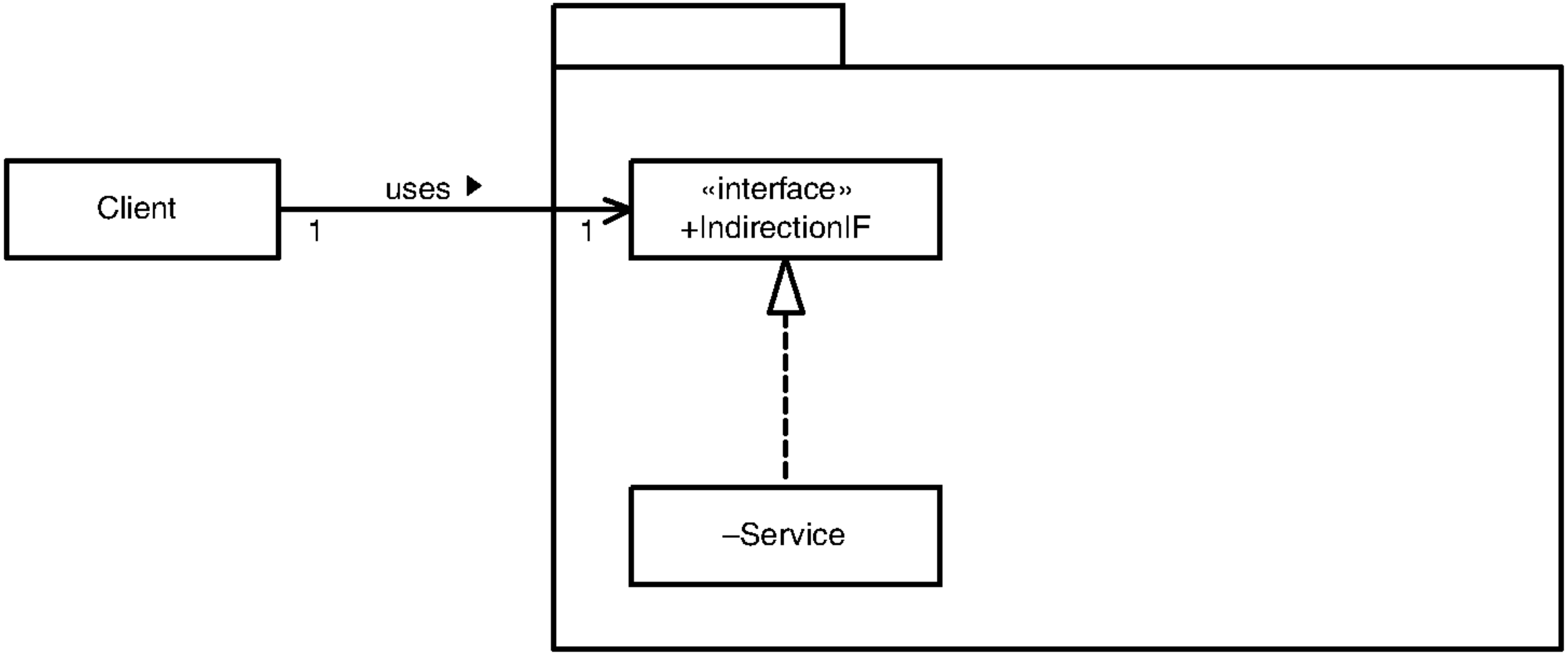
Class Interface

- An Interface is a collection of operations that represent services offered by a class. The interface specifies something like a contract that a class must adhere to.

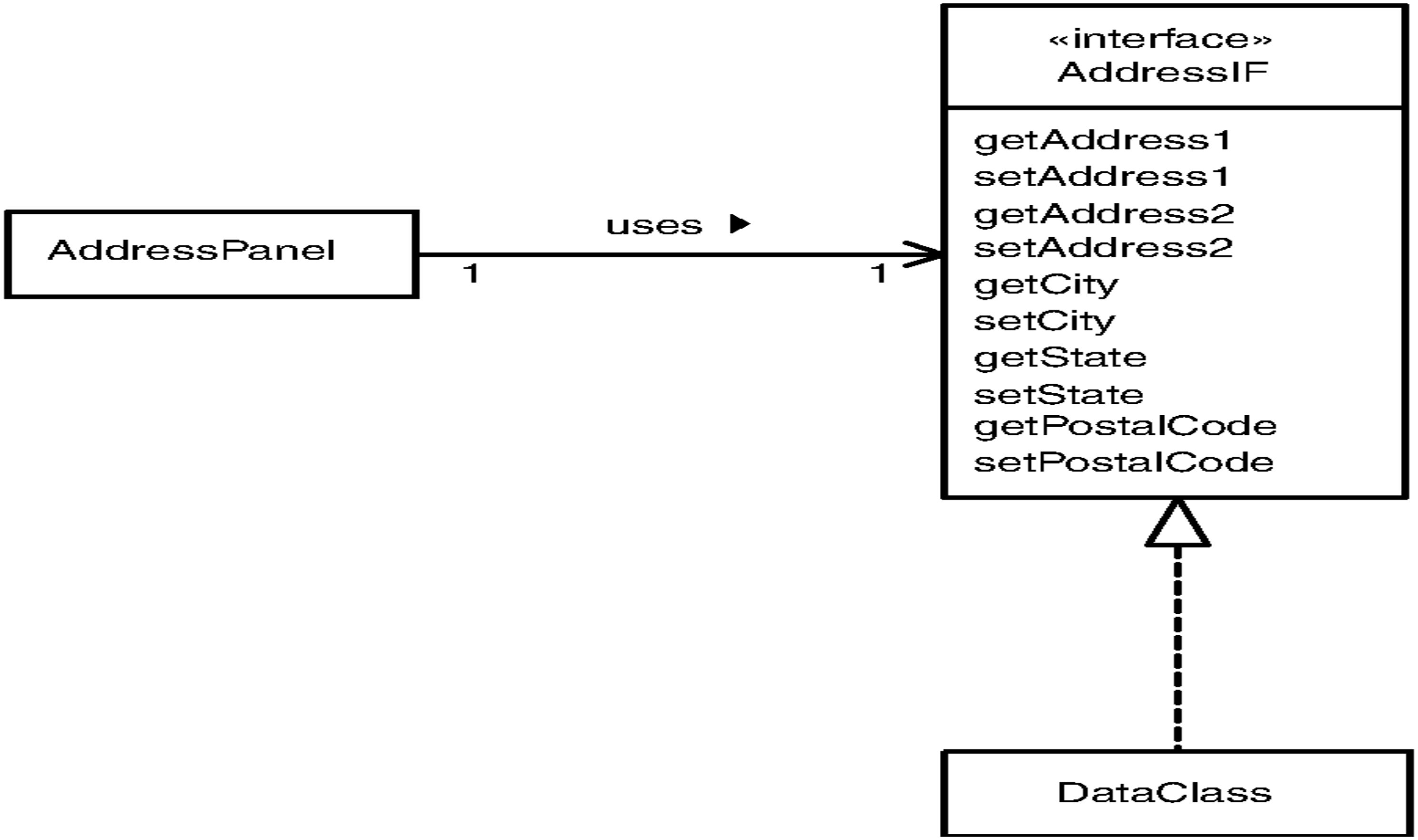
- UML defined two kinds of interfaces: Provided Interface and Required Interface
  - Provided Interfaces are interfaces that a class provides to potential clients for the operations that it offers

  - Required Interfaces are interfaces that a class needs to fulfill its duties.

Class Interface



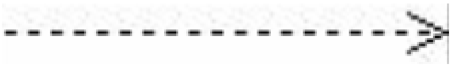
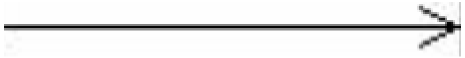
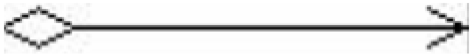
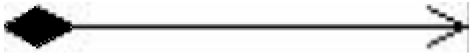
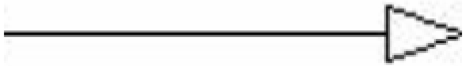
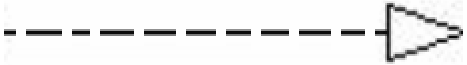
Class Interface Example



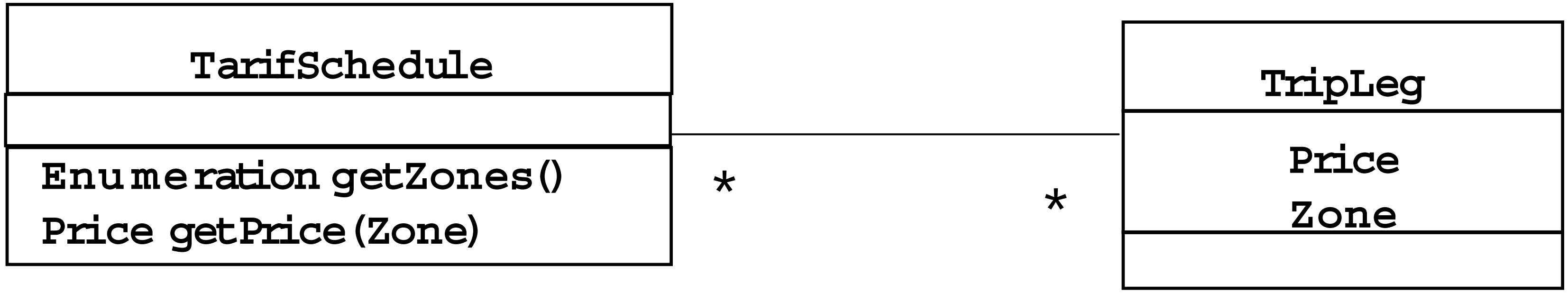
```
public interface AddressIF {
    public String getAddress1();
    public void setAddress1(String address1);
    ...
    public String getPostalCode() ;
    public void setPostalCode(String PostalCode);
} // interface AddressIF

class ReceivingLocation extends Facility implements AddressIF{
    private String address1;
    ...
    private String postalCode;
    ...
    public String getAddress1() { return address1; }
    public void setAddress1(String address1) {
        this.address1 = address1;
    } // setAddress1(String)
    ...
    public String getPostalCode() { return postalCode; }
    public void setPostalCode(String postalCode) {
        this.postalCode = postalCode;
    } // setPostalCode(String)
} // class ReceivingLocation
```

Class Diagram Relationship Summary

Relationship	<any line>	meaning
Dependency		”uses a”
Association		”has a”
Aggregation		”owns a”
Composition		”is composed of”
Generalization		”is a”
Realization		”implements”

Associations

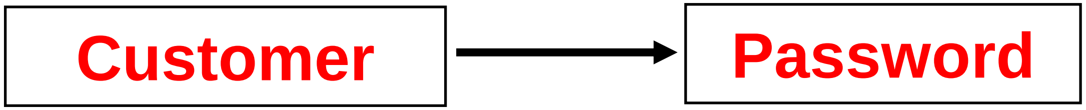


- Associations denote relationships between classes.
- You might think of an association as representing a “peer” relationship
- Instances of classes involved in an association will communicate in the real time
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

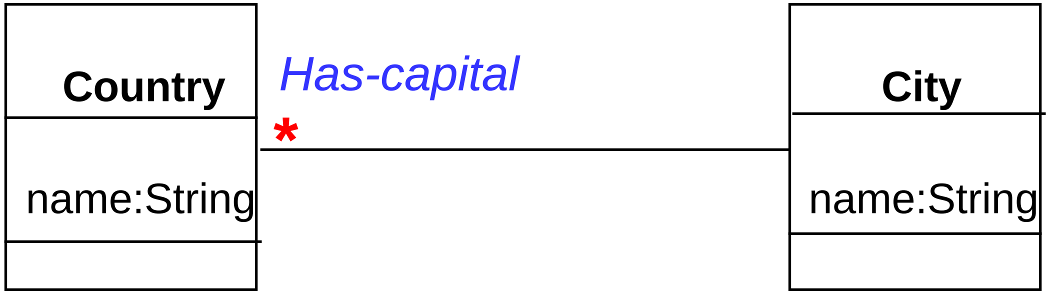
Association Direction

- An association is assumed to be bedirectional

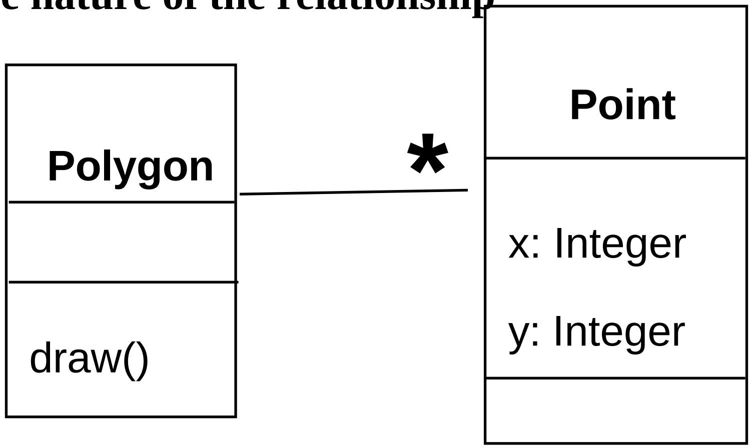
- One-way Navigation
  - Customer has access to his or her password, but no one can in turn use a Password to identify a Customer



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



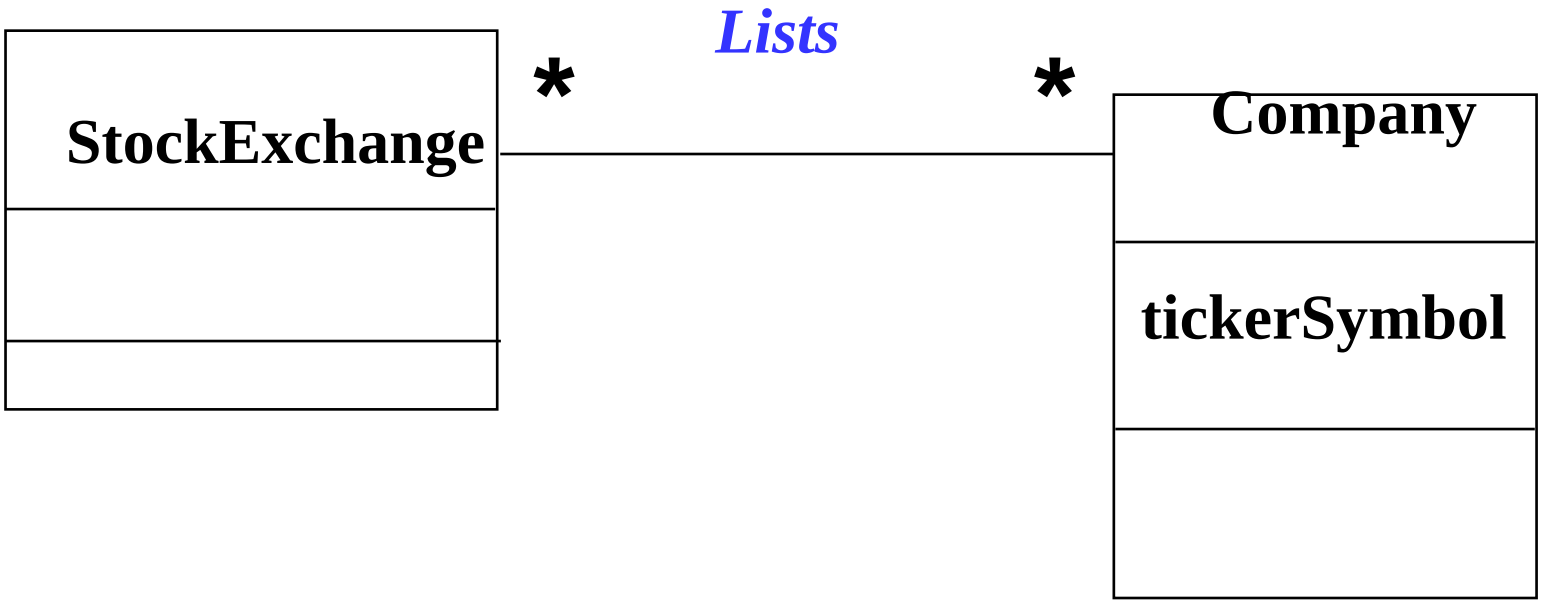
One-to-one association, **an association can have a name that indicates the nature of the relationship**



```
Public Class Polygon  {
            private Point [ ]  pnts;
            ……
}
```

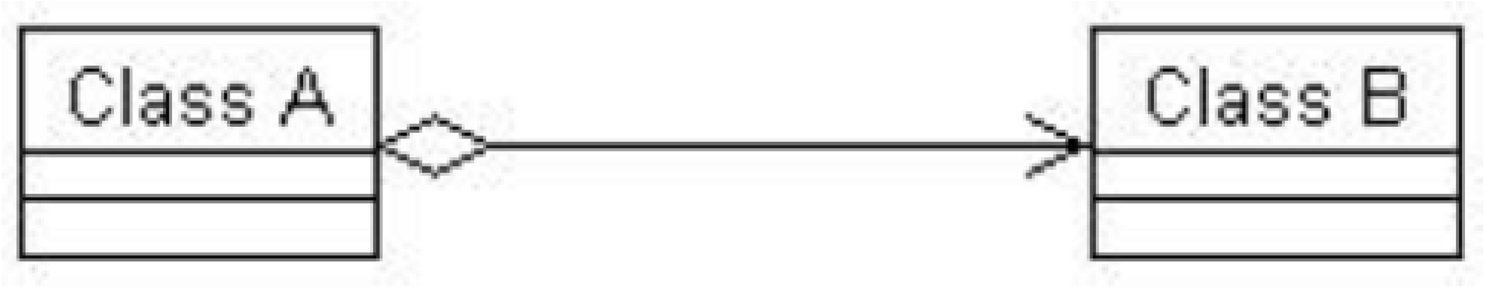
One-to-many association

Many-to-Many Associations



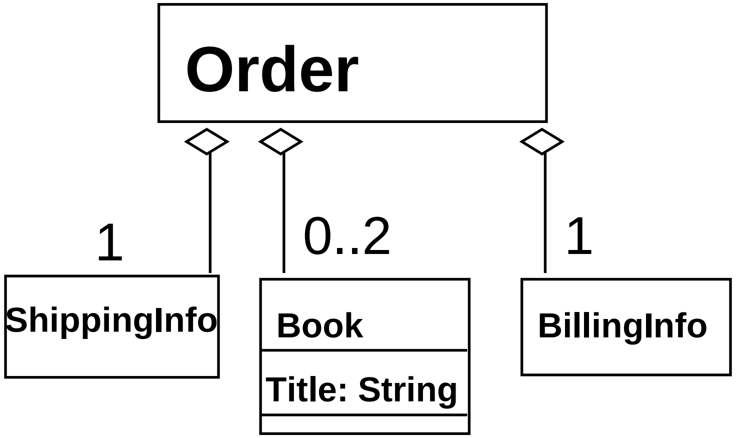
Relationship::Aggregation

- An *aggregation* indicates a whole/part relationship between two classes.
  - ”owns a”
- One component is comprised of many parts
- Destroying the whole does *not* destroy the parts



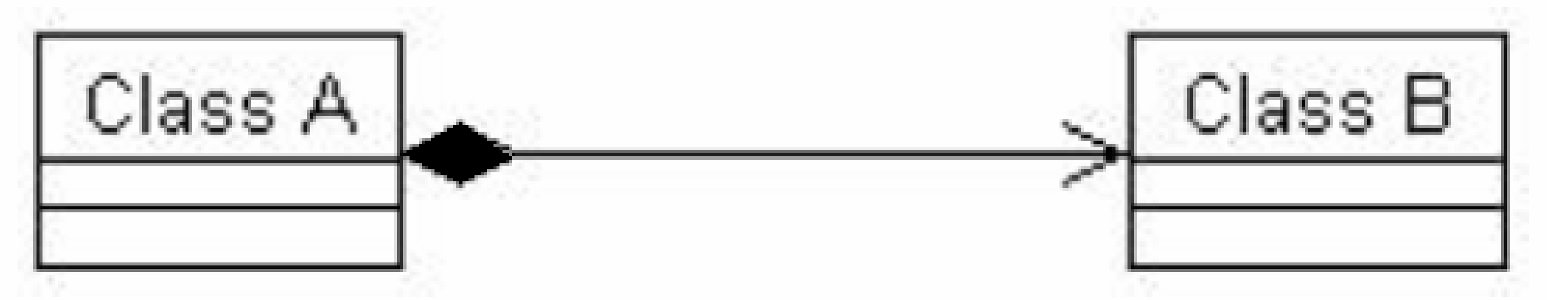
Aggregation Example

- An ***aggregation*** is a special case of association denoting a “consists of” hierarchy (Whole/Part Relationship).
- The ***aggregate*** is the parent class, the ***components*** are the children class.



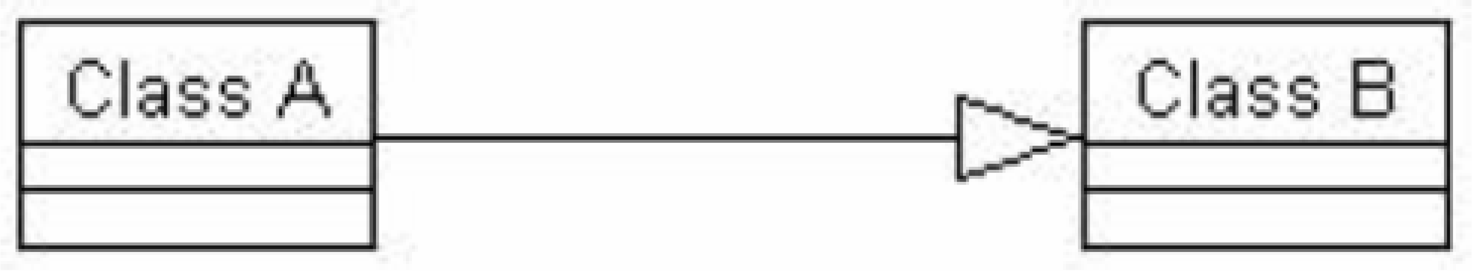
Relationship::Composition

- An *composition* is a stronger form of aggregation
- Indicates a whole/part relationship between two classes.
- One component ”is composed of” many parts
  - ”is composed of”
- Destroying the whole DOES destroy the parts

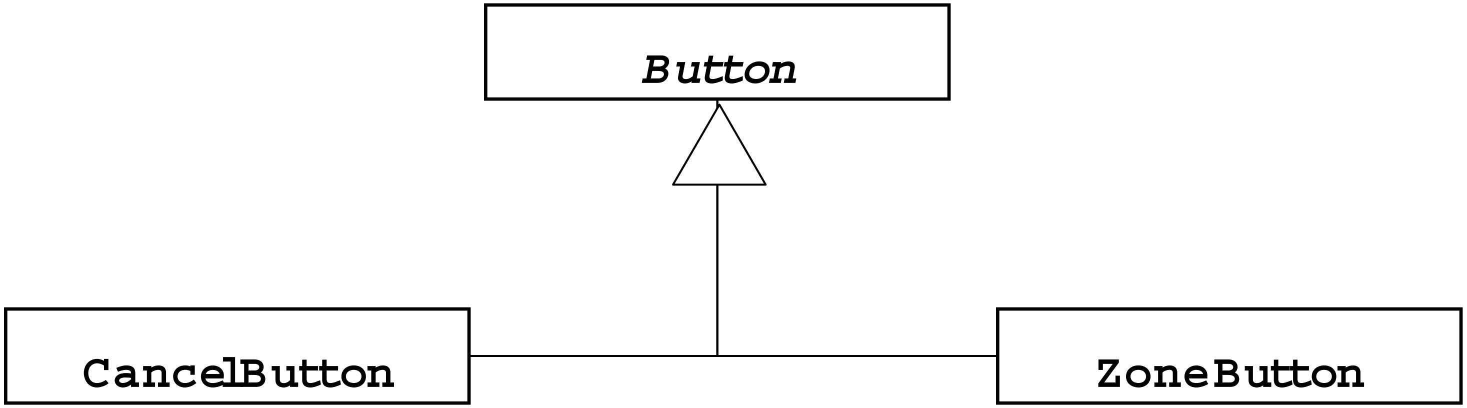


Relationship::Generalization

- A *generalization* indicates a parent/child inheritance relationship between two classes.
- Class A ”is a kind of” class B.

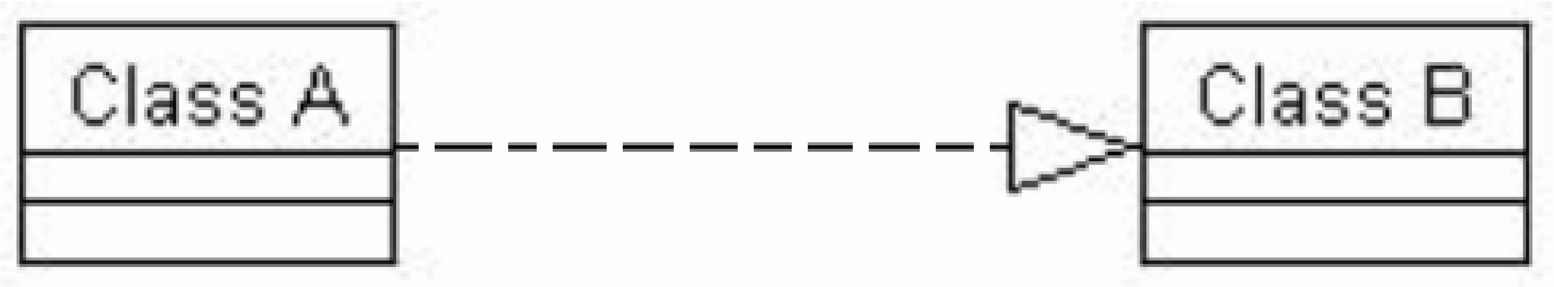


Generalization: Example



Relationship::Realization

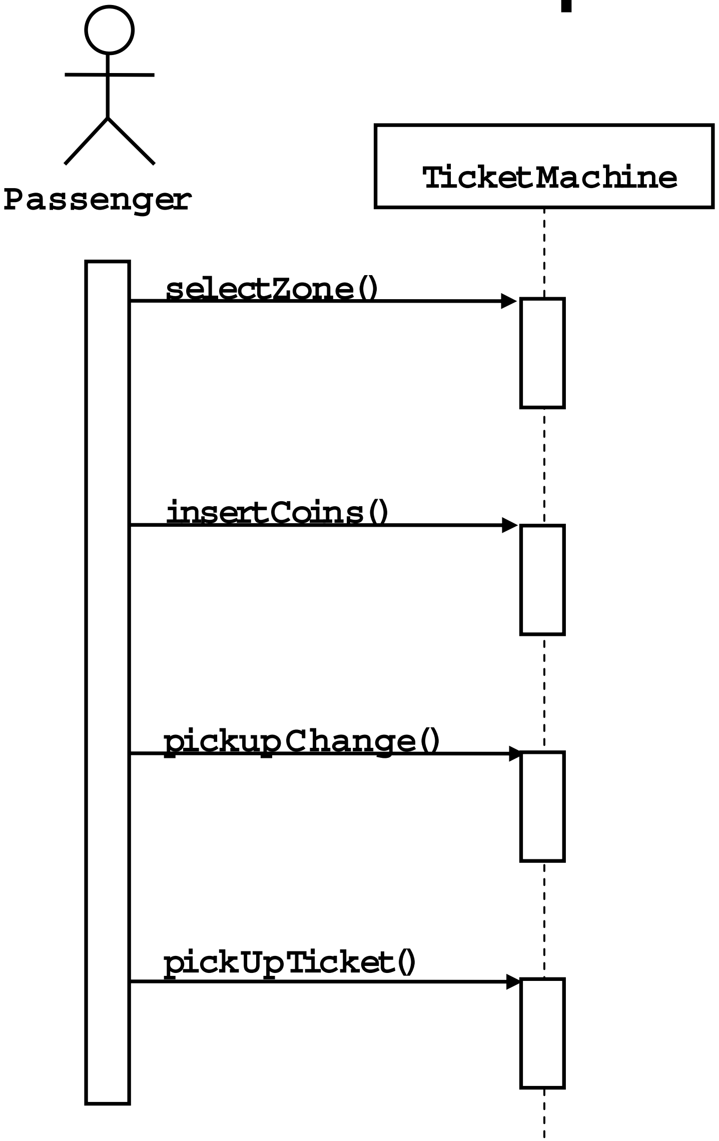
- A *realization* indicates that a class implements, or realizes, an interface on the other end.
- Realization is similar to generalization (inheritance).
- Realization indicates that the class realizing an interface is an implementation of the referenced interface.
- Interfaces define only operation signatures.
- A realization ties an interface to a concrete implementation.



Sequential Diagram

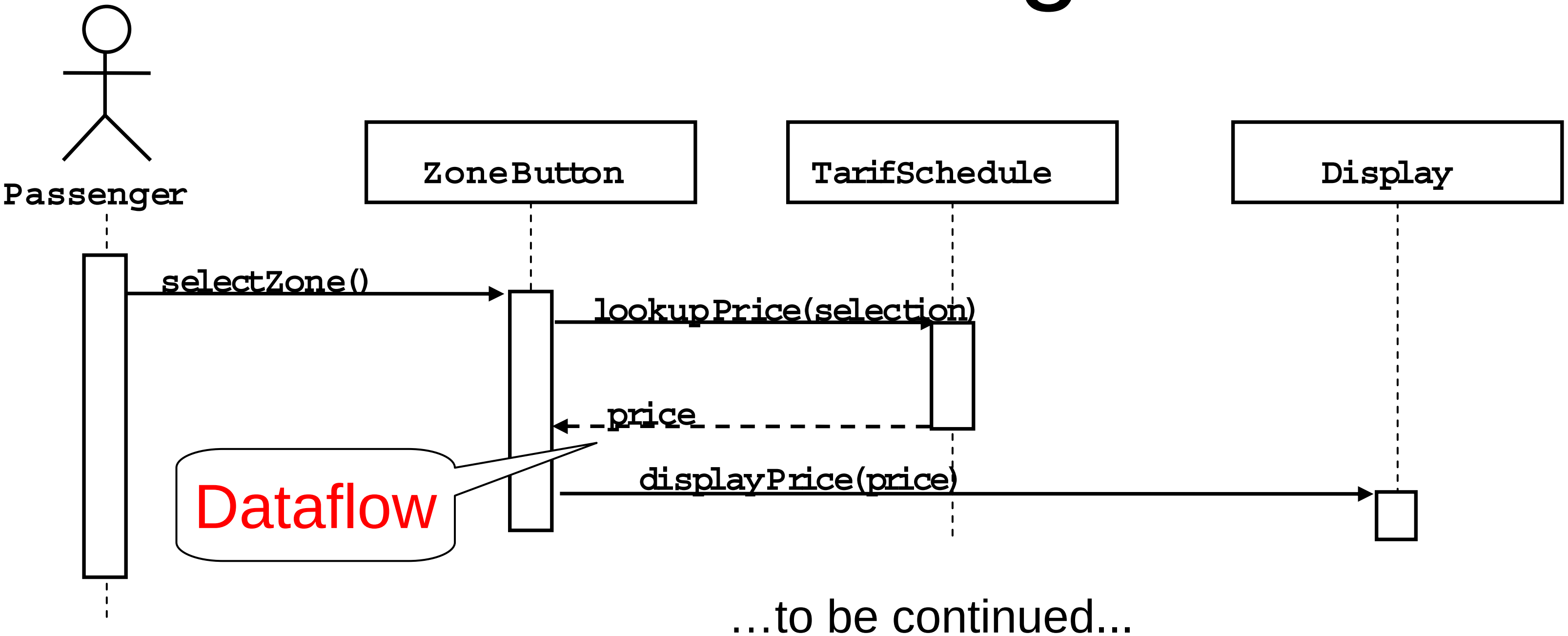
- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

UML sequence diagrams



- Focuses on the time-ordering of messages between objects
- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects (“participating objects”)
- Used during system design
  - to refine subsystem interfaces
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

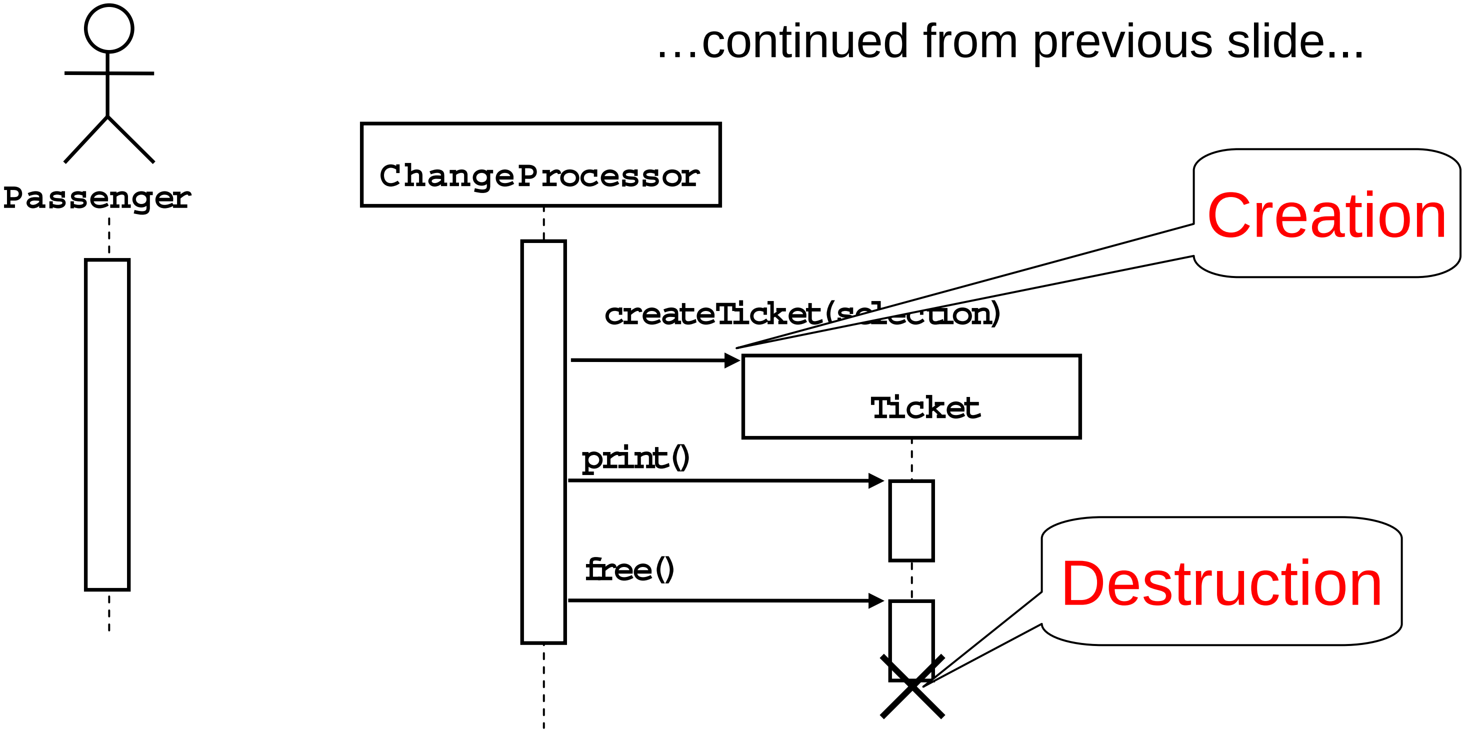
Nested messages



…to be continued...

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

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.

Sequence Diagram 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).

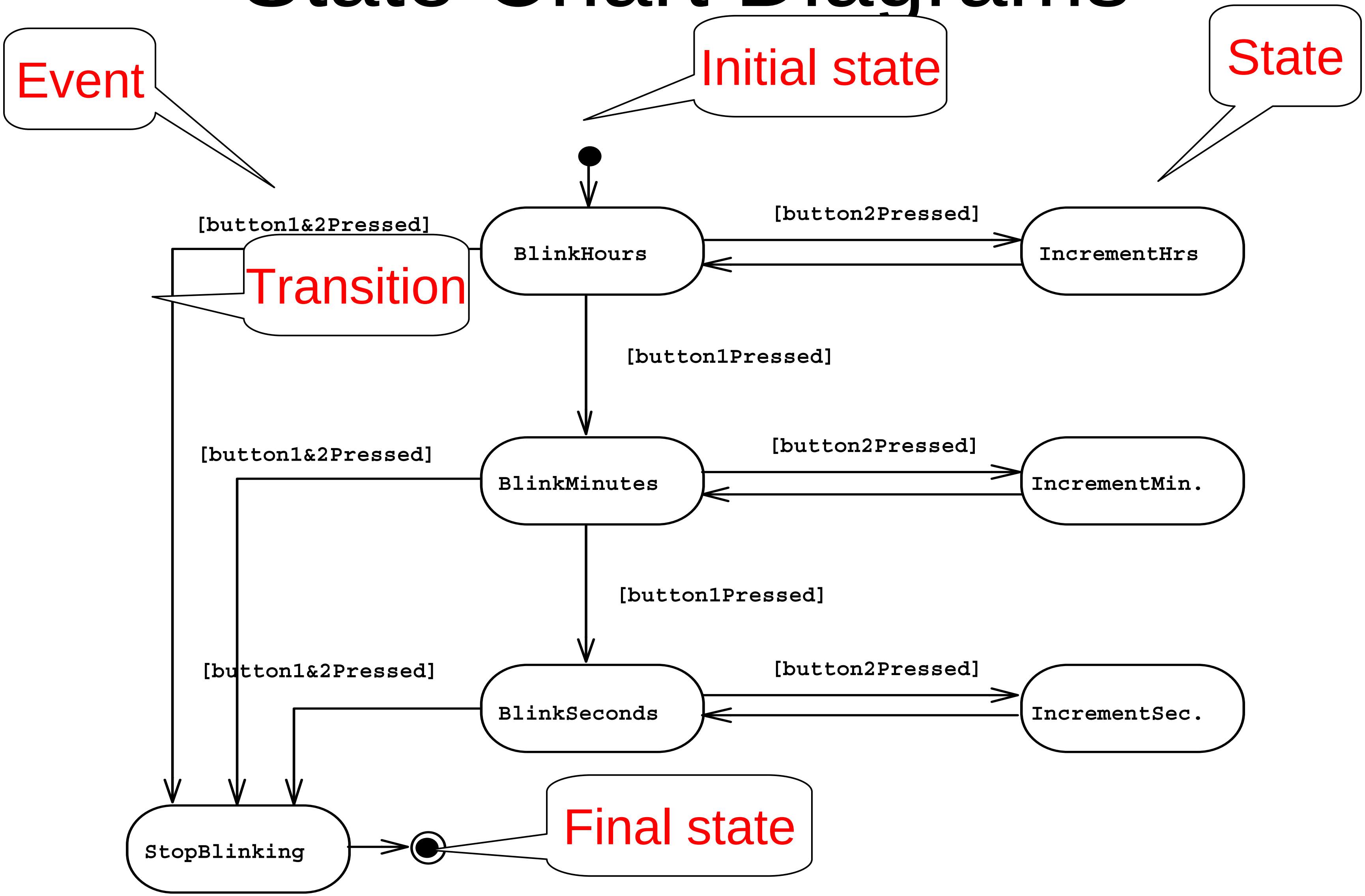
Statechart Diagrams

- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

Statechart Diagrams

- A Statechart is a notation for describing the sequence of states an object goes through in response to external events
- A State is a condition satisfied by the attributes of an object. It depicted by a rounded rectangle
  - An Incident object has four states: Active, Inactive, Closed and Archived.
- A transition represents a change of state triggered by events and is depicted by open arrows connecting two states
- A Small solid black circle indicates the initial state
- A circle surrounding a small solid black circle indicates a final state

State Chart Diagrams



Represent behavior as states and transitions

Statechart Diagrams Summary

- Statechart diagrams are used to represent nontrivial behavior of a subsystem or an object

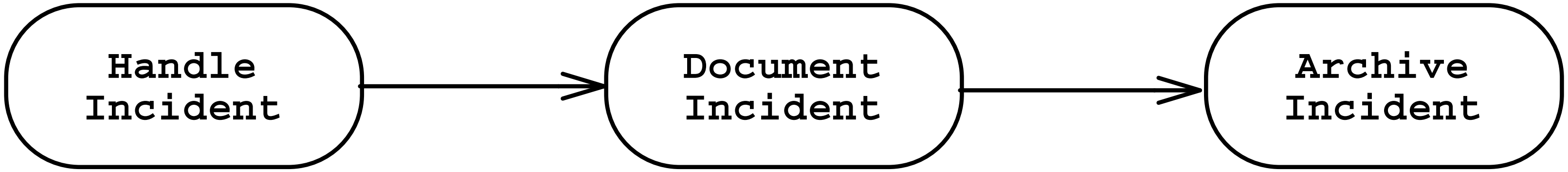
- What is the difference between interaction diagrams and Statechart Diagrams
  - Statechart diagrams make explicit which attribute or set of attributes have an impact on the behavior of a *Single* object
  - Interaction Diagrams are used to identify participating objects and services they provide.

Activity diagrams

- The Object-Oriented (OO) Paradigm
  - Inheritance
  - Visibility
  - Signature
  - Polymorphism
    - Overload
    - overwrite
  - delegation
- Software Model
- UML (Unified Modeling Language)
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

Activity Diagrams

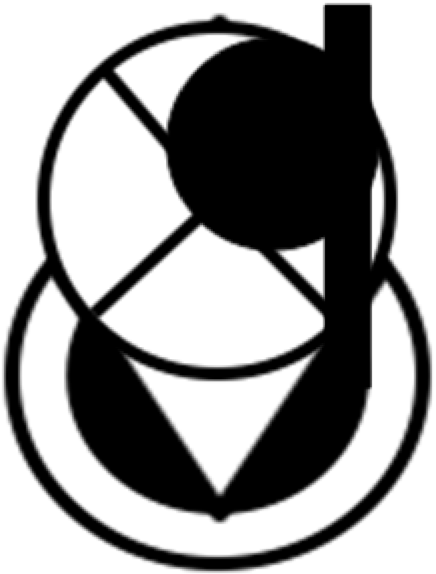
- An activity diagram shows flows of control among activities and actions associated with a particular object or set of objects



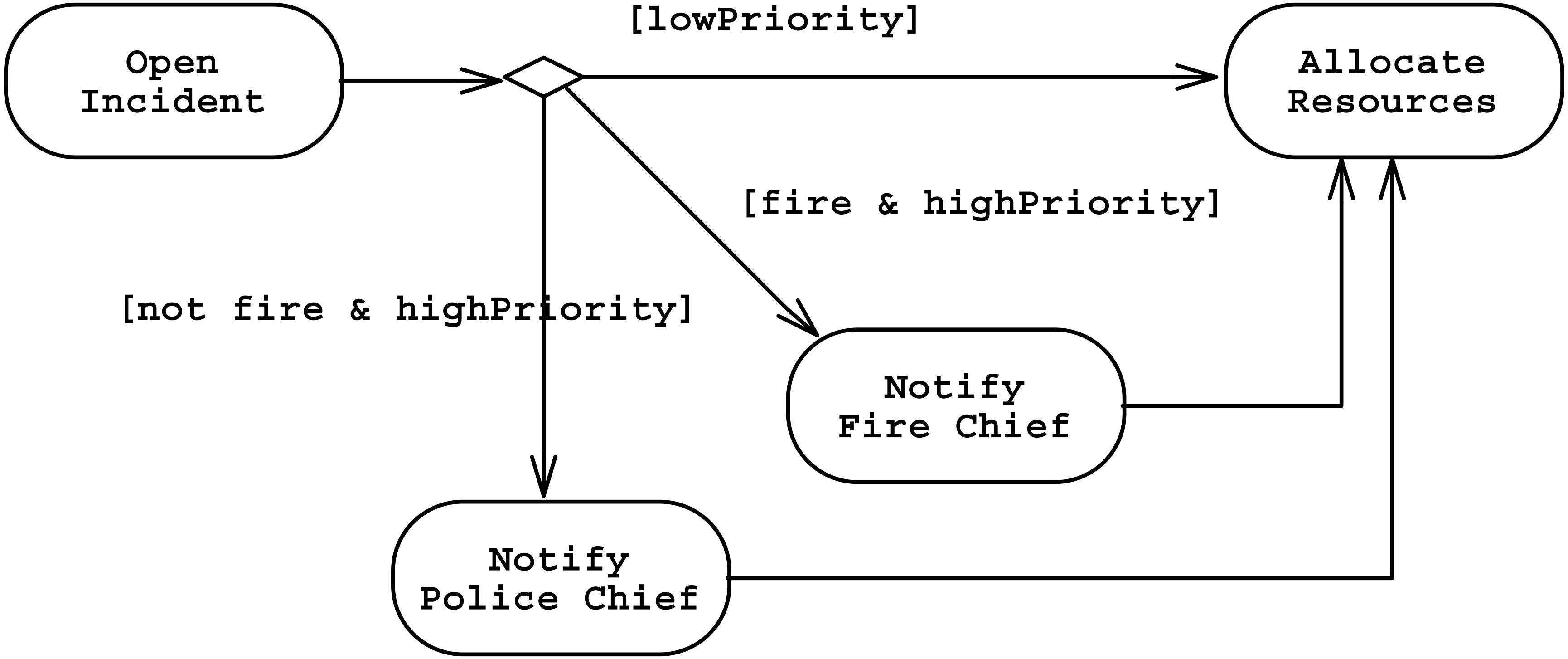
- Modelers typically use activity diagram to illustrate the following
  - The flow of complicated use case
  - A workflow across use cases
  - The logic of an algorithm

Control Nodes in an Activity Diagram

- Initial node
- Final node
  - –Activity final node
  - –Flow final node
- Fork node
- Join node
- Merge node
- Decision node



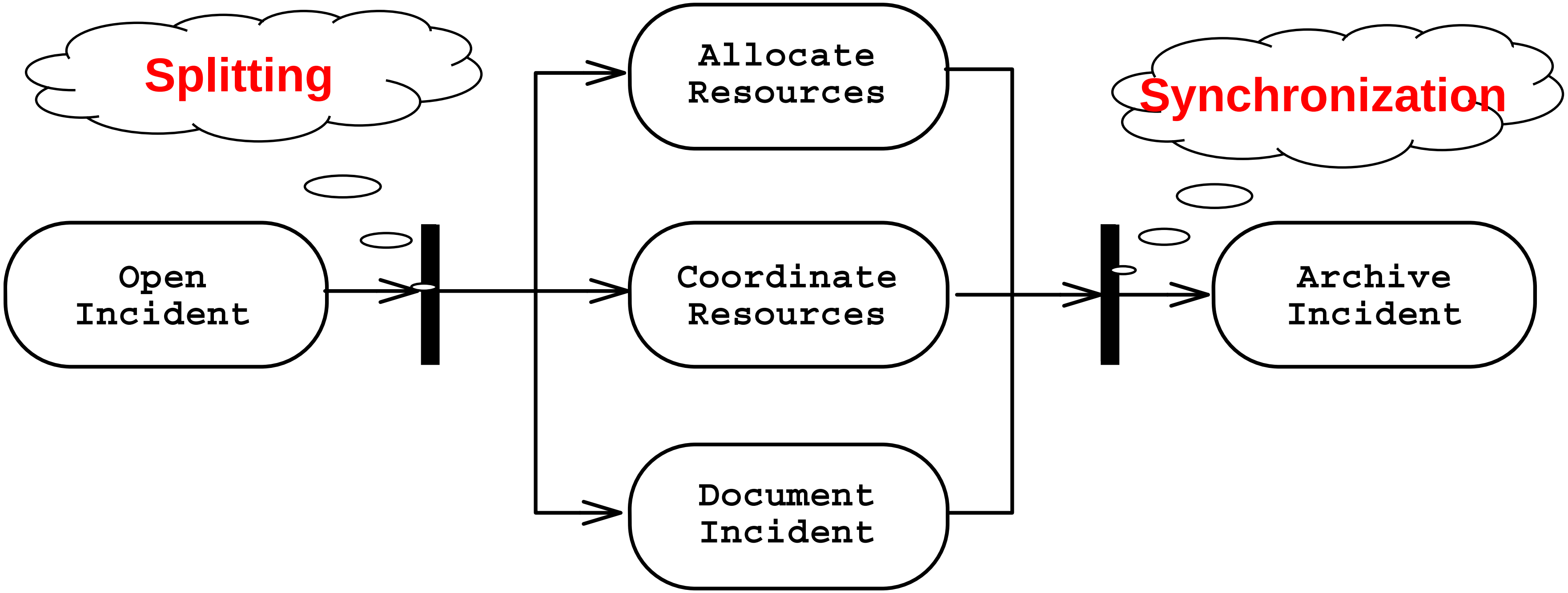
Activity Diagram: Modeling Decisions



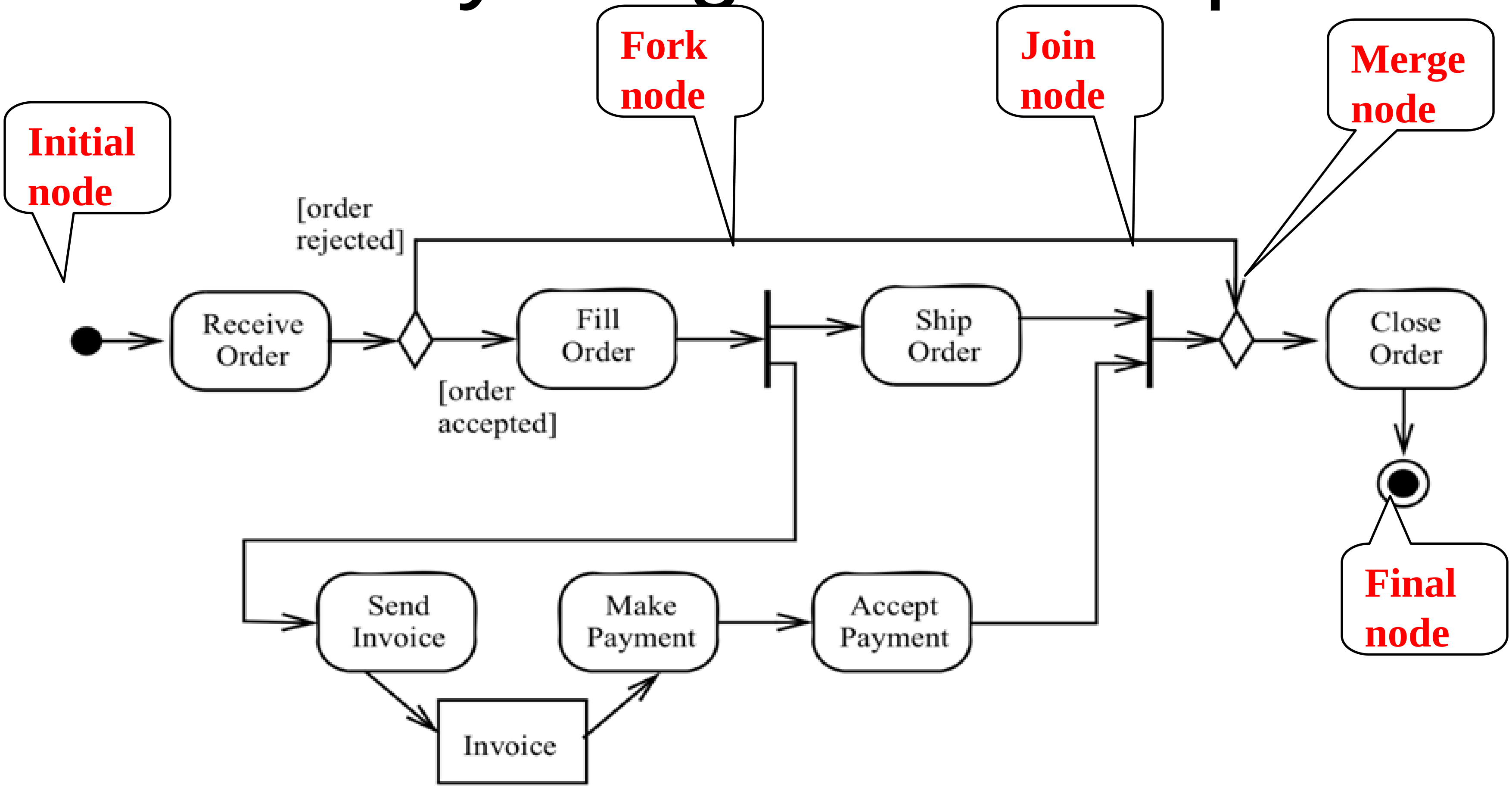
Decisions are branches in the control flow. They denote alternative transitions based on a condition o the state of an object or a set of objects. Depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows

Activity Diagrams: Modeling Concurrency

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

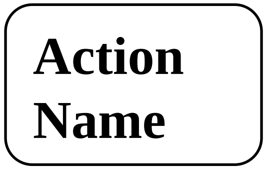


Activity Diagram Example



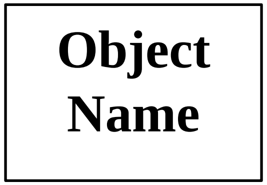
Action Nodes and Object Nodes

- Action Node



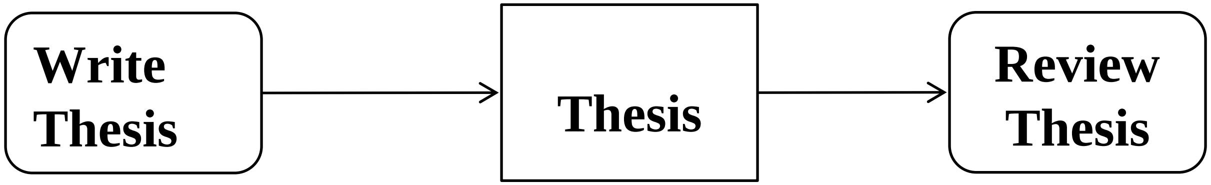
Action Name

- Object Node



Object Name

- An action is part of an activity which has local pre- and post conditions
- Historical Remark:
  - In UML 1 an action was the operation on the transition of a state machine.

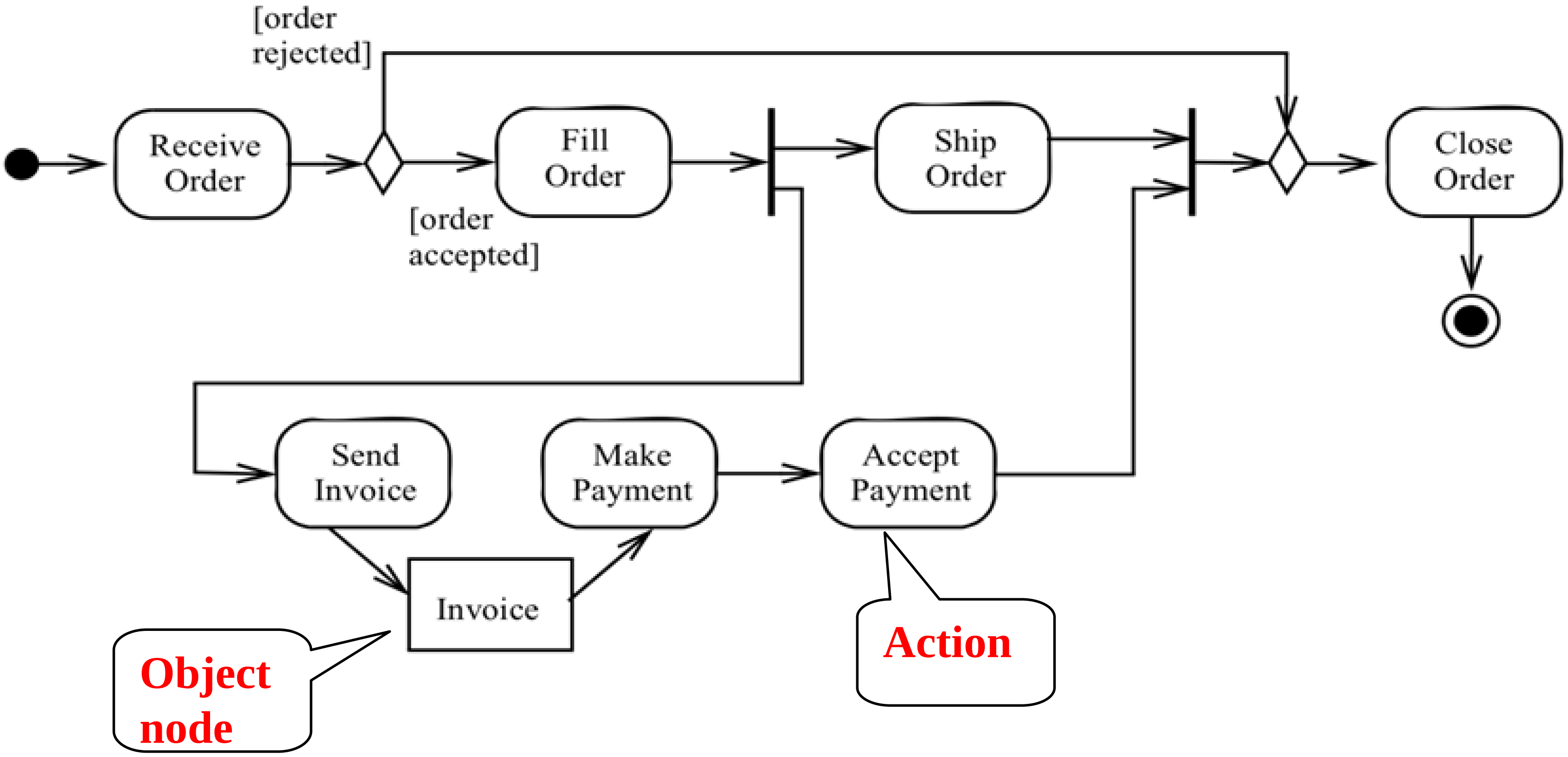


Write Thesis

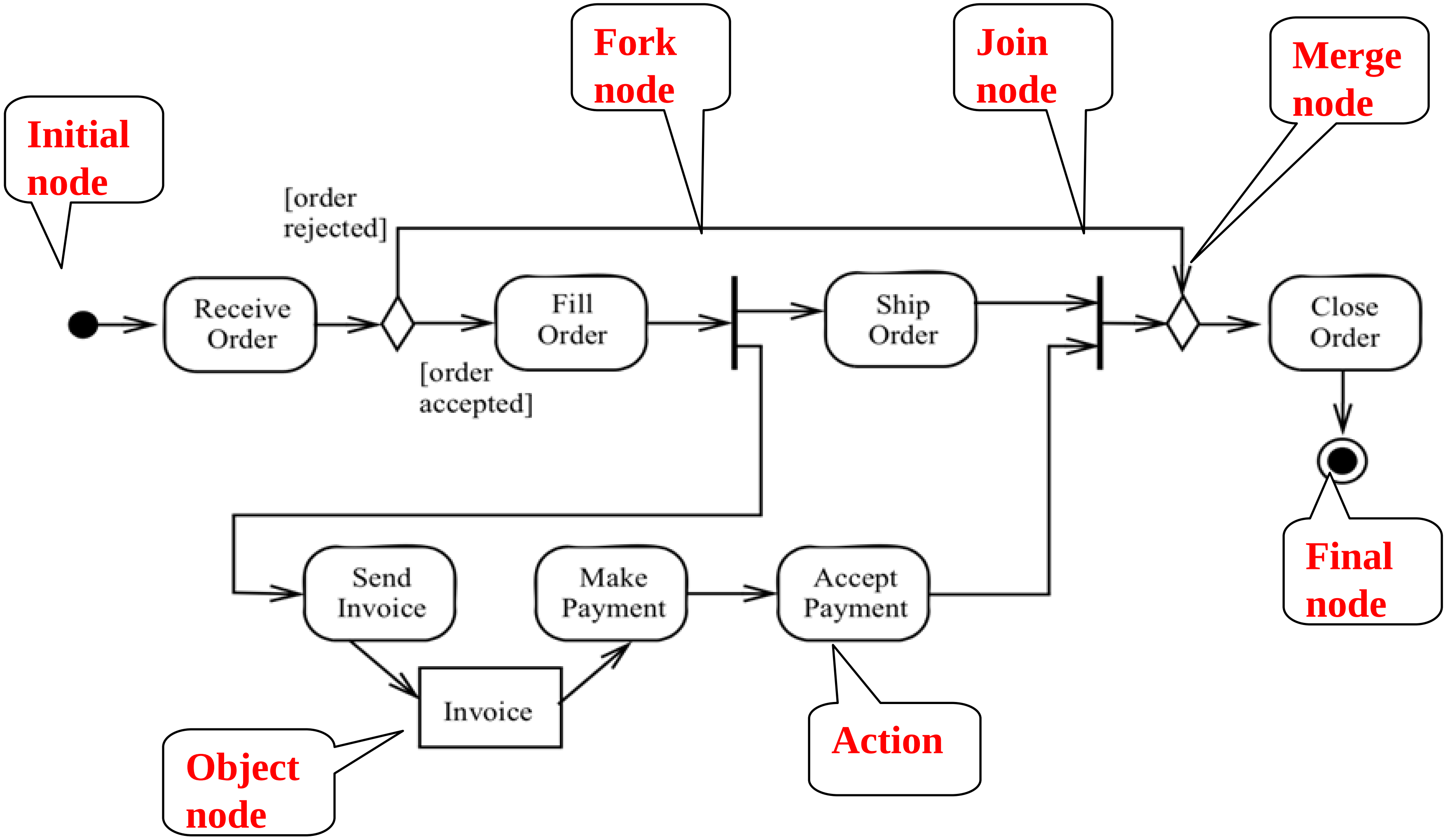
Thesis

Review Thesis

Activity Diagram Example



Summary: Activity Diagram Example



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