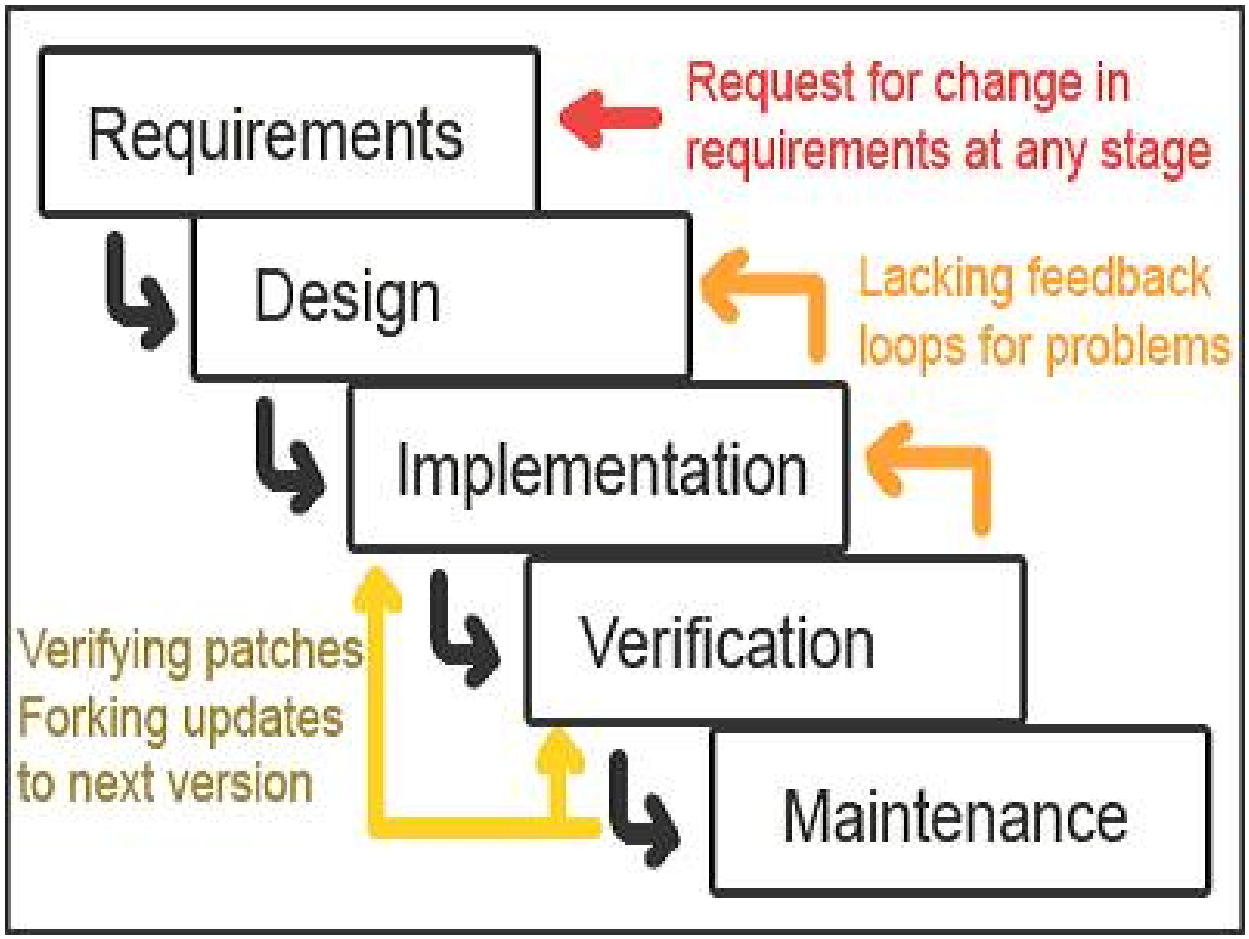
UNIFIED MODELING LANGUAGE (UML)

Yuqun Zhang

Requirements Engineering

- One element of the **Waterfall Model**
  - Requirements Engineering
  - Design
  - Implementation
  - Testing (Verification)
  - Maintenance

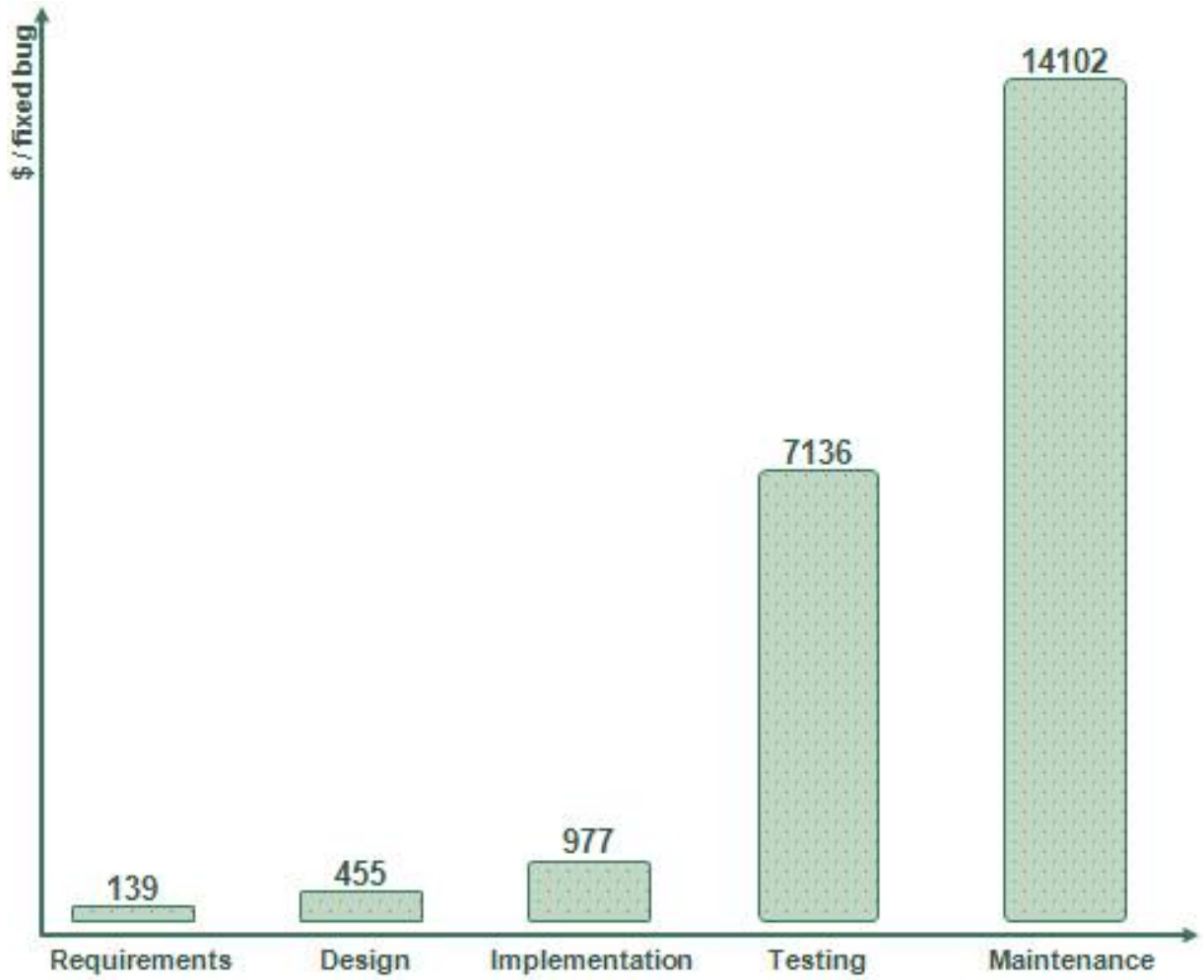
Waterfall Model



Why are Requirements Important?

- Clearly a loaded question
- Better stated: why is defining requirements formally before implementing important?
  - Much of the success or failure of a project has been determined before construction (implementation) begins
  - The foundation must be laid well and planning should be adequate
- The overall goal of requirements engineering is risk reduction
  - Discover problems and inconsistencies early before implementing
  - Not really an exact “science” though much formalism exists
    - Model checking, theorem proving, knowledge representation, etc.

Identify Problems Early



Modeling

- Describing a system at a high level of abstraction
  - A model of the system
  - Used for requirements and specification

- Many notations have existed over time
  - State machines
  - Entity-relationship diagrams
  - Dataflow diagrams

History

- 1980s
  - The rise of Object Oriented Programming
  - New class of OO modeling languages
  - By the early 1990s, there were over 50 OO modeling languages

- 1990s
  - Three leading OO notations decide to combine
    - Grady Booch (BOOCH)
    - Jim Rumbaugh (OML: Object Modeling Language)
    - Ivar Jacobsen (OOSE: Object Oriented Software Engineering)
  - Why?
    - Natural evolution towards each other
    - Effort to set an industry standard

UML

- Unified Modeling Language ("Union of all Modeling Languages")
  - Enormous language
  - Many loosely related styles under one roof
- But…
- Provides a **common**, **simple**, **graphical** representation of software design and implementation
- Allows **developers**, **architects**, and **users** to discuss the workings of the software

- http://www.omg.org

Modeling Guidelines

- Nearly everything in UML is optional
- Models are rarely complete
- UML is “open to interpretation”
- UML is designed to be extended

Static Modeling in UML

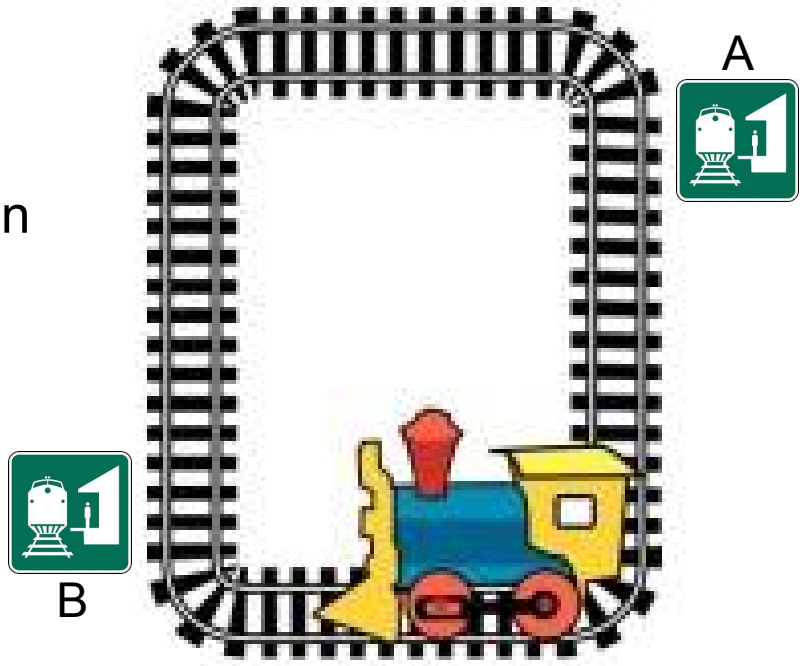
- Static modeling captures the **fixed**, **code-level** relationships in the system
  - **Class diagrams (widely used)**
  - Package diagrams
  - Component diagrams
  - Composite structure diagrams
  - Deployment diagrams

Behavioral Modeling with UML

- Behavioral diagrams are used to capture the **dynamic execution** of a system
  - **Use case diagrams (widely used)**
  - Interaction diagrams
    - **Sequence diagrams (widely used)**
  - Collaboration diagrams
  - **State diagrams (widely used)**
  - **Activity diagrams (widely used)**

Running Example: Automatic Train

- Consider an unmanned people-mover
  - E.g., as in many airports
- Train
  - Moves on a circular track
  - Visits each of two stations (A and B) in turn
  - Each station has a “request” button
    - i.e., a waiting passenger requests the train to stop at this station
  - Each train has two “request” buttons
    - i.e., a boarded passenger request the train to stop at a station



Use Case Diagrams

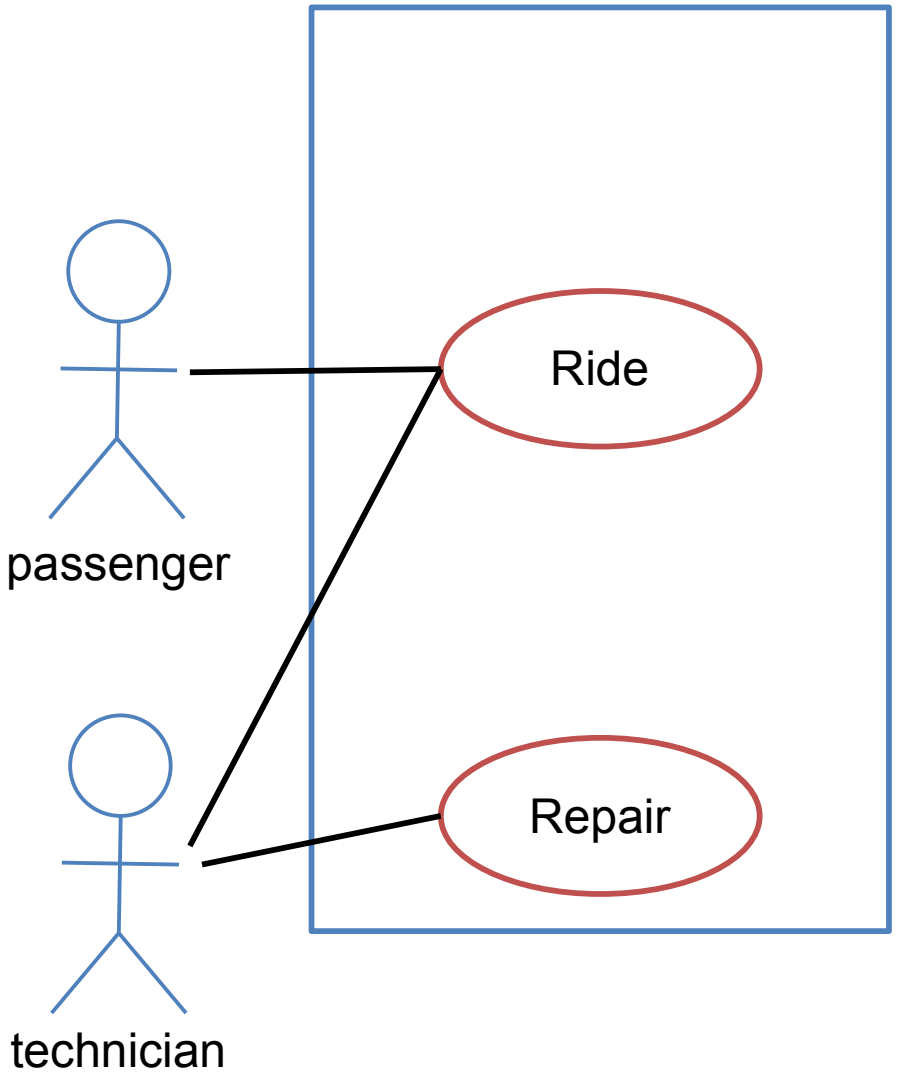
- Use case diagrams capture the requirements of a system from the user's perspective
  - The term *use case* refers to a particular piece of functionality that the system must provide (to a user)
  - Use cases are at a higher level of abstraction than other UML elements
- There will be one or more use-cases per kind of users
  - It is common for any reasonable system to have many many kinds of use cases

An Example Use Case

- Name: Normal Train Ride
- Actors: Passenger
- Entry Condition: Passenger at station
- Exit Condition: Passenger leaves station
- Event flow
  - Passenger arrives and presses request button
  - Train arrives and stops at platform
  - Doors open
  - Passenger P steps into train
  - Doors close
  - P presses request button for final stop
  - …
  - Doors open at final stop
  - P exits train

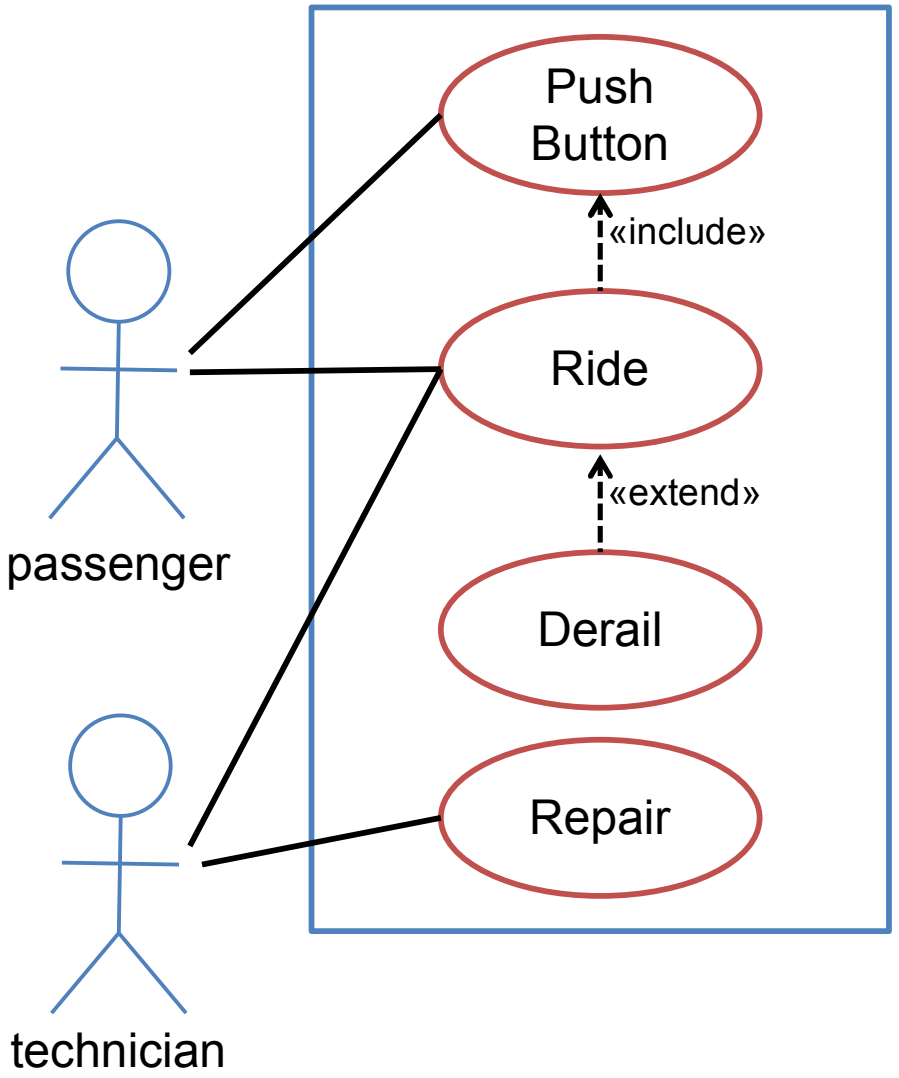
An Example Use Case Diagram

- Graph showing
  - **Actors** – stick figures
    - A **role** that a user takes when invoking a use case
    - A single user may be represented by multiple actors
  - **Use cases** – ovals
  - Edges from actor to use case showing that the actor is involved in that use case
    - Denote **association**

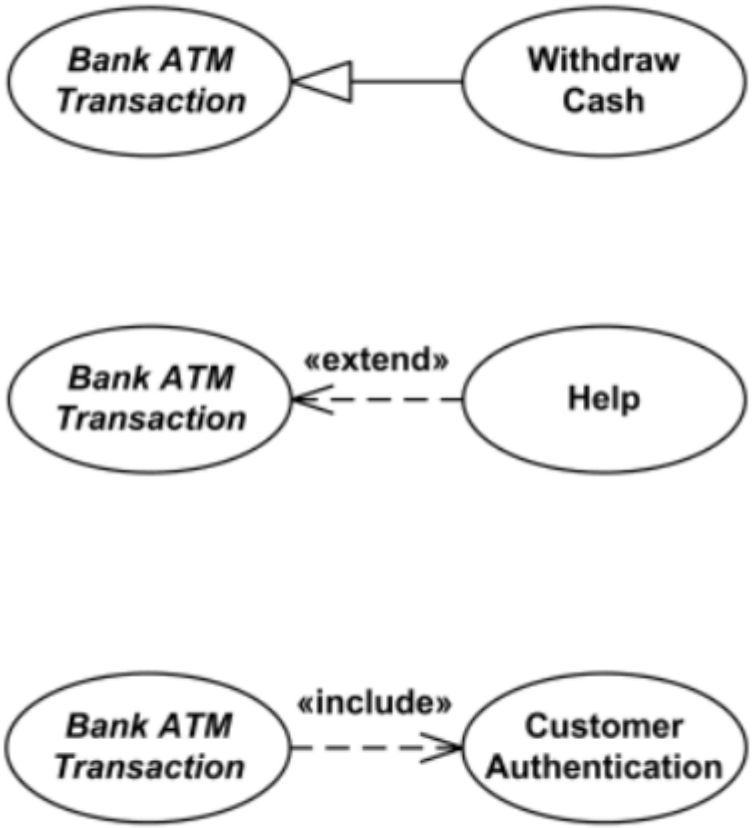


More on Use Case Diagrams

- Use cases have relationships to each other
  - Inclusion (e.g., *push button* included in *ride*)
  - Generalization/specialization (e.g., *push train button* and *push station button* are specializations of *push button*)
  - Extension expresses an exceptional variation of a use case (e.g., *derail* is an exceptional *ride*)

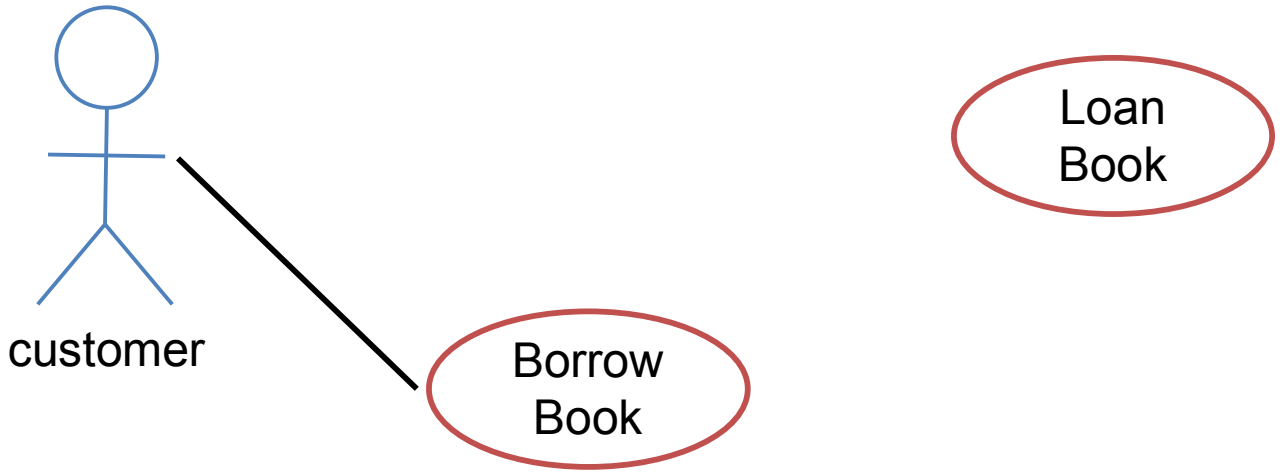


An auxiliary example



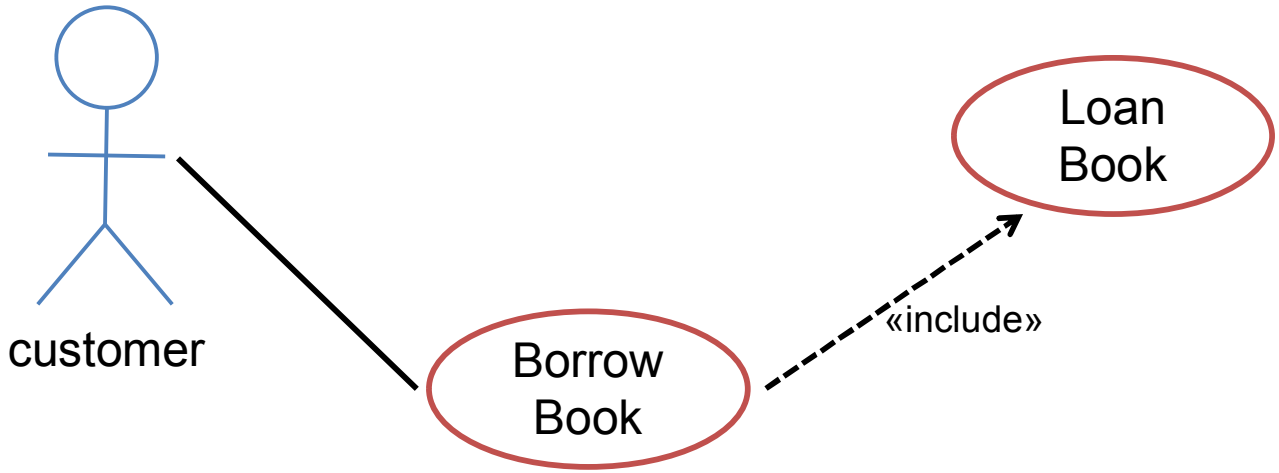
Quickly

- In English, what does this say:



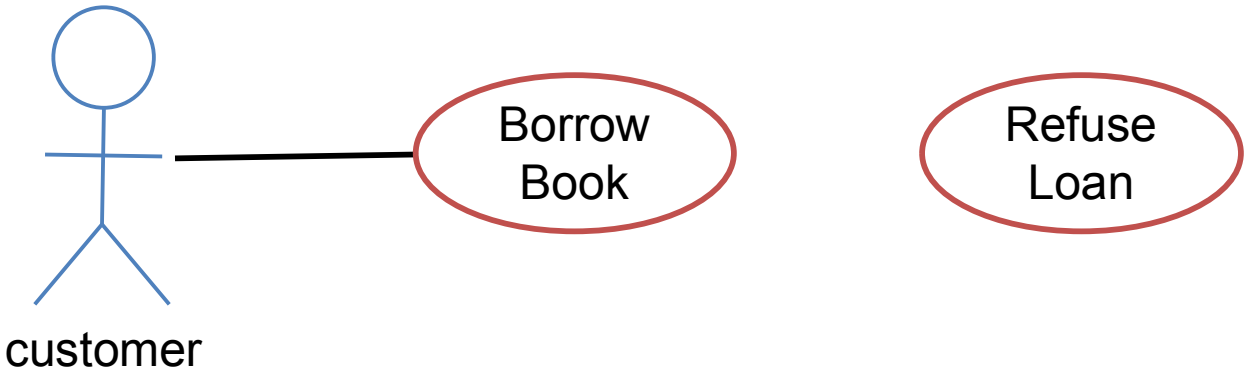
Quickly

- In English, what does this say:



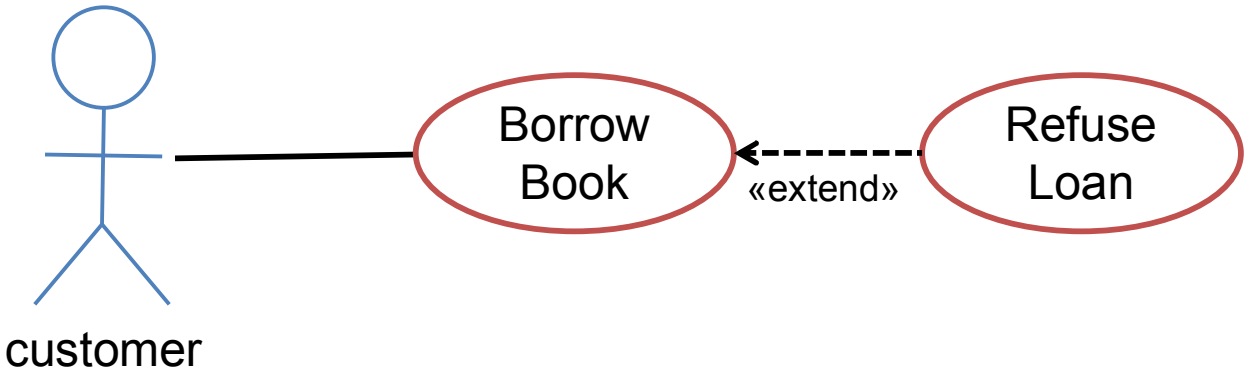
Quickly (again)

- In English, what does this say:



Quickly (again)

- In English, what does this say:



Summary of Use Cases

- Use case diagram
  - Shows all actors, use cases, relationships
  - Actors are agents that are external to the system (e.g., users)
- Supplemental information – usually in a separate document, in English
  - Entry/exit conditions (also called **pre-conditions** and **post-conditions**)
  - Nonfunctional requirements

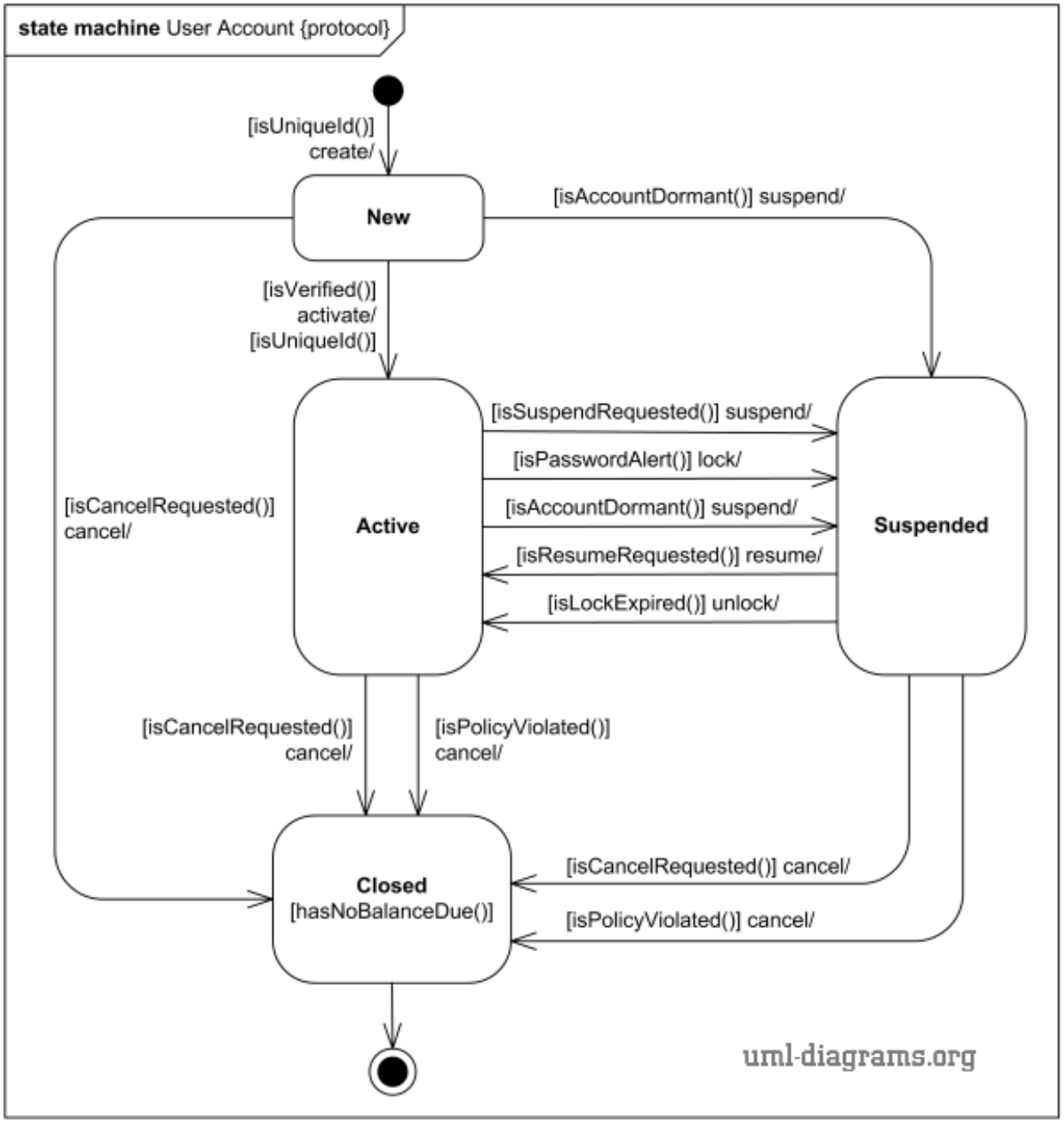
Statechart Diagrams

- Another way of specifying behavioral requirements
  - Built on state machines
- Show the various stages of an entity during its lifetime
- Can be used to show the state transitions of methods, objects, components, etc.
  - Behavioral state machines show the behavior of a particular element in a system
  - Protocol state machines show the behavior of a protocol

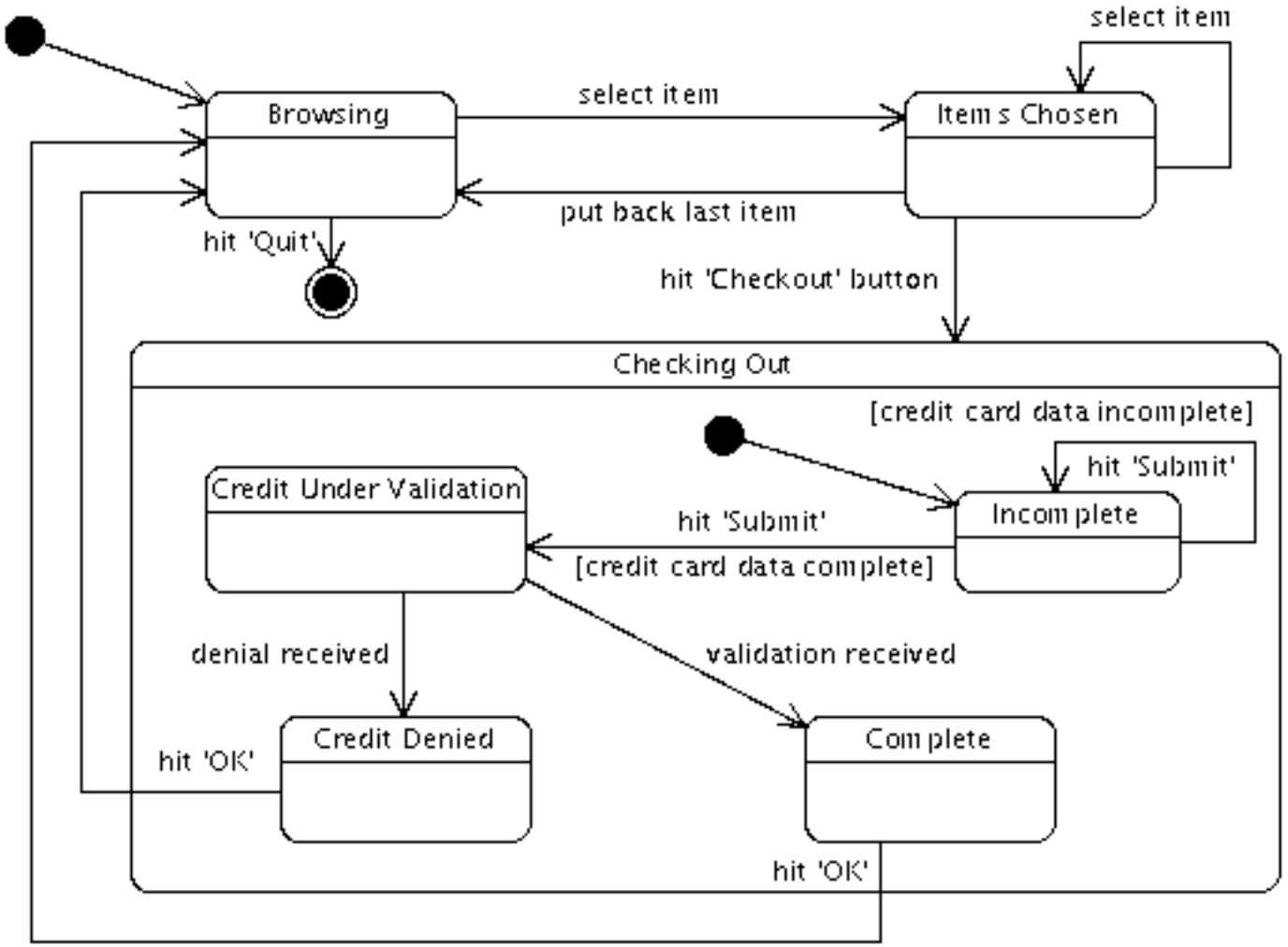
Statechart Diagram Components

- A **state** represents a condition of a modeled entity for which some action is performed, some stimulus is received, or some condition is met elsewhere in the system
- An **action** is an **atomic** execution
  - Atomic means it completes without interruption
- An **activity** is a more complex collection of behavior that may run for a long duration
- A **transition** between two states is represented as an arc from one state to another
  - Transitions can have triggers, guard conditions, and actions
  - Can be labeled with the event or action that creates the entity
    - E.g., *trigger* [*guard*] / *effect*
- The **initial state is** represented as a solid black circle

An Example


state machine User Account {protocol}
[isUniqueId()]
create/
New
[isAccountDormant()] suspend/
[isVerified()]
activate/
[isUniqueId()]
[isSuspendRequested()] suspend/
[isPasswordAlert()] lock/
[isAccountDormant()] suspend/
[isResumeRequested()] resume/
[isLockExpired()] unlock/
[isCancelRequested()]
cancel/
Active
Suspended
[isCancelRequested()]
cancel/
[isPolicyViolated()]
cancel/
[isCancelRequested()] cancel/
[isPolicyViolated()] cancel/
Closed
[hasNoBalanceDue()]
uml-diagrams.org

Another Example

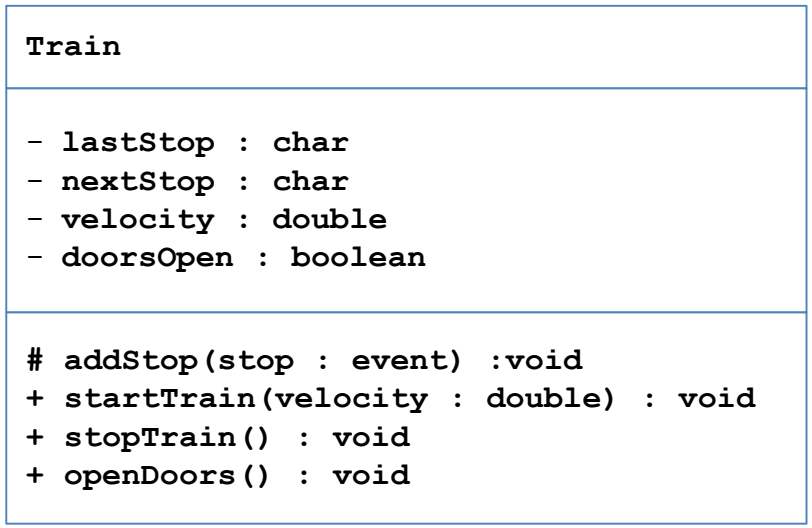


UML Class Diagram

- Models the **static relationships** between the components of a system
  - Describes the **classes** (in the OO sense)
- A single UML model can have many class diagrams
- Classes represent concepts within a system
  - Typically named using **nouns**
- A single class represents one or more objects in the system at runtime
  - Just like a java class
  - The **multiplicity** of a class is specified by a number in the upper right corner of the component
    - Usually omitted and assumed to be more than 1
    - Specifying a multiplicity of 1 indicates the class should be a **singleton**

Class Diagram

- Each box is a class
  - Name of class
  - List fields (aka attributes)
    - Visibility, type, multiplicity
  - List methods
- The more details provided, the more like a design it becomes

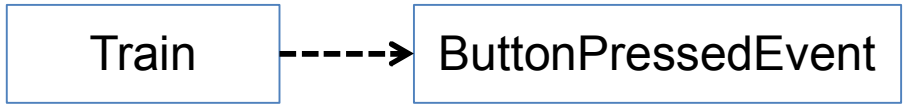


Class Relationships

- Edges show relationships between classes
  - Dependency
  - Association
  - Aggregation
  - Composition
  - Generalization
  - Realization

Dependency

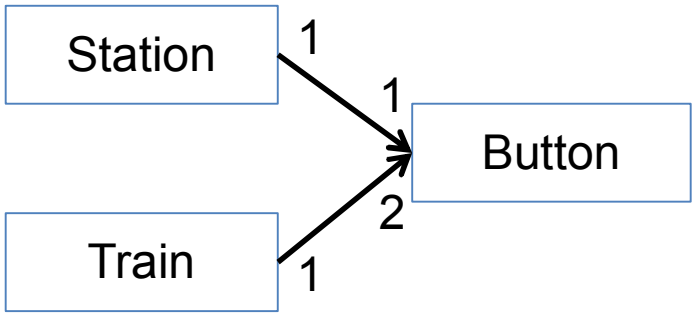
- Dependency is the weakest relationship
  - E.g., class A **uses** class B
  - Depicted by a dotted arrow



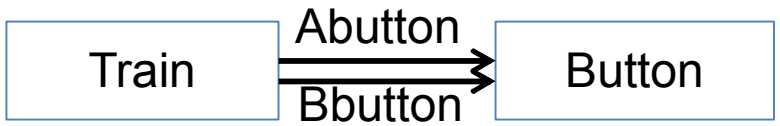

```
public class Train {
     ...
     protected void addStop(ButtonPressedEvent stop){
          // update nextStop
          ...
     }
     ...
}
```

Association

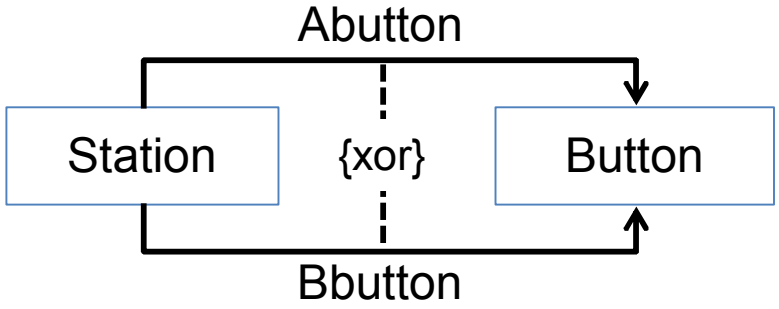
- Indicates a stronger relationship
  - E.g., class A **has a** class B
- Use number labels to indicate multiplicity
  - Use * to indicate arbitrary cardinality



- You can also explicitly name the associations

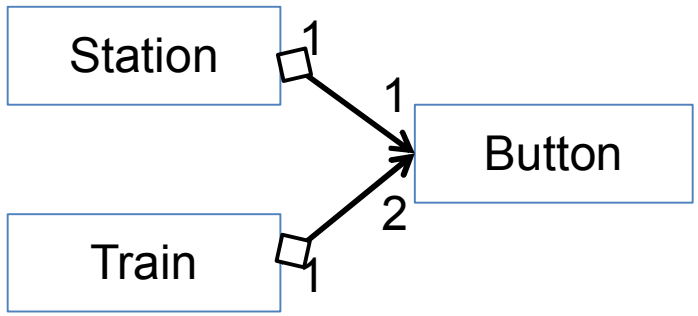


- And make them conditional



Aggregation

- Indicates a strong association
  - E.g., Class A **owns a** Class B
  - Imagine the buttons were “members” of the train/station classes (just a different design, really)
- Another way to differentiate from association:
  - Long-term association vs. (often) lifetime association (aggregation)



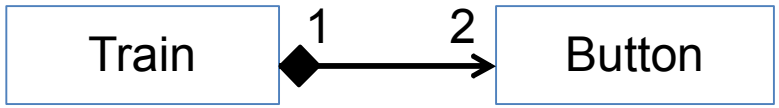
Tell me the differences

- class Person {
- private Heart heart;
- private List<Hand> hands;
- }

- class City {
- private List<Tree> trees;
- private List<Car> cars
- }

Composition

- The strongest of the association relationships
  - E.g., Class A **is made up of** Class B
- A nice way to think about the difference between aggregation and composition:
  - In Java, composition is often indicative of the inner class style relationship



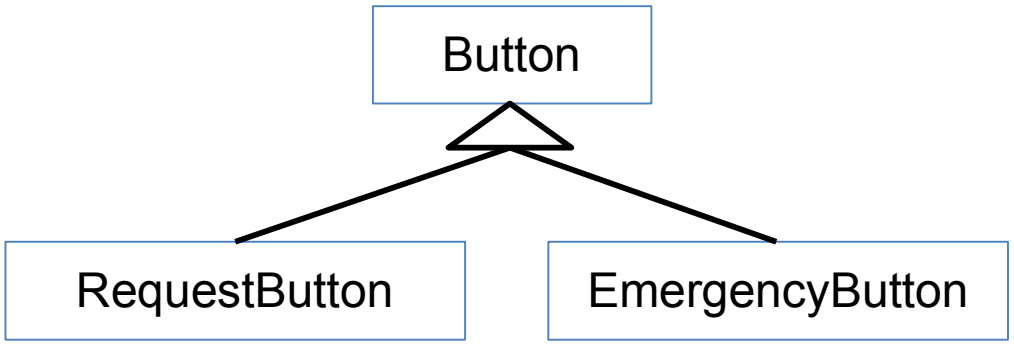
Aggregation vs. Composition

```
public class WebServer {
  private HttpListener listener;
  private RequestProcessor processor;
  public WebServer(HttpListener listener, RequestProcessor processor) {
    this.listener = listener;
    this.processor = processor;
  }
}
```

```
public class WebServer {
  private HttpListener listener;
  private RequestProcessor processor;
  public WebServer() {
    this.listener = new HttpListener(80);
    this.processor = new RequestProcessor( “/www/root” );
  }
}
```

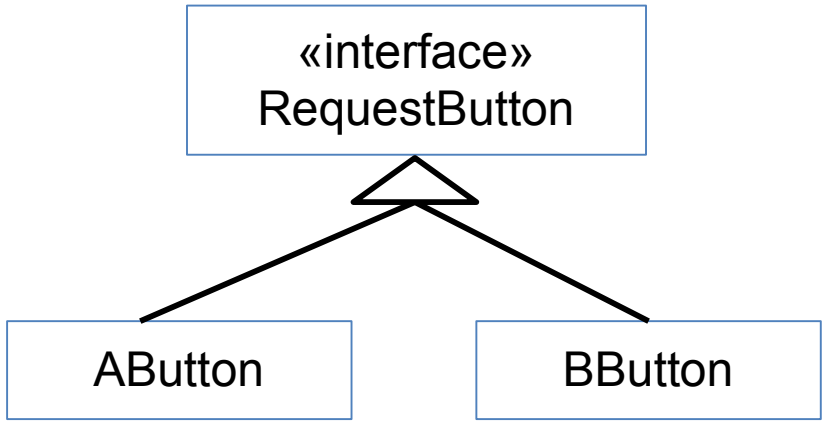
Generalization

- Generalization is used to show inheritance
- A subclass B has an **is a** relationship with superclass A
  - Or superclass A is a generalization of subclass B
  - This is the **extends** keyword in Java



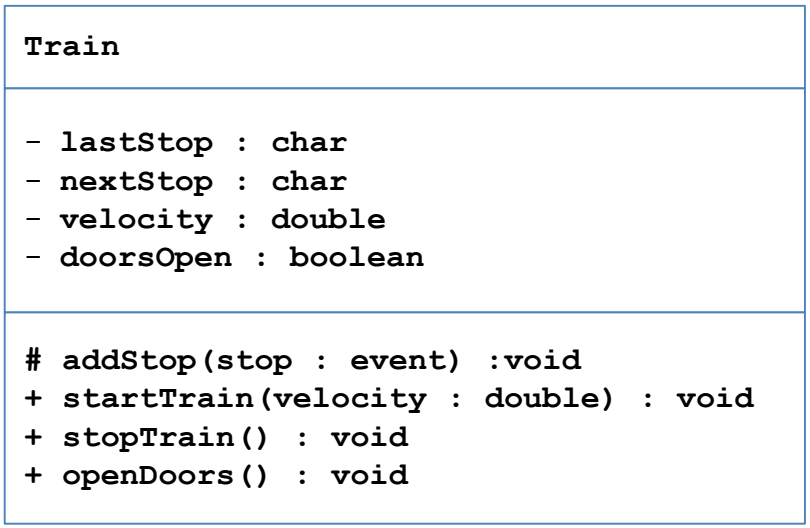
Realization

- Realization is used to show subtyping
  - E.g., Class A **implements** interface B



Operations in Class Diagrams

- Operation descriptions include
  - Visibility
    - Public +
    - Private -
    - Protected #
    - Package ~
  - Parameter list
    - Name
    - Type



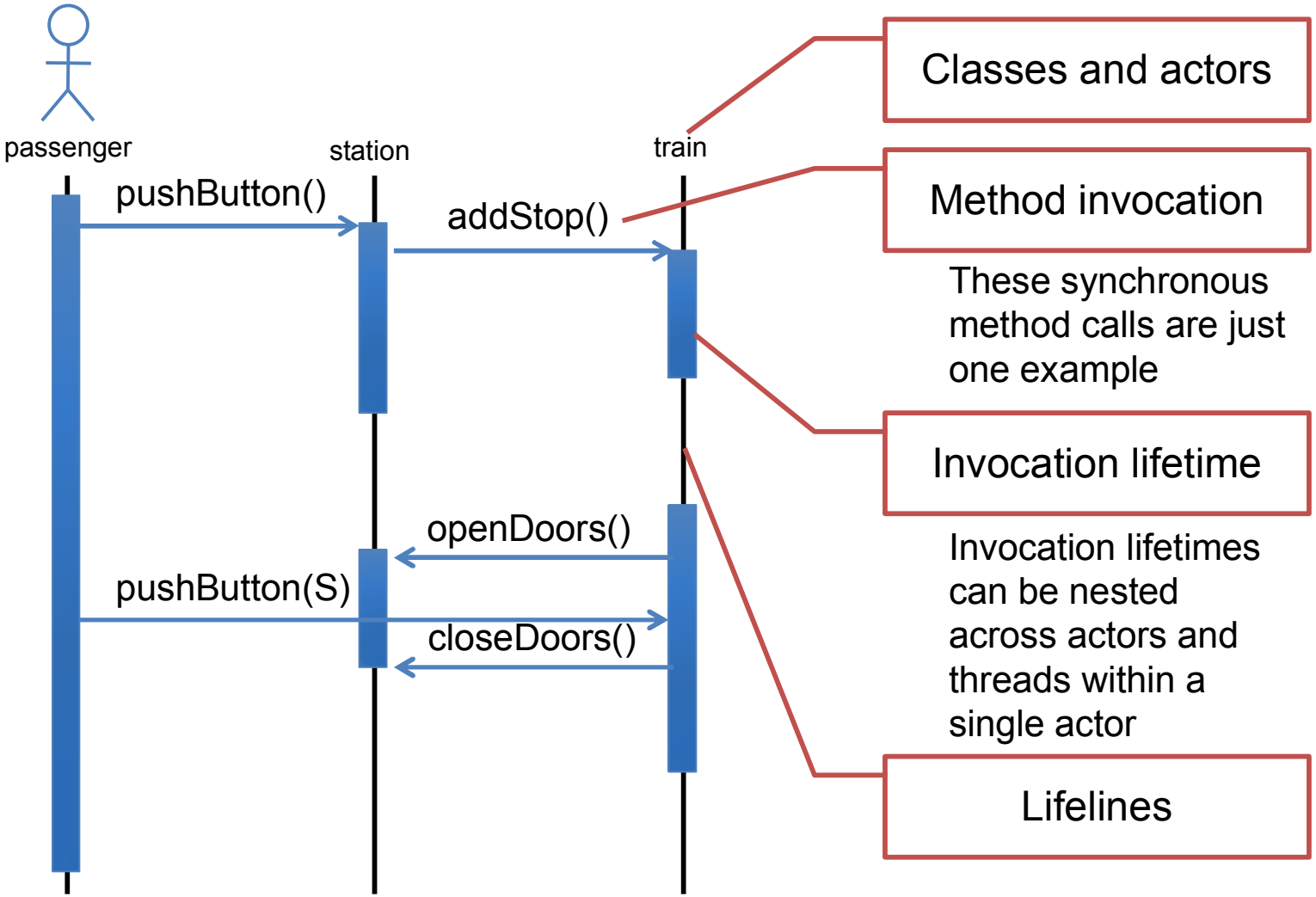
Interaction Diagrams

- Focus on **communication** between elements
  - Sequence diagrams
  - Communication diagrams
  - Interaction overview diagrams
  - Timing diagrams

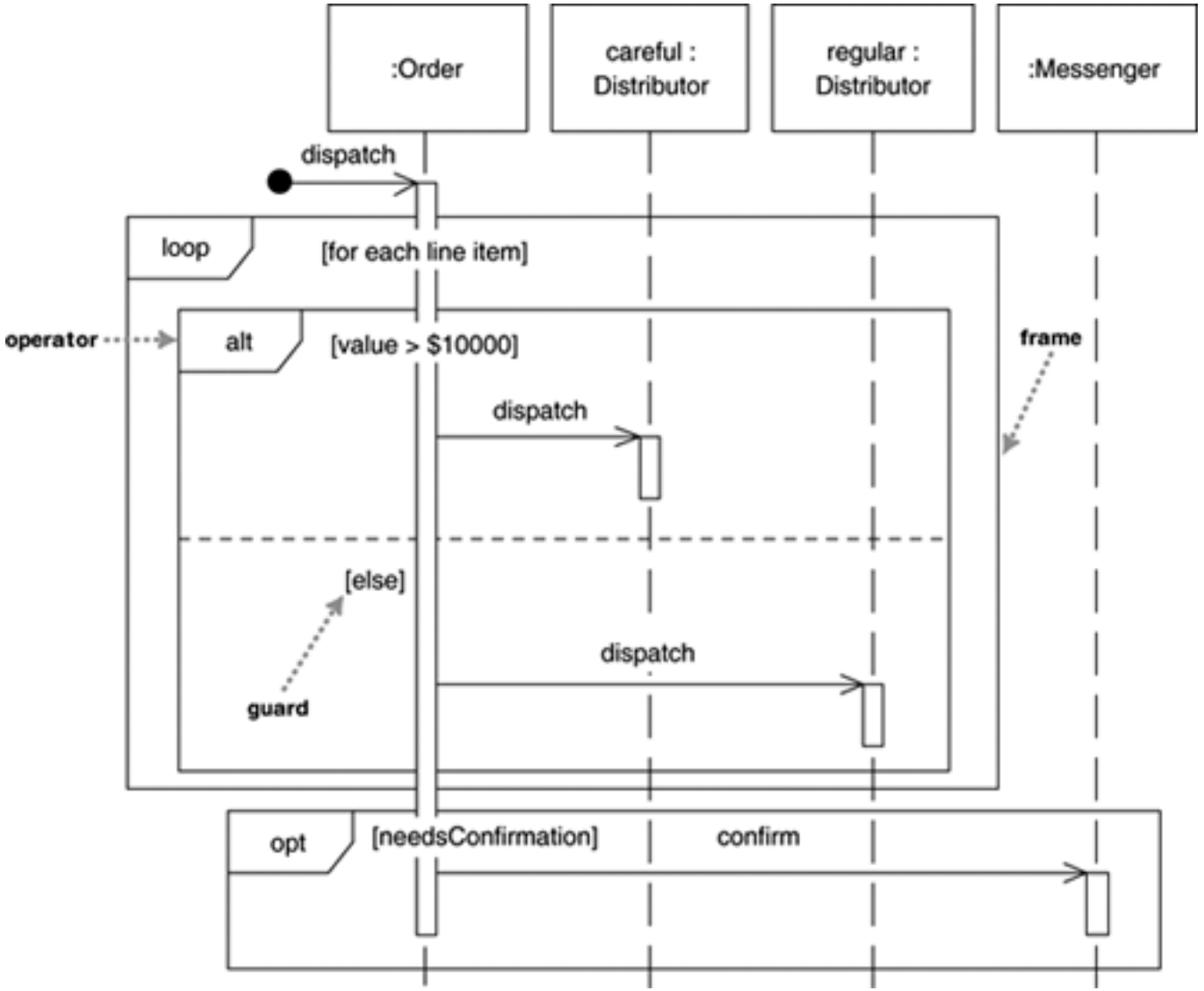
UML Sequence Diagrams

- Sequence diagrams show a time-based view of messages between objects
- Think of it as a **table**:
  - Columns are classes and/or actors
  - Rows are time steps
  - Entries show control/data flow (e.g., method invocations, important changes in state)
- Each object has a dashed **lifeline** running vertically down the diagram
  - Objects destroyed during the time covered by the sequence are not usually drawn beyond the message that killed the object

Example Sequence Diagram



Loops and Alternatives

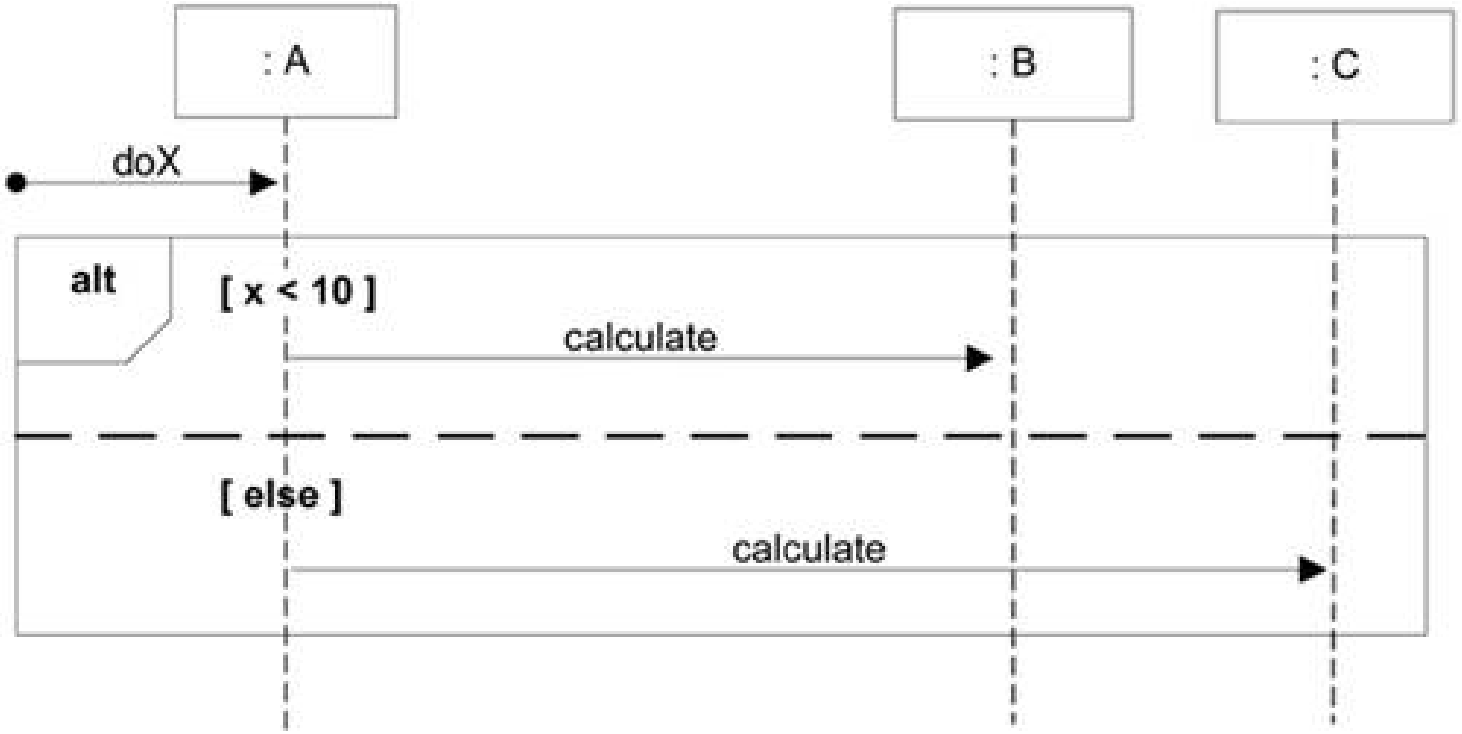


UML Sequence Diagram Frames

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards
loop	Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. There is discussion to extend to include a FOR loop (e.g., loop (i, 1, 10).
opt	Optional fragment that executes if guard is true
par	Parallel fragments that execute in parallel
region	Critical region within which only one thread can run

An Exercise

- Write pseudocode that has the same meaning as the following sequence diagram



Answer

```
public class A {
    public void doX() {
        if (x < 10) {
            B.calculate();
        } else {
            C.calculate();
        }
    }
}
```

The Other Way Around

- Draw a sequence diagram to fully represent this code:

```
public class MasterControl {
    public static void main(String[] args) {
        ...
        Input.execute(args[0]);
        CircularShift.execute();
        Alphabetizing.execute();
        Output.execute();
    }
}

public class Alphabetizing{
    public static void execute() {
        int[][] circular_shifts = CircularShift.getCircularShifts();
        int[] line_index = Input.getLineIndex();
        char[] chars = Input.getChars();
    }
}
```

A Caveat

- This is just a small subset of UML
  - There are lots of other types of diagrams which are also useful in different contexts
  - Feel free to use them; they're in your book and not too difficult to understand

UML: The Good

- A common language
  - Makes it easier to share requirements, specifications, and designs
- Visual syntax is useful (at least to a point)
  - “A picture is worth a thousand words”
  - For the non-technical, it is easier to grasp simple and intuitive diagrams even than pseudocode
- To the extent UML is precise, it forces clarity
  - Much better (in this sense) than natural language
- Commercial tool support
  - Something natural language could never have

UML: The Bad

- It's a hodge podge of ideas
  - The union of the most popular modeling languages
  - Other (sometimes useful) sublanguages remain largely unintegrated
- Visual syntax does not scale well
  - Many details are hard to depict visually
    - Often results in ad hoc text attached to diagrams
  - No visualization advantage for large diagrams
    - 1000 pictures are very hard to understand
- Semantics is not completely clear
  - Parts of UML underspecified, inconsistent
  - Plans to fix…

QUESTIONS?