CS 130 SOFTWARE ENGINEERING

UML:

UNIFIED MODELING LANGUAGE

Professor Miryung Kim

UCLA Computer Science

Based on Materials from Miryung Kim, Christine Julien and Adnan Aziz

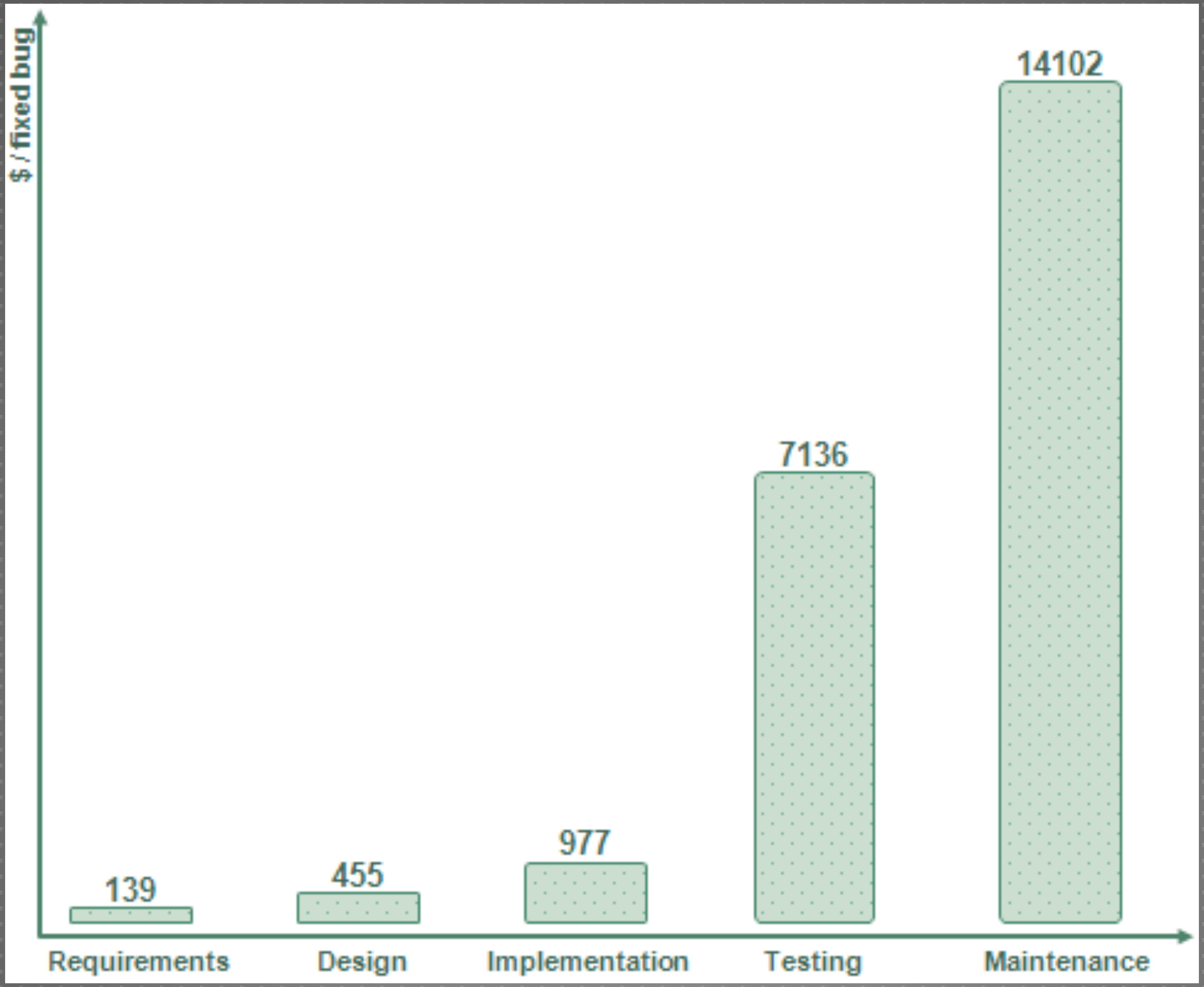
REQUIREMENTS ENGINEERING

- One element of the **Waterfall Model**
  - Requirements Engineering
  - Design
  - Implementation
  - Testing
  - Evolution
- In the real world, however, requirements engineering (and the other components of the model) are likely to be ongoing

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

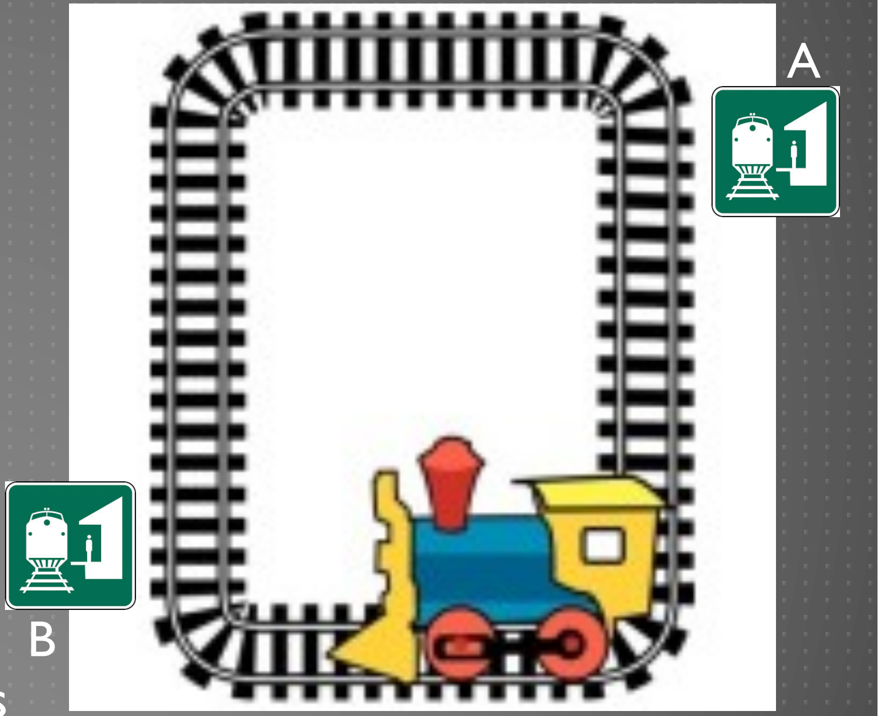
- Static modeling captures the **fixed, code-level** relationships in the system
  - Class diagrams (widely used)
  - Package diagrams
  - Component diagrams
  - Composite structure diagrams
  - Deployment diagrams

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 DIAGRAM

USE CASE DIAGRAMS

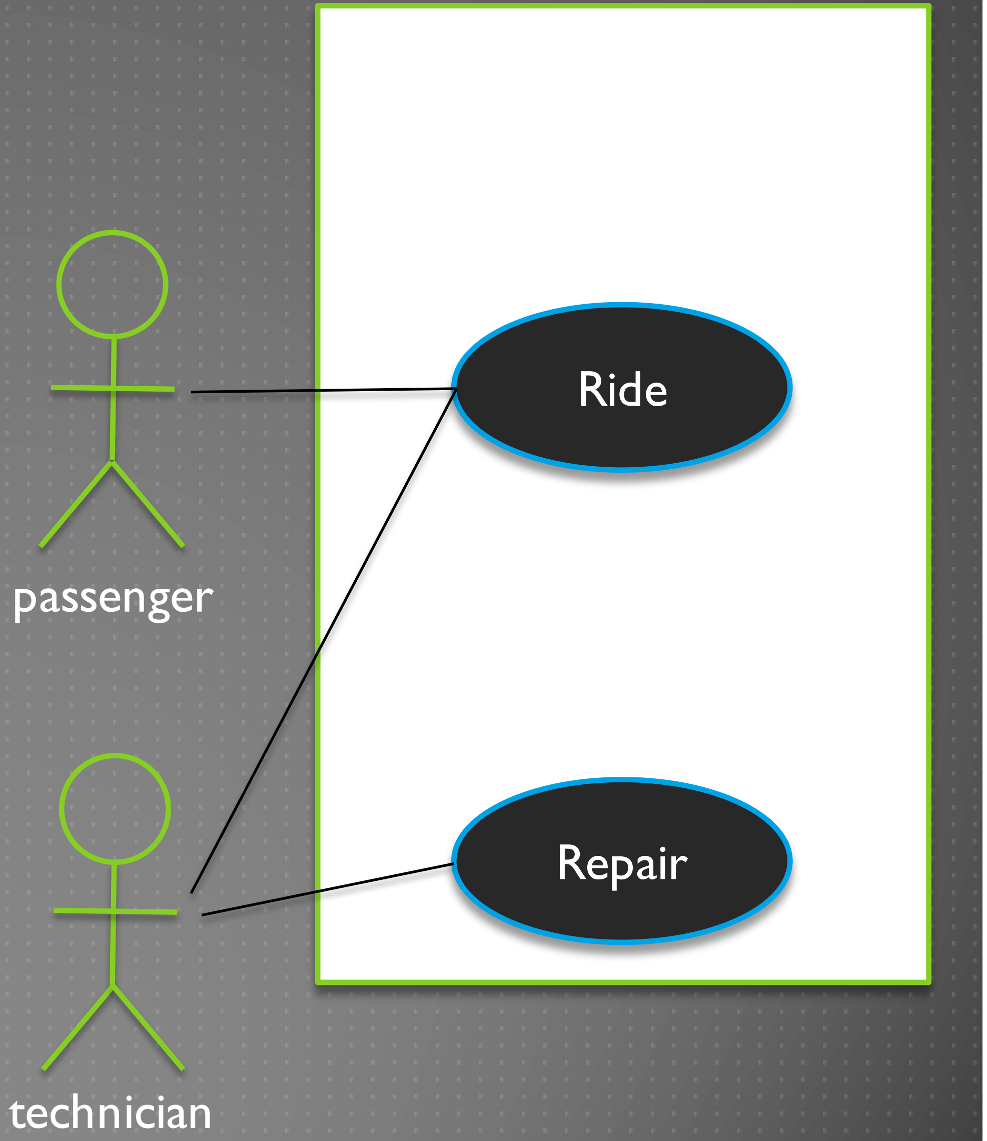
- Use case diagrams capture the requirements of a system from the user's perspective
  - The term *use case* refers to a particular piece of functionality that the system must provide (to a user)
  - Use cases are at a higher level of abstraction than other UML elements
- There will be one or more use-cases per kind of users
  - It is not uncommon 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
- Non functional requirements: ??

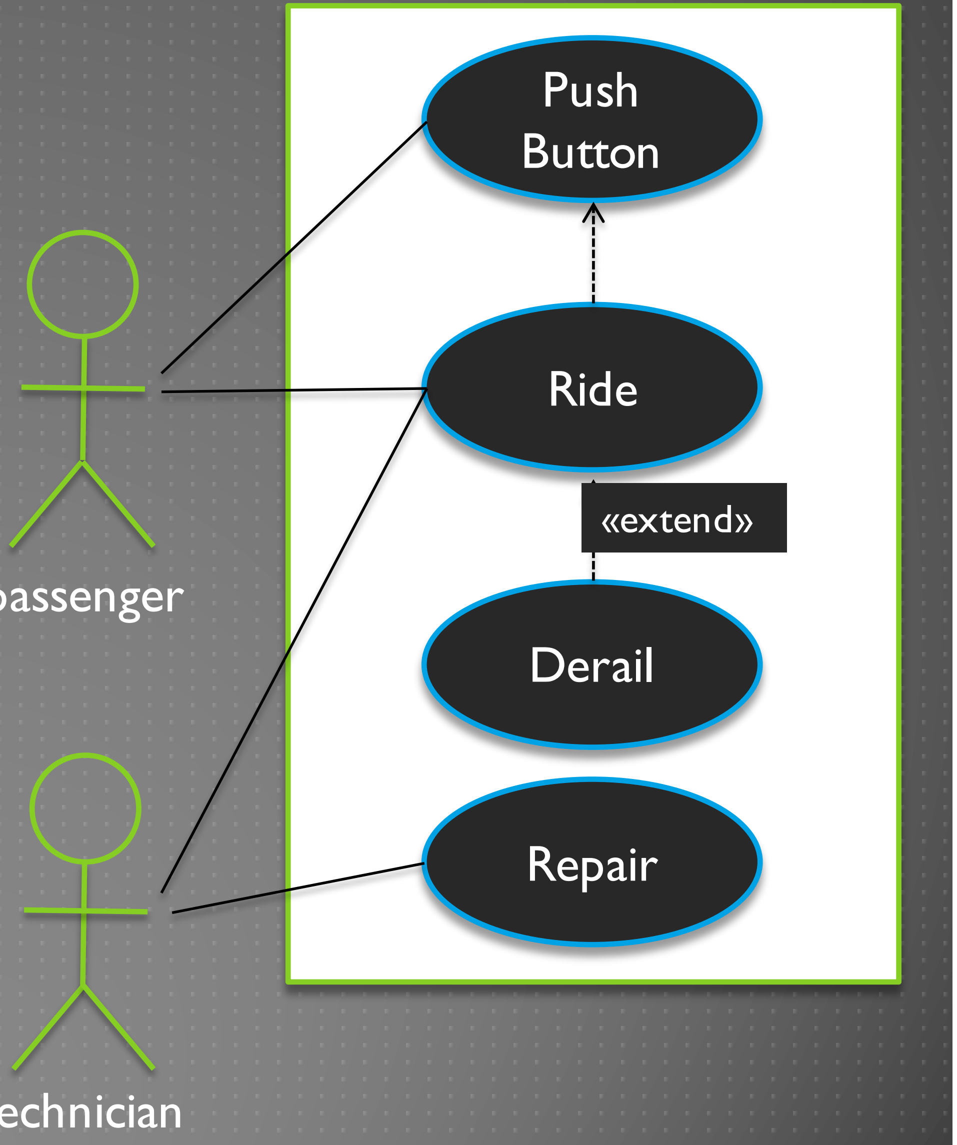
AN EXAMPLE USE CASE DIAGRAM

- 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**



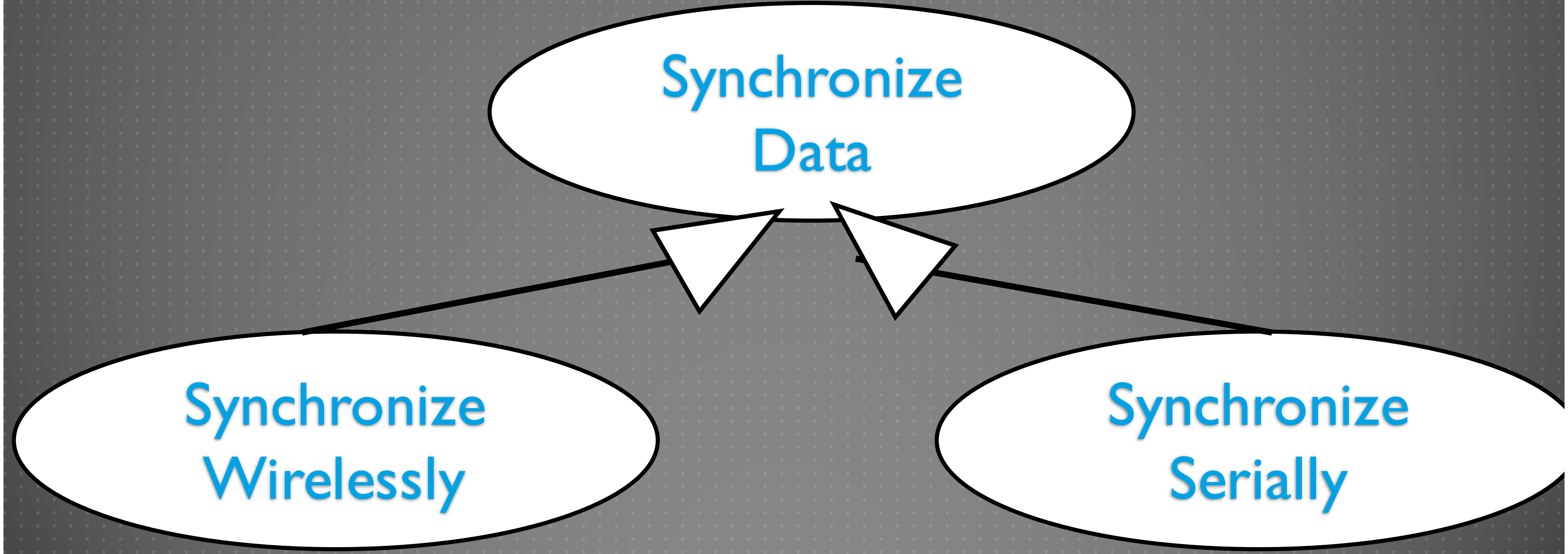
MORE ON USE CASE DIAGRAMS

- Use cases have relationships to each other
  - Inclusion (e.g., *push button* include 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*)



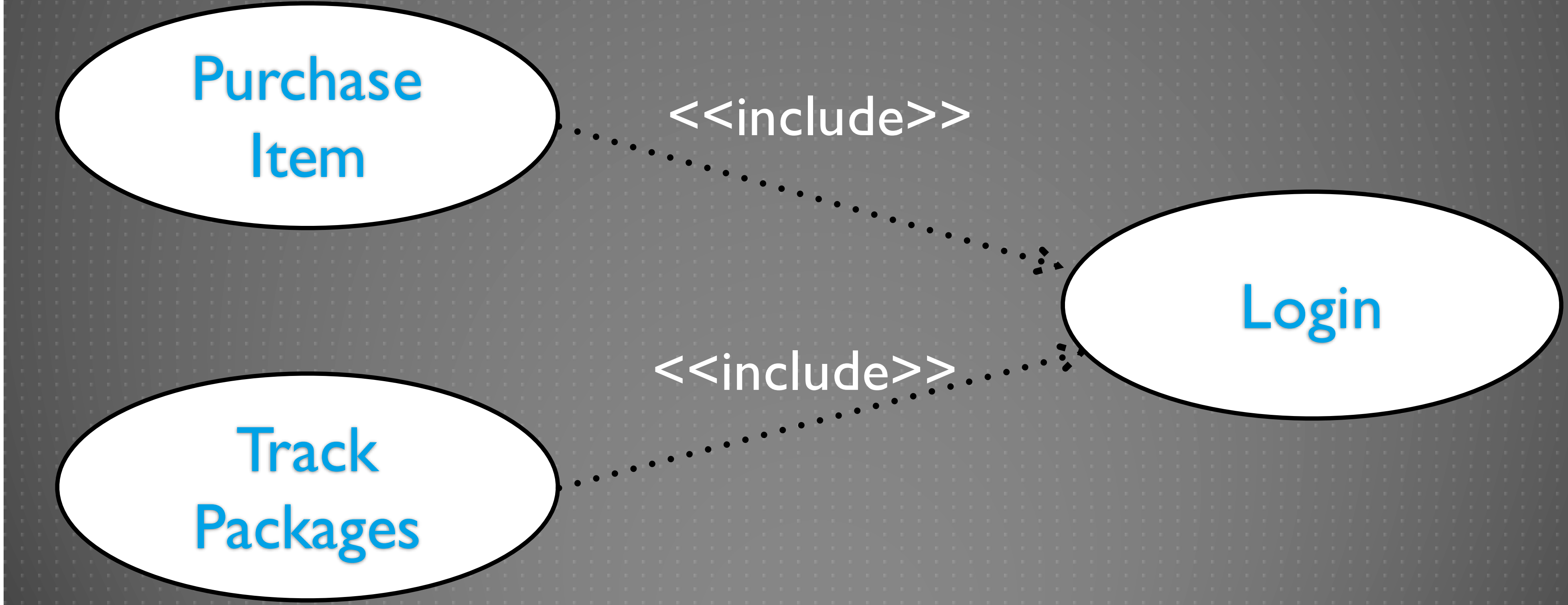
USE CASE GENERALIZATION

- Just like a class generalization, a specialized use case can replace or enhance the behavior.



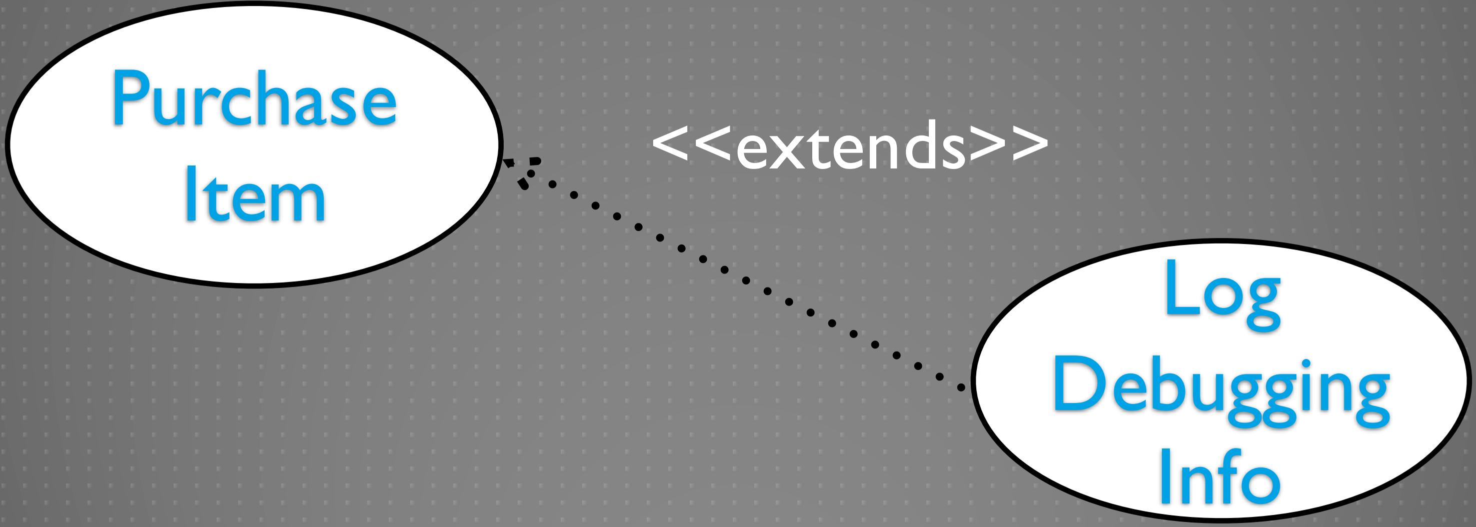
USE CASE INCLUSION

- A use case can include the behavior of another use case.



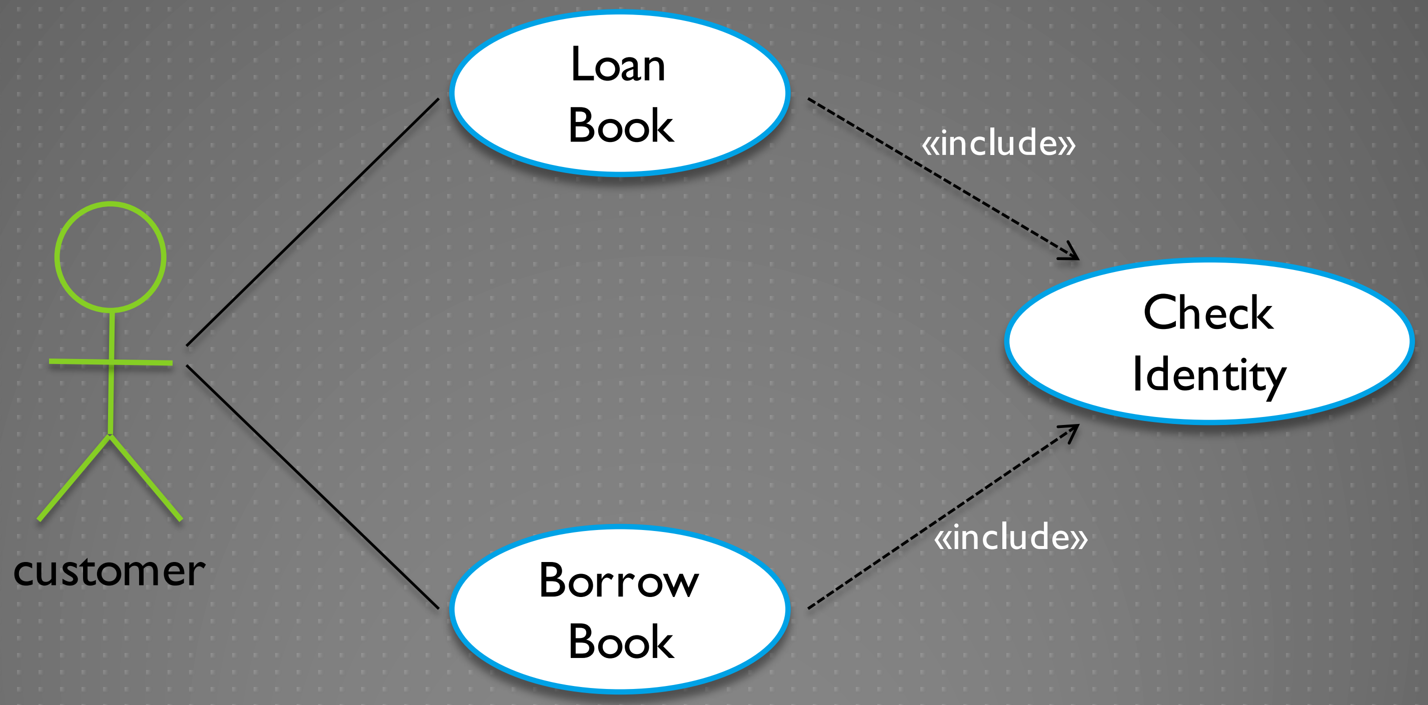
USE CASE EXTENSION

- Use case extension encapsulates a distinct flow of events that are not considered part of the normal or basic flow.



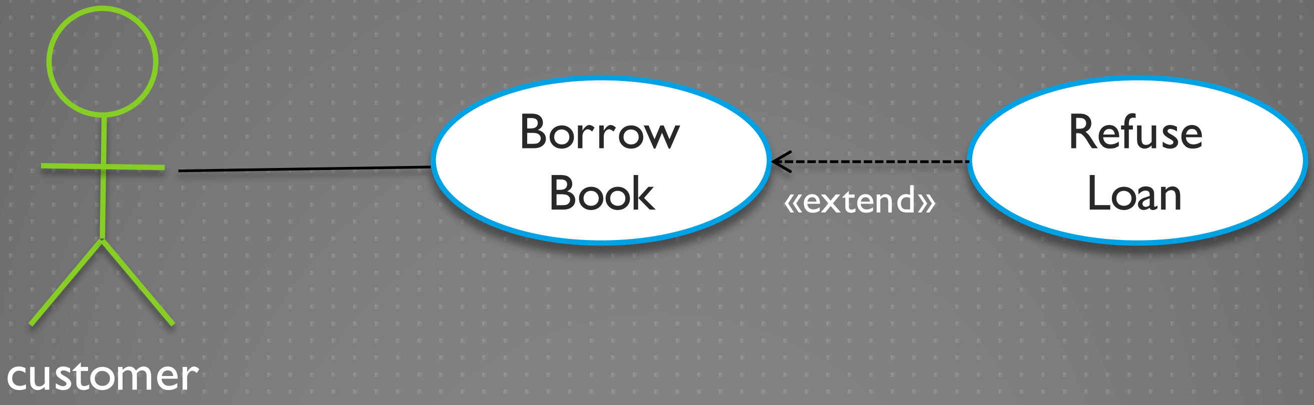
THINK-PAIR-SHARE:

- In English, what does this say:



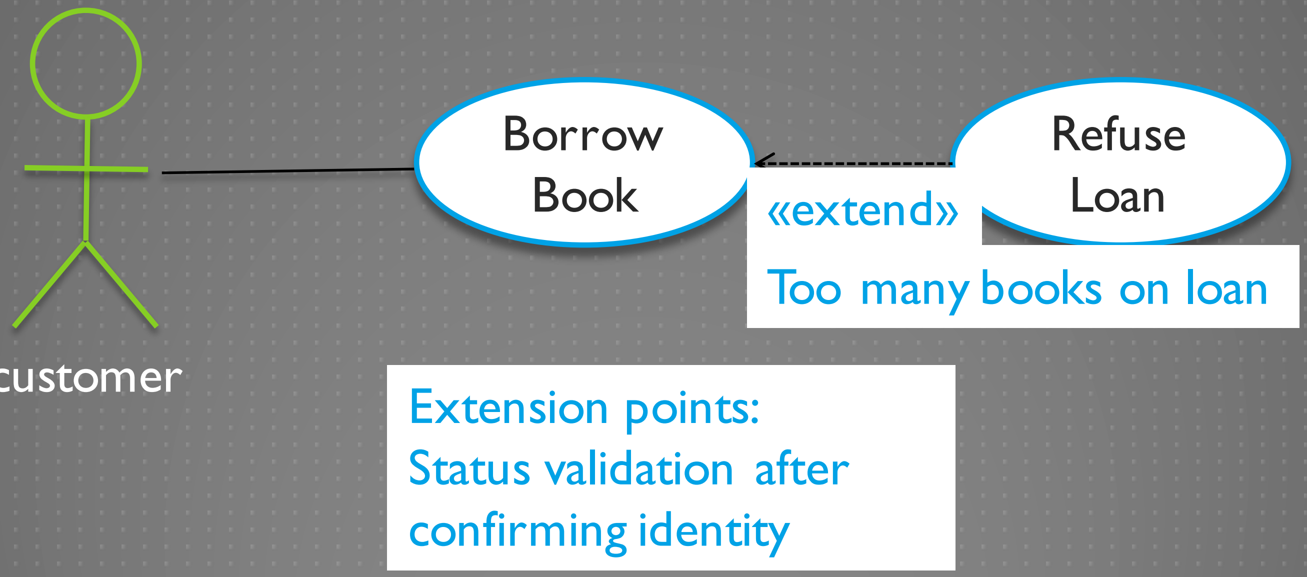
THINK-PAIR-SHARE:

- In English, what does this say:



THINK-PAIR-SHARE:

- In English, what does this say:



customer

Borrow Book

Refuse Loan

«extend»

Too many books on loan

Extension points:
Status validation after confirming identity

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**)
  - Story
  - Main and alternative **flows**
  - Nonfunctional requirements

STATE DIAGRAMS

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

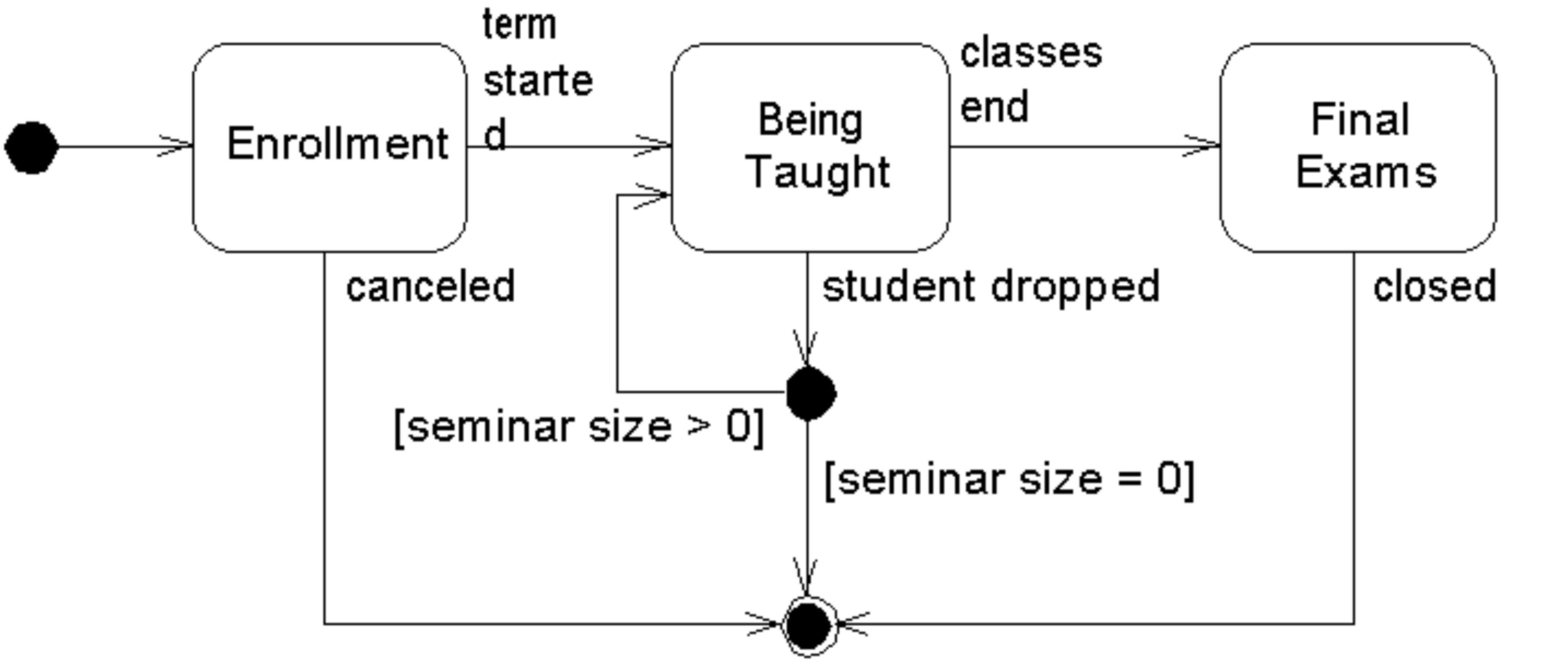
- A **state** represents a condition of a modeled entity for which some action is performed, some stimulus is received, or some condition is met elsewhere in the system
- An **action** is an **atomic** execution
  - Atomic means it completes without interruption
- An **activity** is a more complex collection of behavior that may run for a long duration

STATECHART DIAGRAM COMPONENTS

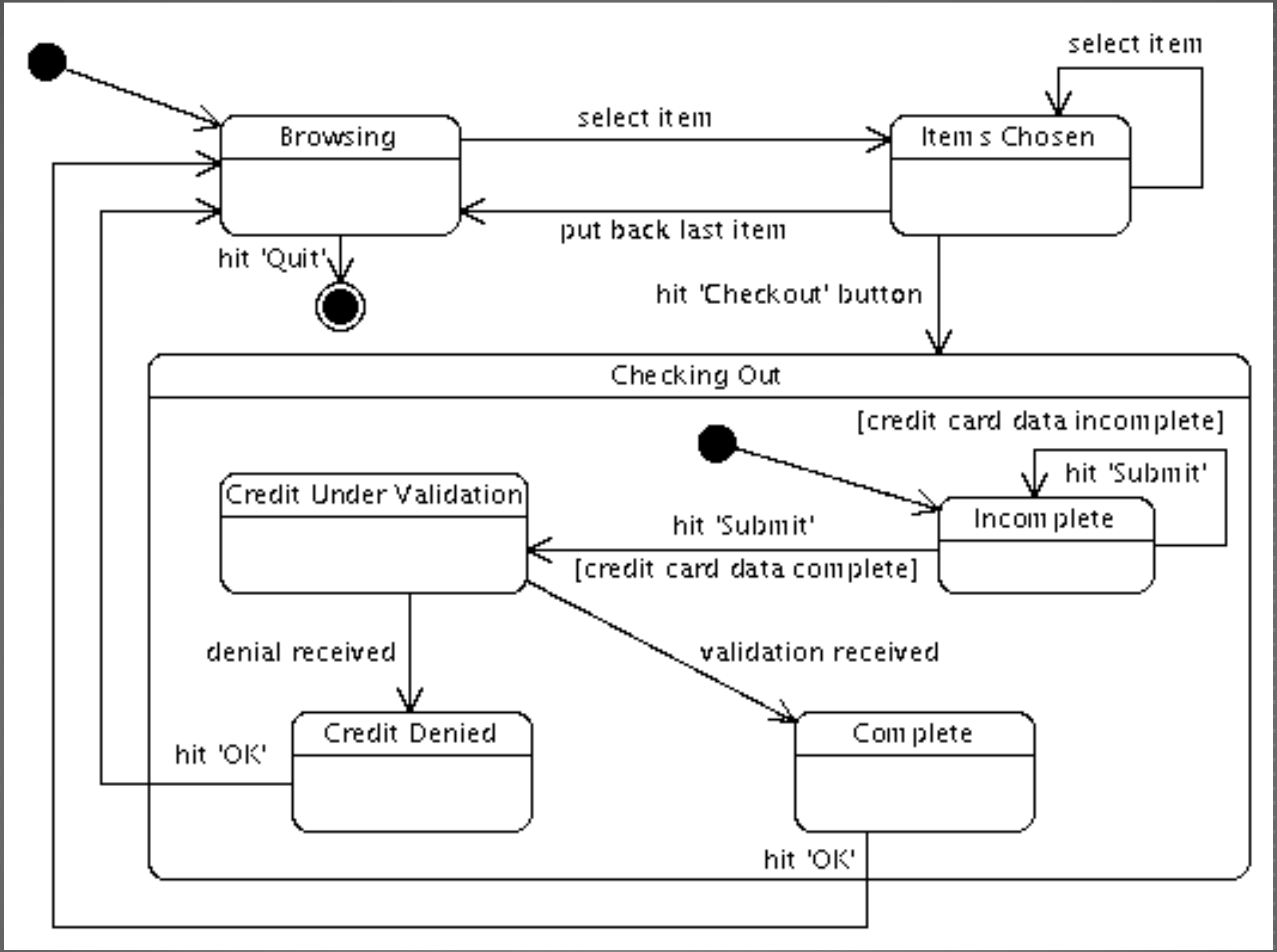
- A **transition** between two states is represented as an arc from one state to another
  - Transitions can have triggers, guard conditions, and actions
  - Can be labeled with the event or action that creates the entity
    - E.g., *trigger [guard] / effect*
- The **initial state** is represented as a solid black circle

THINK PAIR SHARE:

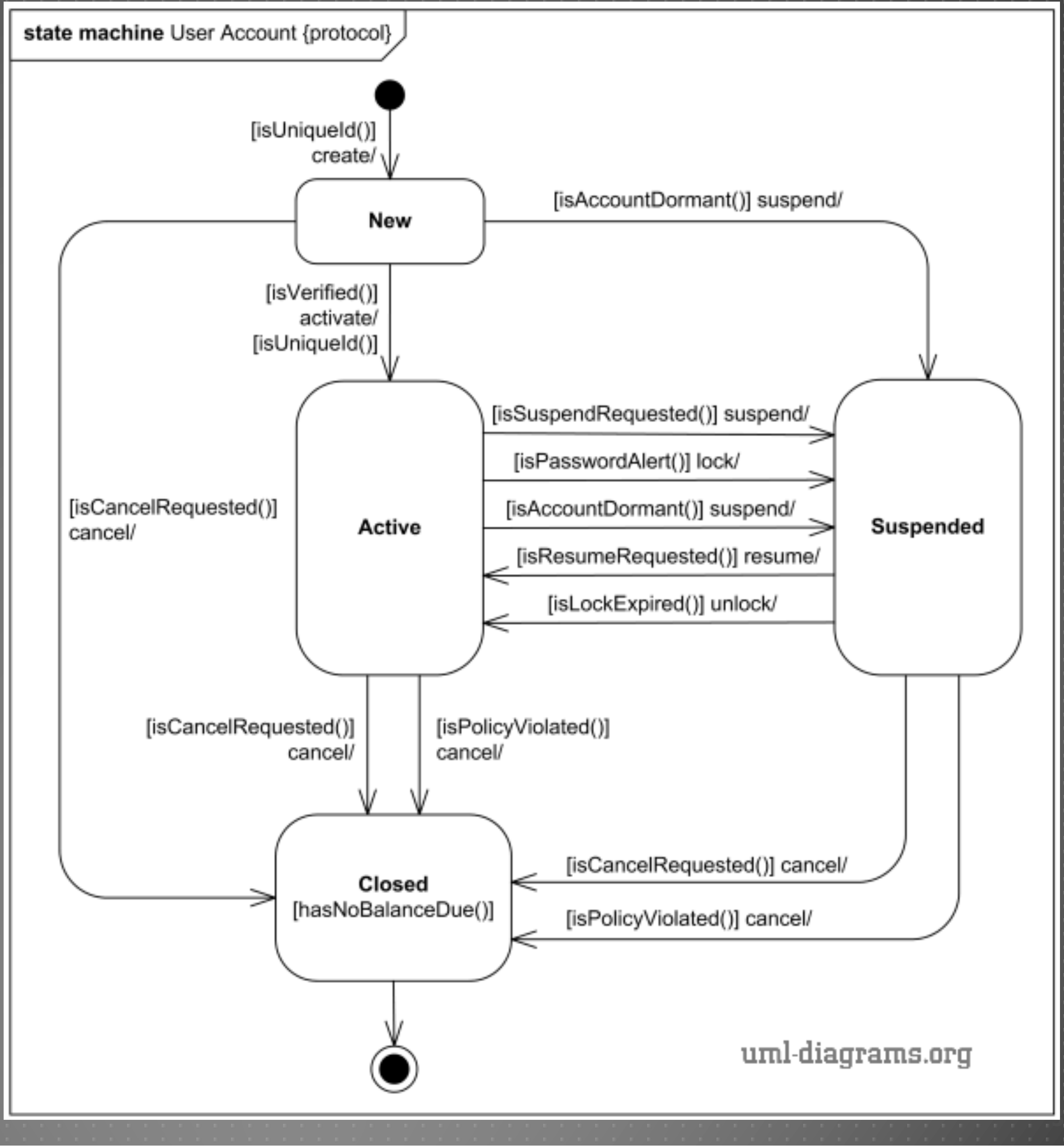
What does it mean?



ANOTHER EXAMPLE



REVIEW QUESTION


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

CLASS DIAGRAMS

UML CLASS DIAGRAM

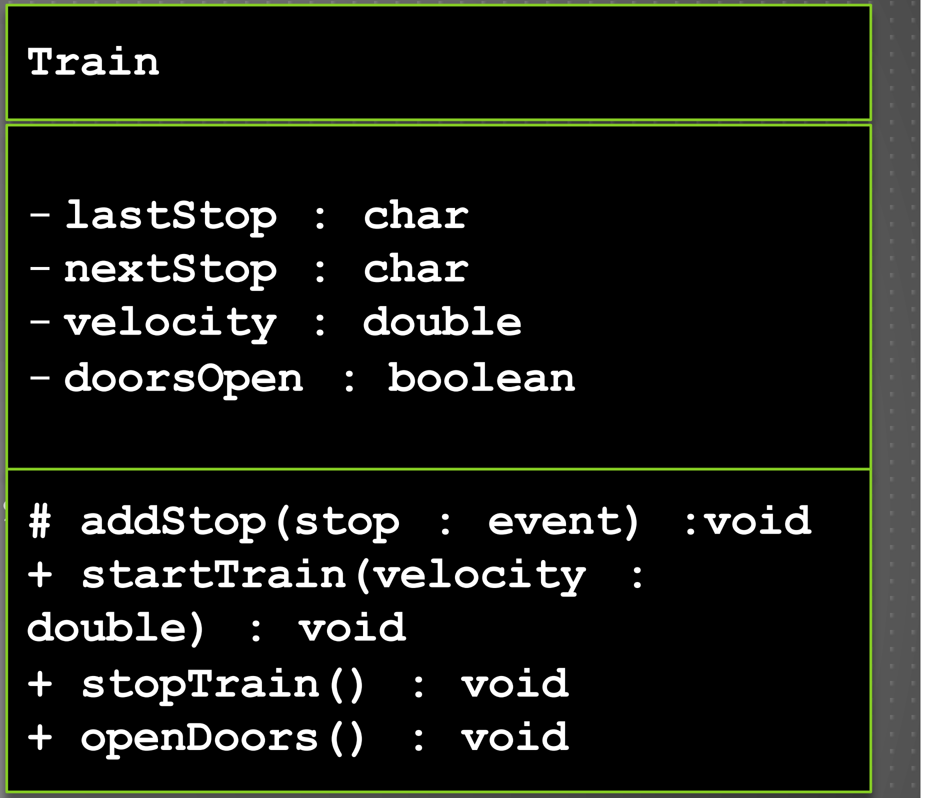
- Models the **static relationships** between the components of a system
  - Describes the **classes** (in the OO sense)
- A single UML model can have many class diagrams
- Classes represent concepts within a system
  - Typically named using **nouns**

UML CLASS DIAGRAM

- 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 detail provided, the more like a design it becomes



```
Train

- lastStop : char
- nextStop : char
- velocity : double
- doorsOpen : boolean

# addStop(stop : event) :void
+ startTrain(velocity :
double) : void
+ stopTrain() : void
+ openDoors() : void
```

CLASS PROPERTY

Ordered	Uniqueness	Collection Type
FALSE	FALSE	Bag
TRUE	TRUE	OrderedSet (e.g. TreeSet)
FALSE	TRUE	Set
TRUE	FALSE	Sequence

CLASS RELATIONSHIPS

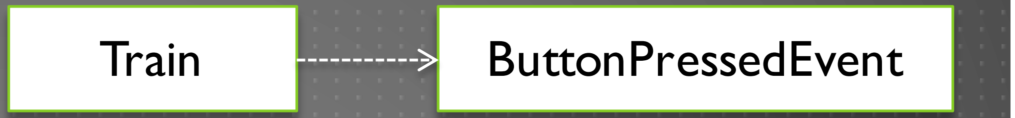
- Attributes can also be represented a class relationship notation.
- A line is drawn between the owning class and the target attribute's class.
- A quick visual indication of which classes are related.

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



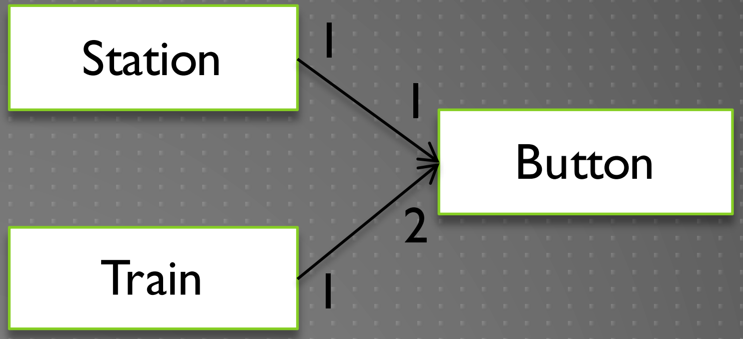
Train

ButtonPressedEvent

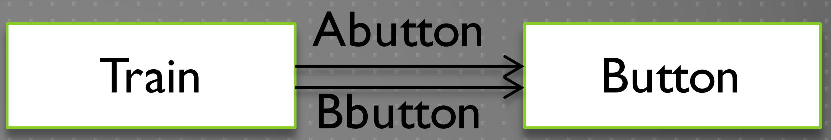
```
public class Train {
     ...
     protected void addStop(ButtonPressedEvent stop){
          // update nextStop
          ...
     }
     ...
}
```

ASSOCIATION

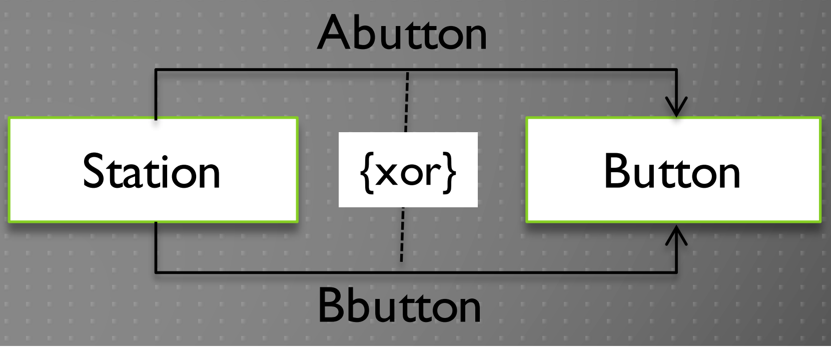
- Indicates a stronger relationship
  - E.g., class A **has a** class B
- Use number labels to indicate multiplicity
  - Use * to indicate arbitrary cardinality
- You can also explicitly name the associations
- And make them conditional



Station
1
1
Button
2
Train
1



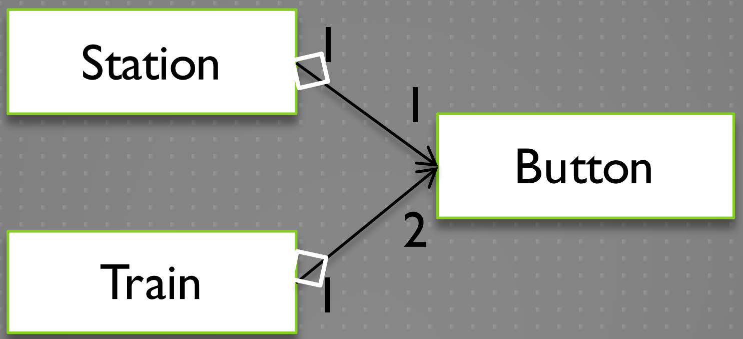
Train
Abutton
Bbutton
Button



Abutton
Station
{xor}
Button
Bbutton

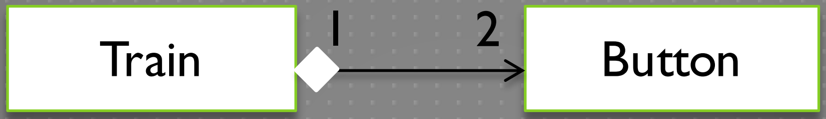
AGGREGATION

- 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)



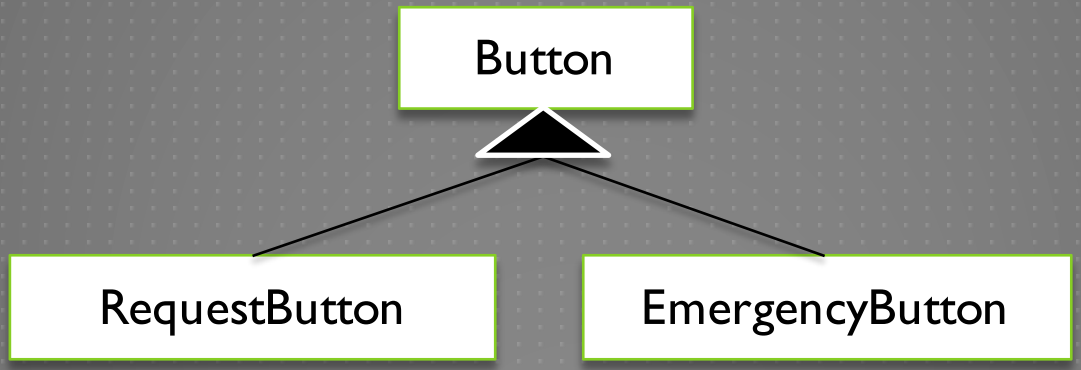
COMPOSITION

- 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 C++, aggregation is usually defined by pointers/references, while composition is defined by containing instances
  - In Java, composition is often indicative of the inner class style relationship



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

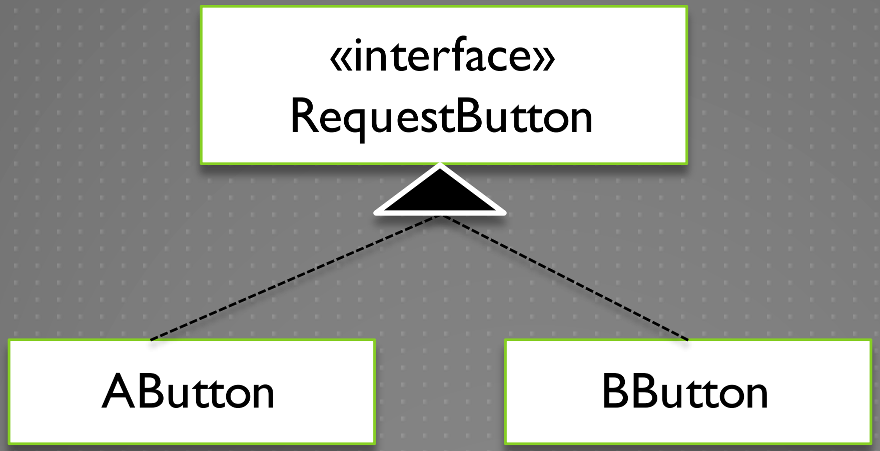

Button

RequestButton

EmergencyButton

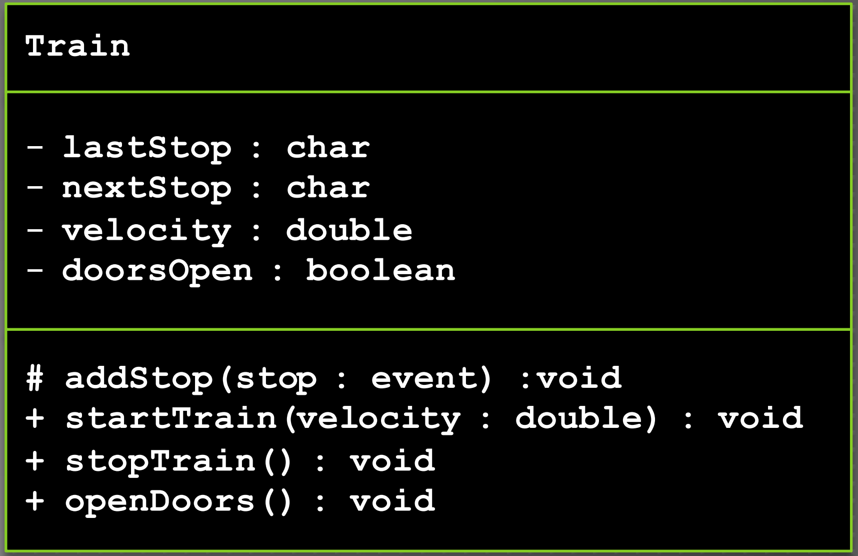
REALIZATION

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



OPERATIONS IN CLASS DIAGRAMS

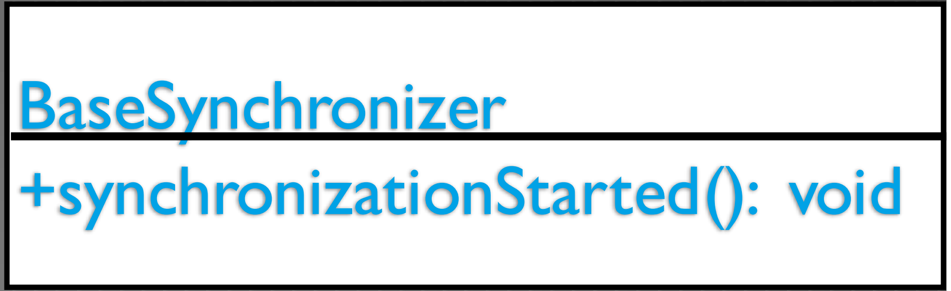
- Operation descriptions include
  - Visibility
    - Public +
    - Private -
    - Protected #
    - Package ~
  - Parameter list
    - Direction (in/out)
    - Name
    - Type
    - Multiplicity
  - Polymorphism
  - Abstract operations (italics)



```
Train

- lastStop : char
- nextStop : char
- velocity : double
- doorsOpen : boolean

# addStop(stop : event) :void
+ startTrain(velocity : double) : void
+ stopTrain() : void
+ openDoors() : void
```

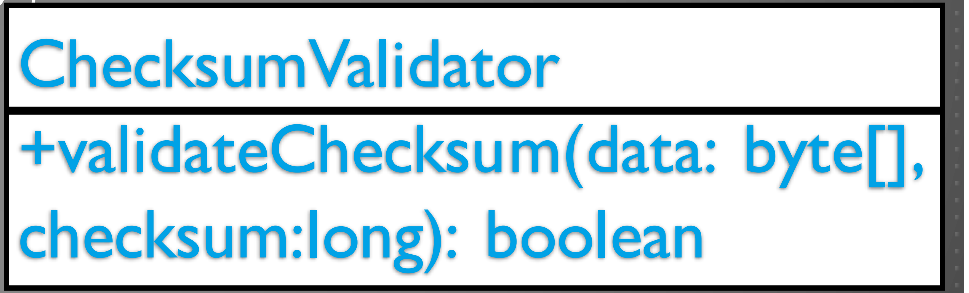
OPERATIONS

```
public class BaseSynchronizer {

    public void synchronizationStarted() {

    }
}

public class ChecksumValidator {

    static public boolean
validateChecksum(byte□ data, long checksum) {

    }
}
```



BaseSynchronizer

+synchronizationStarted(): void

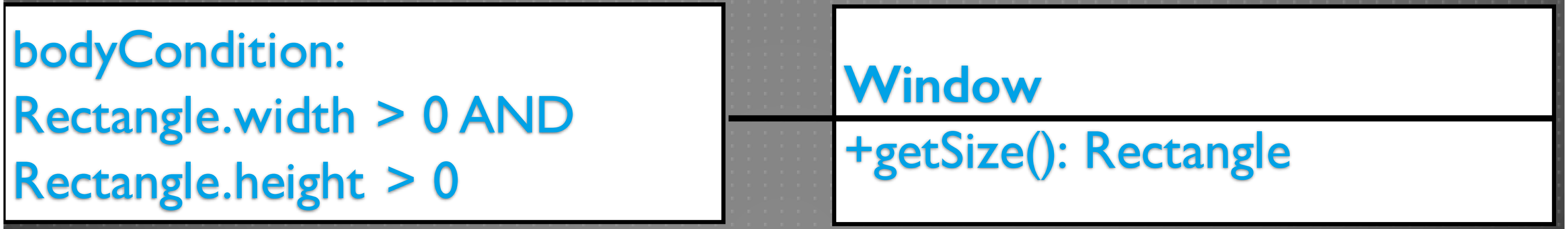


ChecksumValidator

+validateChecksum(data: byte[], checksum:long): boolean

CONSTRAINTS ON OPERATIONS

- *pre condition:* express what the state of the system must be before the associated operation can be invoked.
- *post condition:* express what the state of the system will be after the operation completes.
- *body condition (invariants):* express constraints on the method. It must be overridden by subclasses



bodyCondition:
Rectangle.width > 0 AND
Rectangle.height > 0

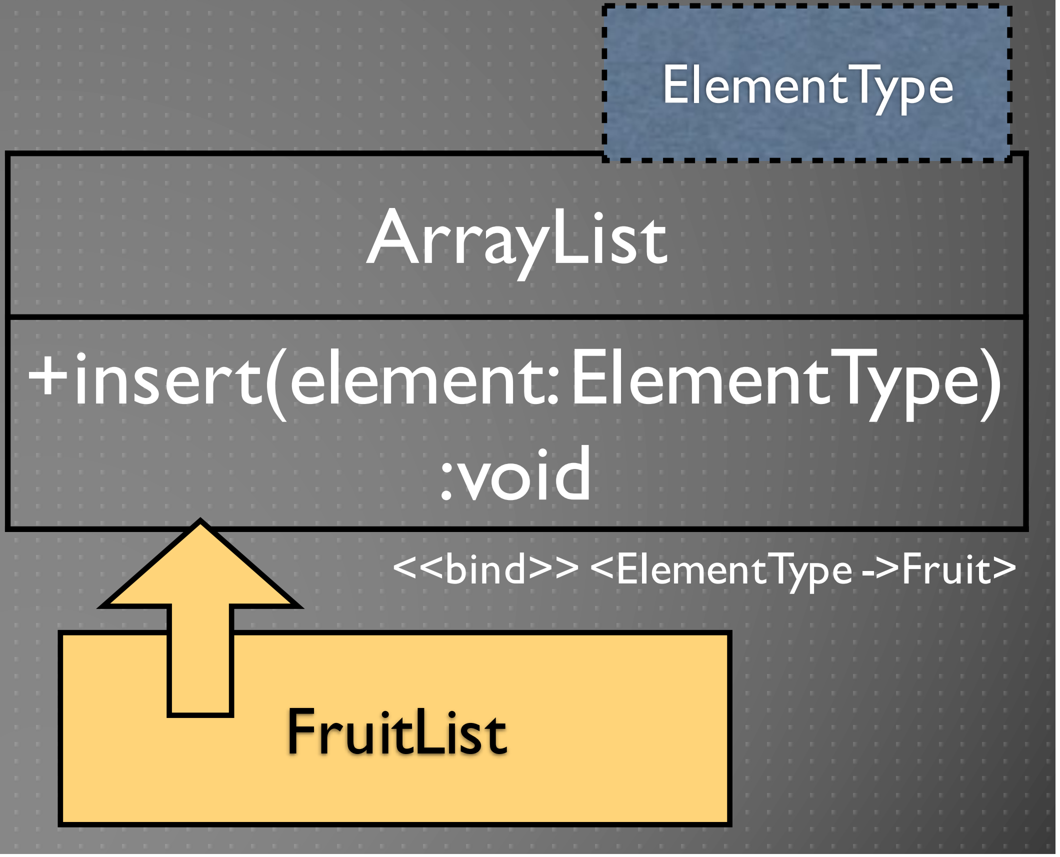
Window

+getSize(): Rectangle

TEMPLATE CLASS

- templates (generic types) allow a developer to design a class without specifying the exact types on which the class operates.

```java
import java.util.ArrayList;

public class FruitList
 extends ArrayList<Fruit> {
}
```



ElementType

ArrayList

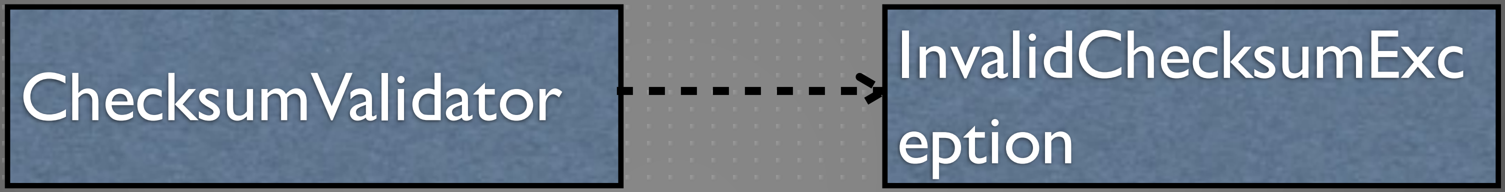
+insert(element: ElementType) :void

<<bind>> <ElementType ->Fruit>

FruitList

THINK PAIR SHARE

```
public class ChecksumValidator {
    public boolean execute() {
        try {
            this.validate();
        }catch (InvalidChecksumException e){
                ...
        }
        return true;
    }
    public void validate() throws
InvalidChecksumException { ....
    }
}
```

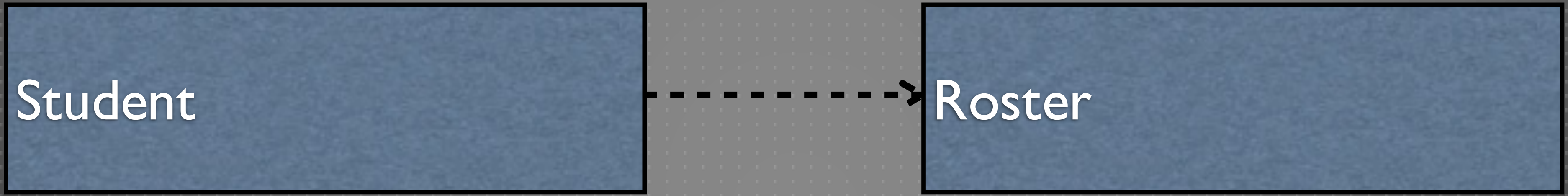


ChecksumValidator

InvalidChecksumException

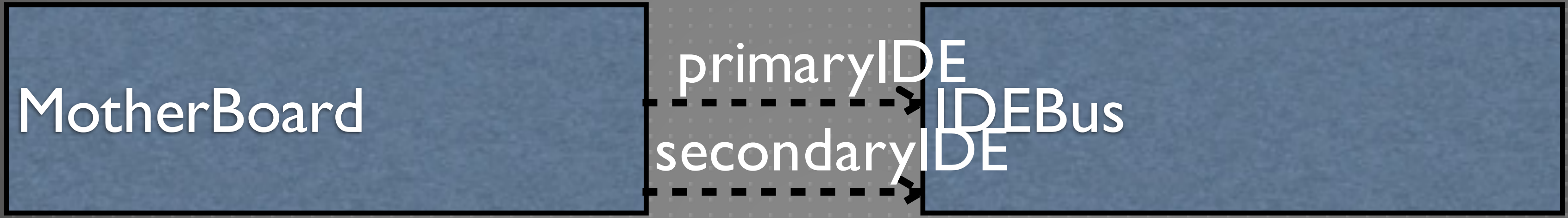
THINK PAIR SHARE:

```
public class Student {

   Roster roster;

   public void 
storeRoster(Roster r) {
      roster=r;
   }
}
```



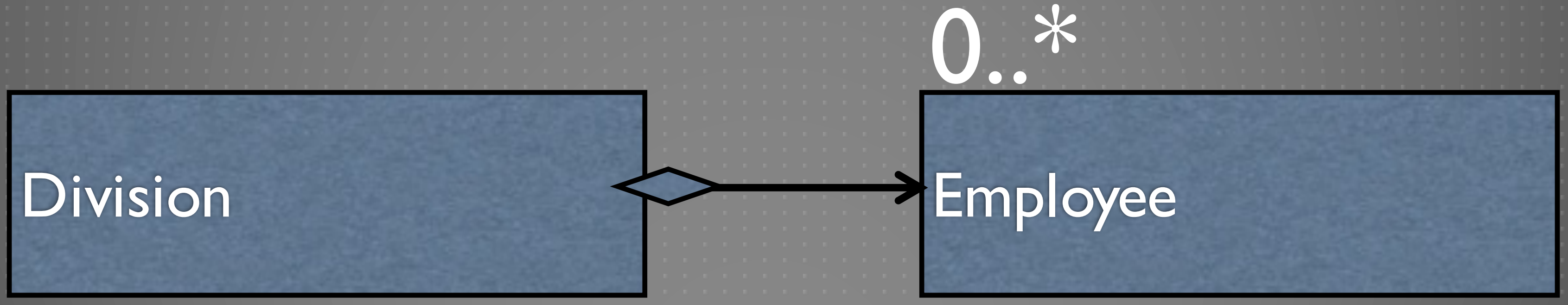
THINK PAIR SHARE

```
public class MotherBoard {
  private class IDEBus {...

  }
  IDEBus primaryIDE;
  IDEBus secondaryIDE;

}
```



THINK PAIR SHARE

```
import java.util.ArrayList;
import java.util.List;
public class Division {

   private List<Employee> division = new 
ArrayList<Employee>();
   private Employee[] employees = new Employee
[10];
}
```



FAQ

- Body conditions
- Association vs. Dependency
- Association vs. Aggregation Notations

SEQUENCE DIAGRAM

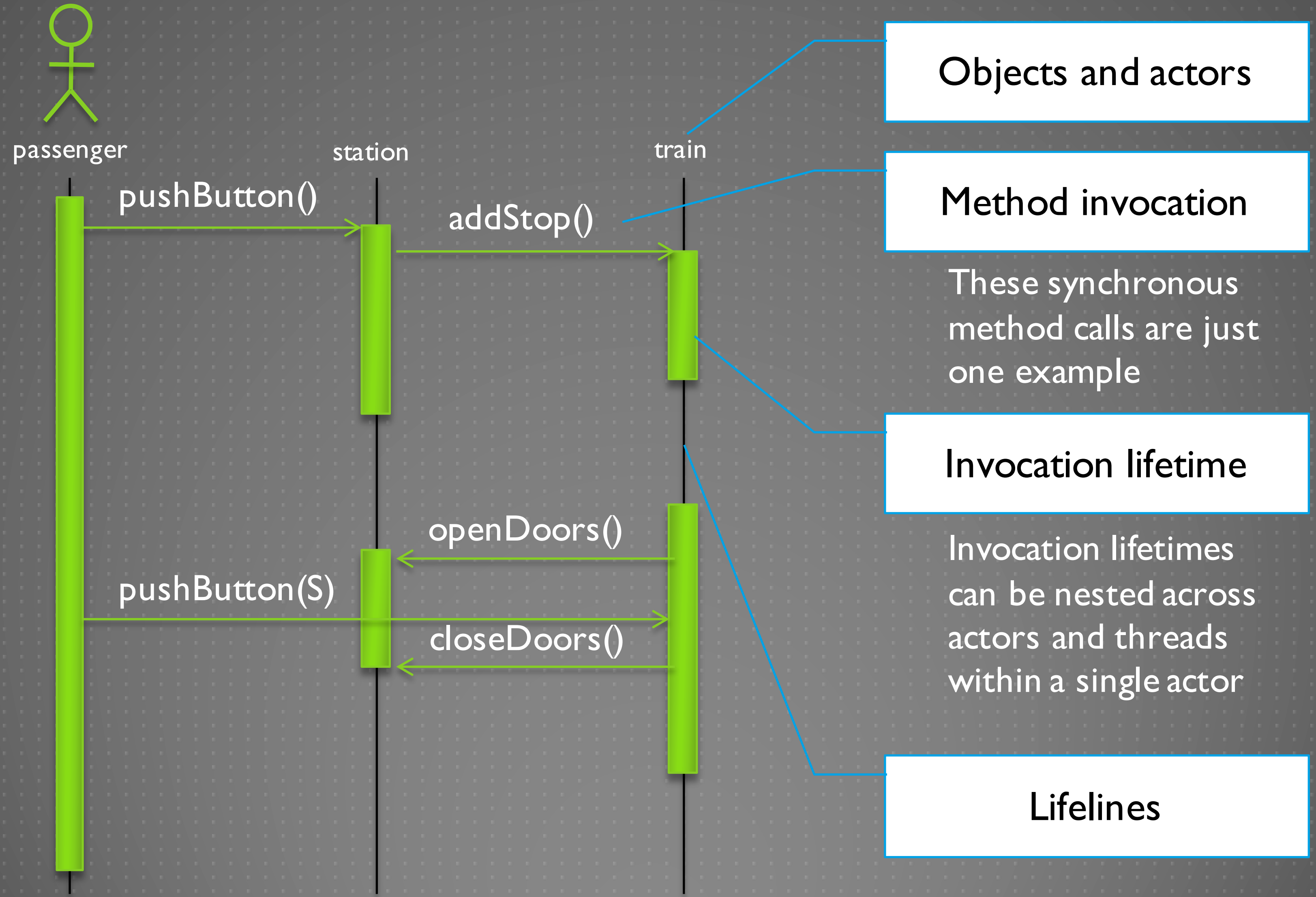
INTERACTION DIAGRAMS

- 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



Objects and actors

Method invocation

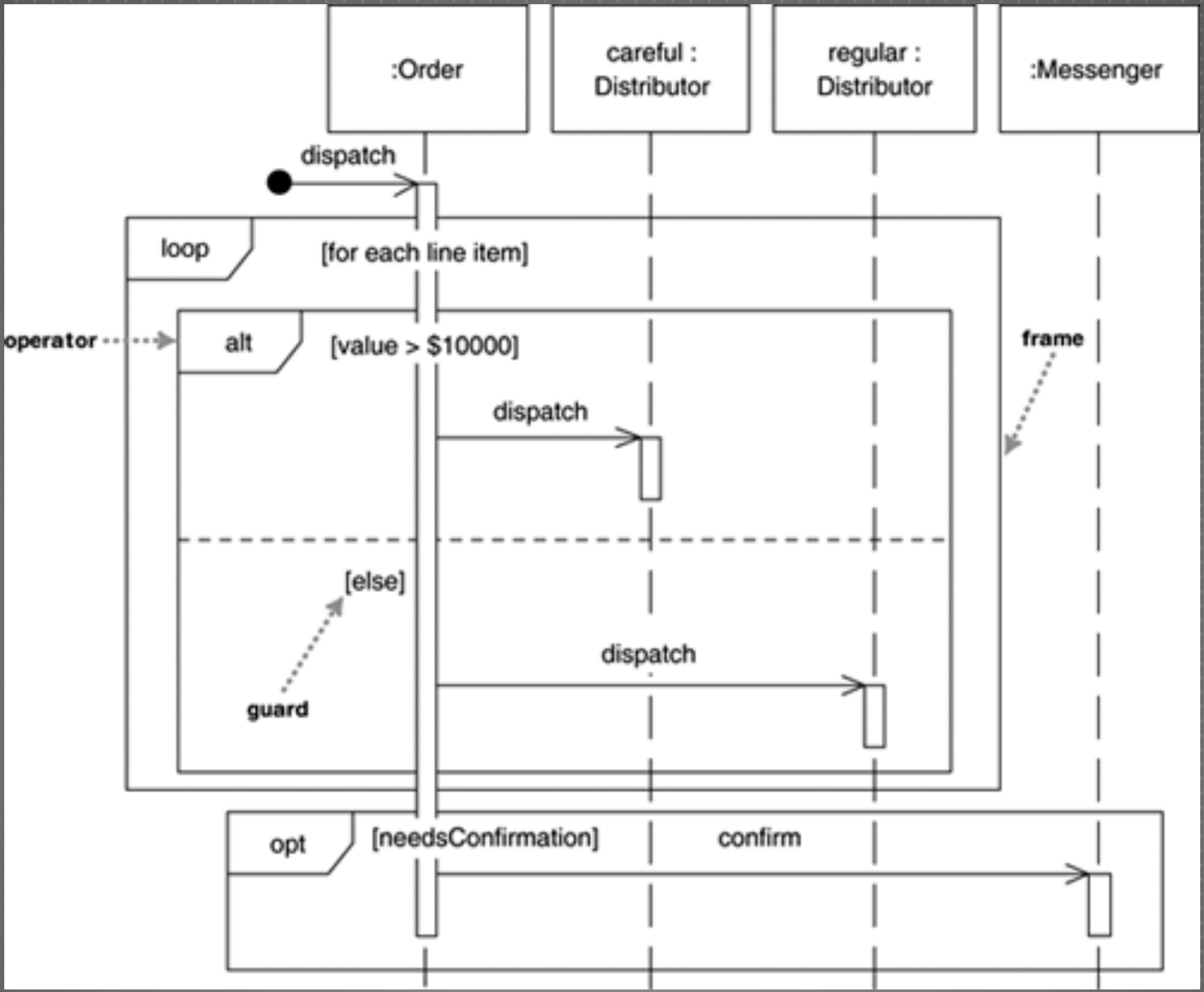
These synchronous method calls are just one example

Invocation lifetime

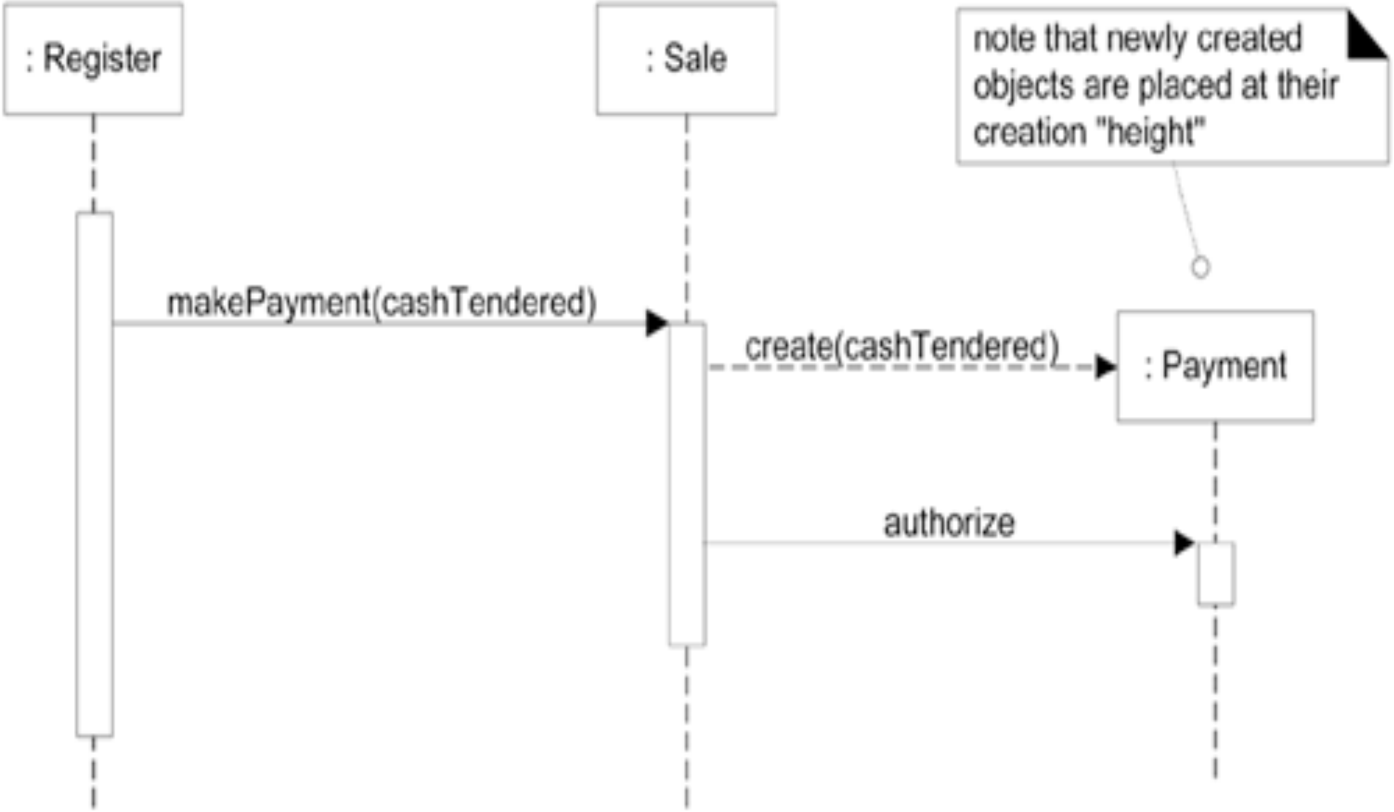
Invocation lifetimes can be nested across actors and threads within a single actor

Lifelines

LOOPS AND ALTERNATIVES

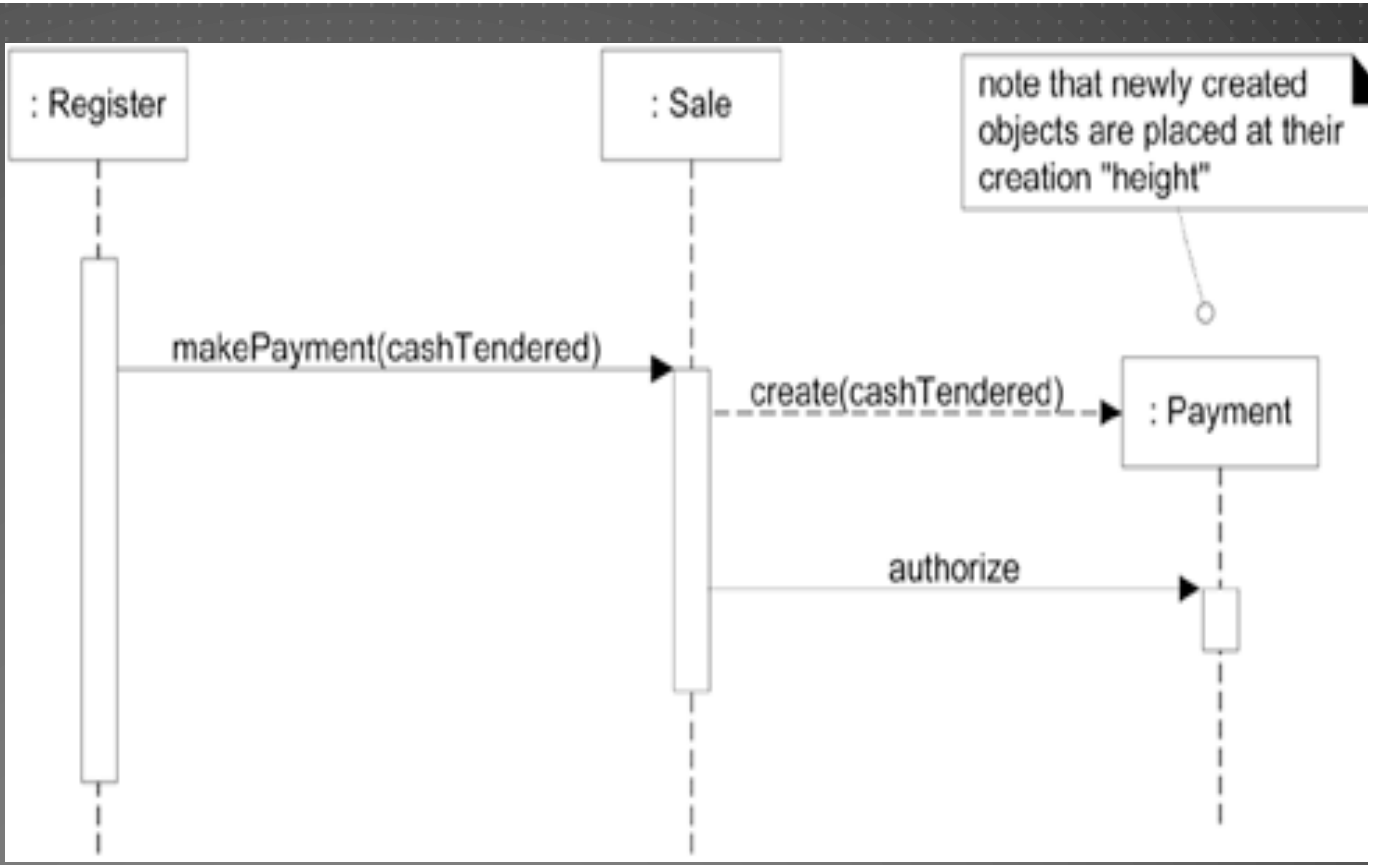


EXAMPLE



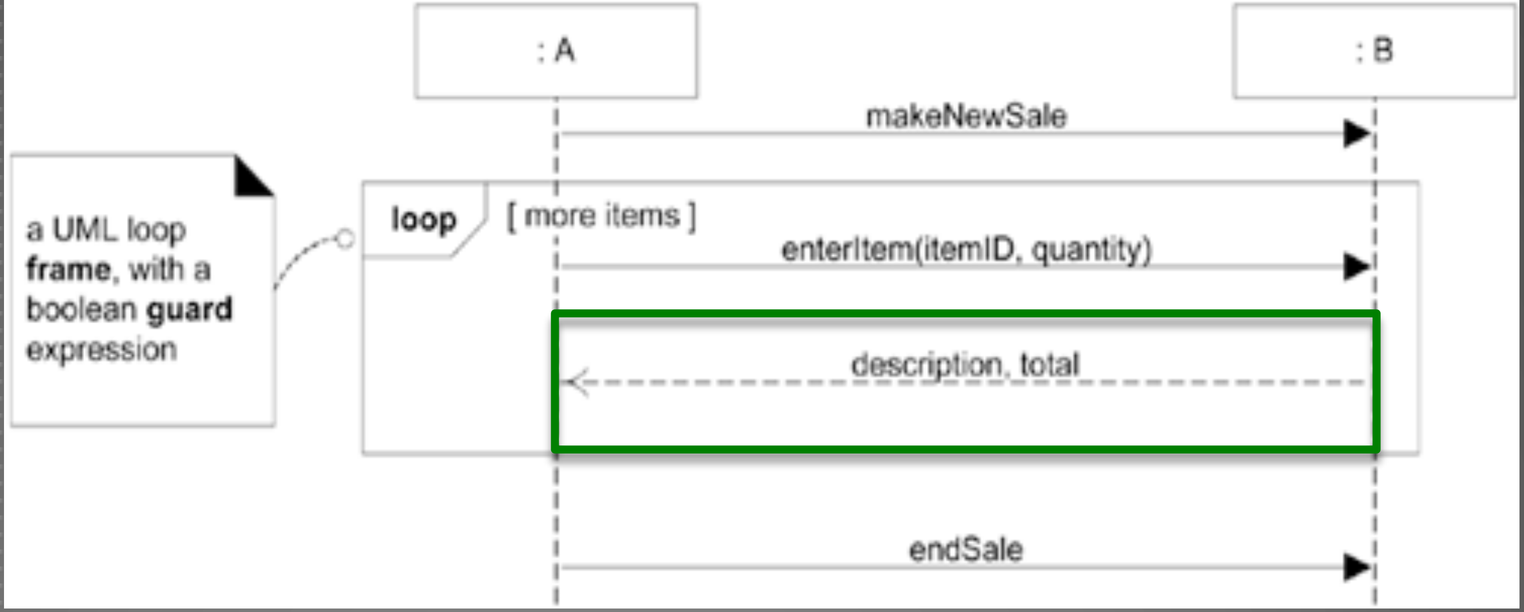
: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

EXAMPLE


: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

```
public class Register {
   public void method (Sale s) {

      s.makePayment(cashTendered);
   }
}


public class Sale {
   public void makePayment(int amount) {
      Payment p = new Payment(amount);
      p.authorize();
   }
}
```

EXAMPLE


: A
: B
makeNewSale
a UML loop frame, with a boolean guard expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale

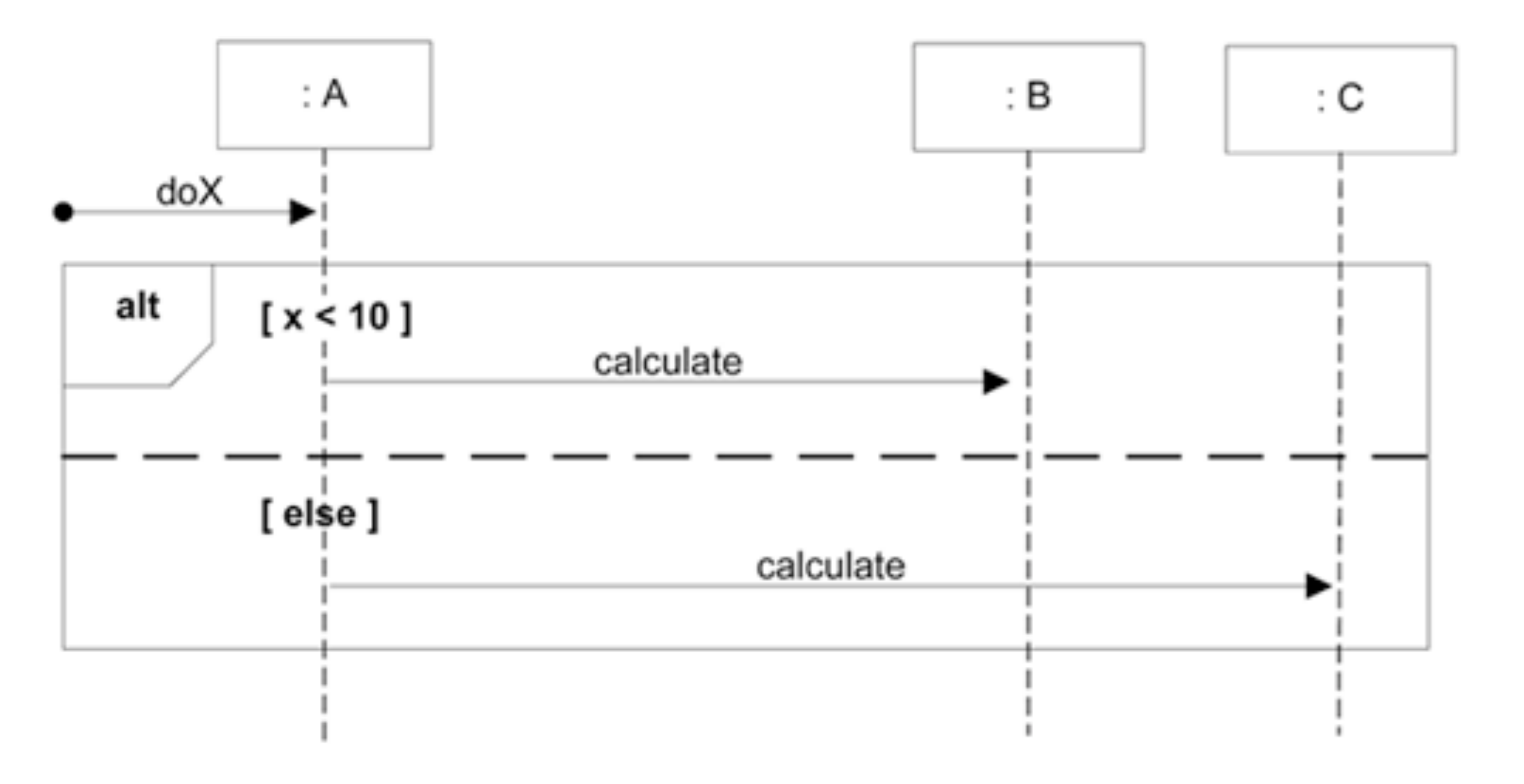
```
public class A {
   List items =null;
   ...
   public void noName (B b) {
      b.makeNewSale();
      for (Item item: getItems()) {
         b.enterItem(item.getID(), quantity);
         total = total +...
         description =...;
      }
      b.endSale();
   }
}
```

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

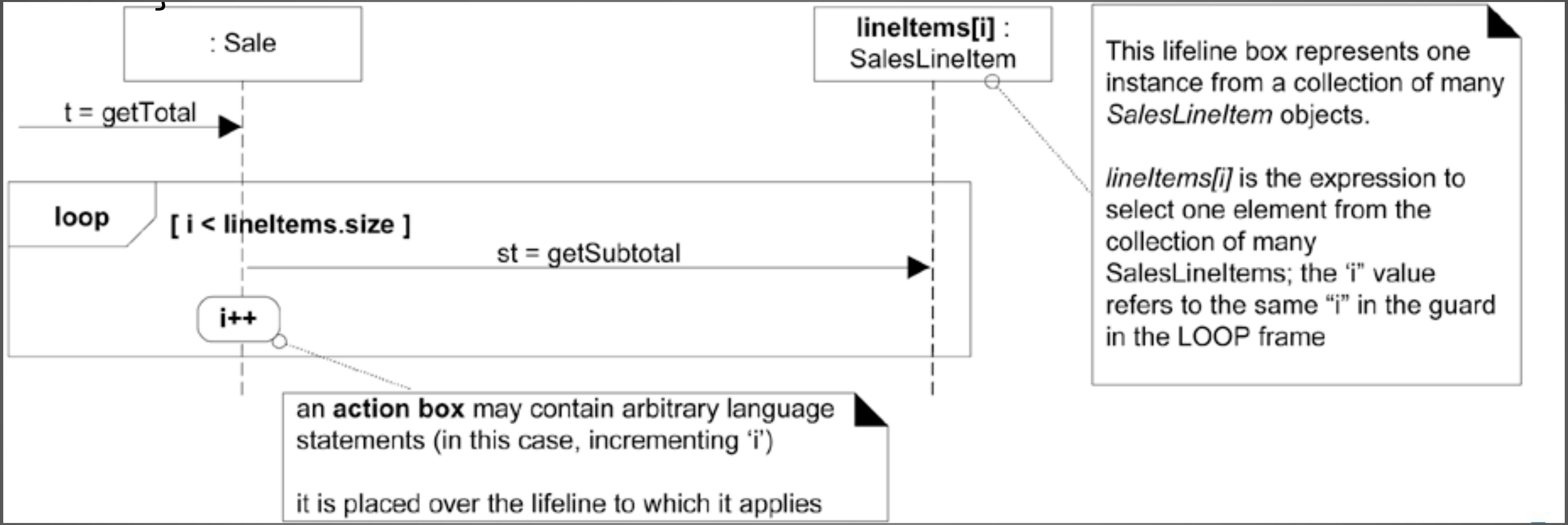
- Write pseudocode that has the same meaning as the following sequence diagram



ANSWER

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

ANOTHER ONE…



This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

lineItems[i] is the expression to select one element from the collection of many SalesLineItems; the 'i" value refers to the same "i" in the guard in the LOOP frame

an **action box** may contain arbitrary language statements (in this case, incrementing 'i')

it is placed over the lifeline to which it applies

ANSWER

```
public class Sale {
    private List<SalesLineItem> lineItems =
        new ArrayList<SalesLineItem>();

    public Money getTotal() {
        Money total = new Money();
        Money st= null;

        for (SalesLineItem lineItem : lineItems) {
            st = lineItem.getSubtotal();
            total.add(subtotal);
        }
        return total;
    }
}
```

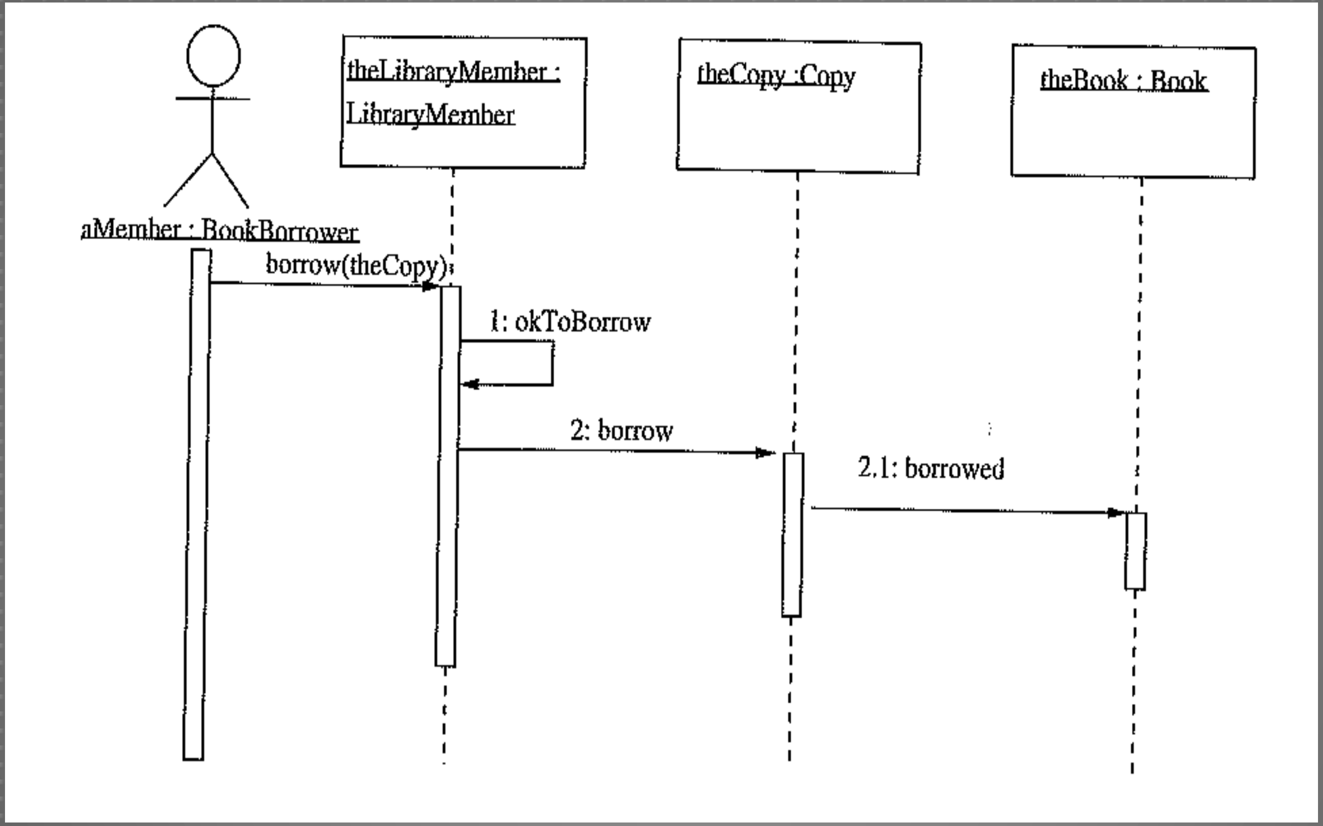
REVIEW QUESTION: THE OTHER WAY AROUND

- 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();
    }
}
```

THINK PAIR SHARE

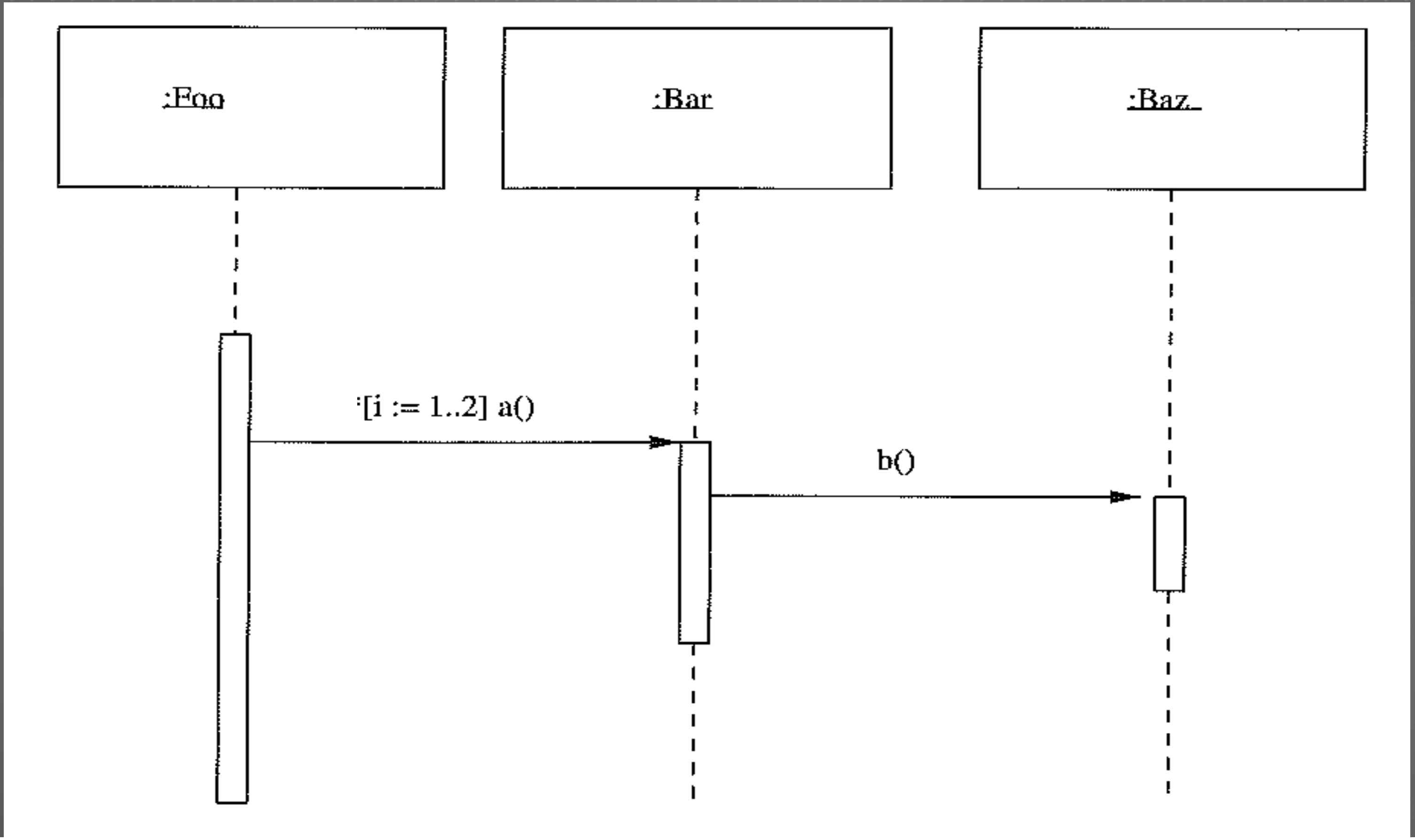
- Write code interpreting the following sequence diagram



```
// the following is executed from BookBorrower type aMember
instance
   LibraryMember theLibraryMember = null;
theLibraryMember.borrow(theCopy);
}
public class LibraryMember {
   public boolean borrow (Copy theCopy) {
       ...
       okToBorrow();
       theCopy.borrow();
   }
   public boolean okToBorrow() { ...
   }
}
public class Copy {
   Book theBook;
   public void borrow() {
       theBook.borrowed();
   }
}
Public class Book { public boolean borrowed(){} }
```

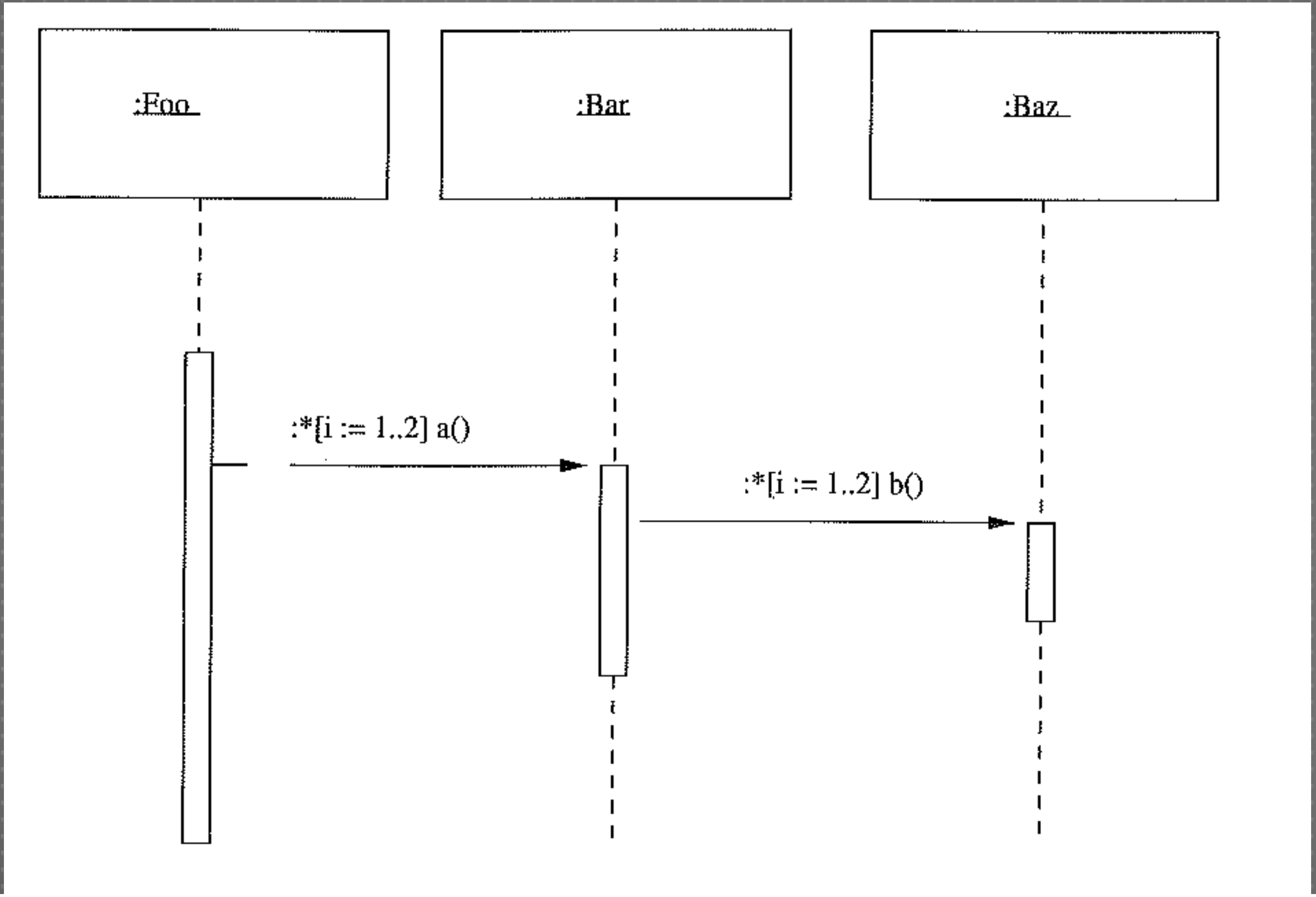
THINK PAIR SHARE

- What are the resulting sequence of method calls?



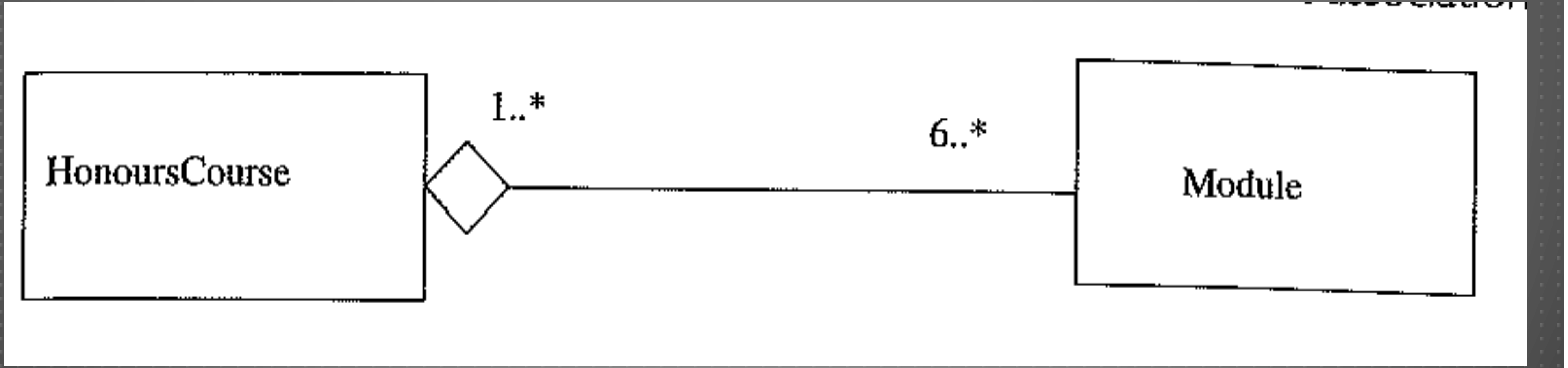
THINK PAIR SHARE

- What are the resulting sequence of method calls?



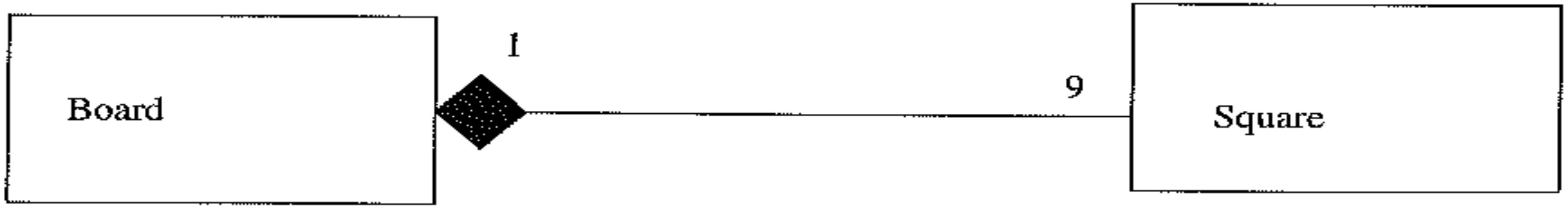
THINK PAIR SHARE

- Write code interpreting the diagram



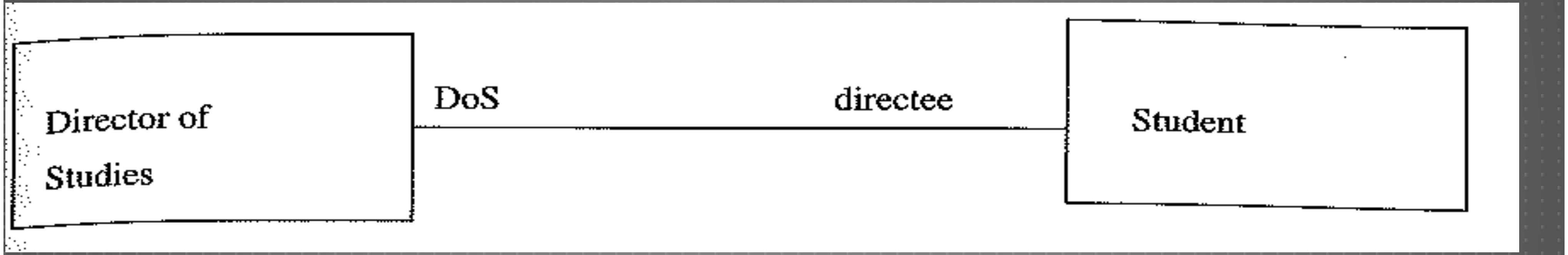
THINK PAIR SHARE

- Write code interpreting the diagram



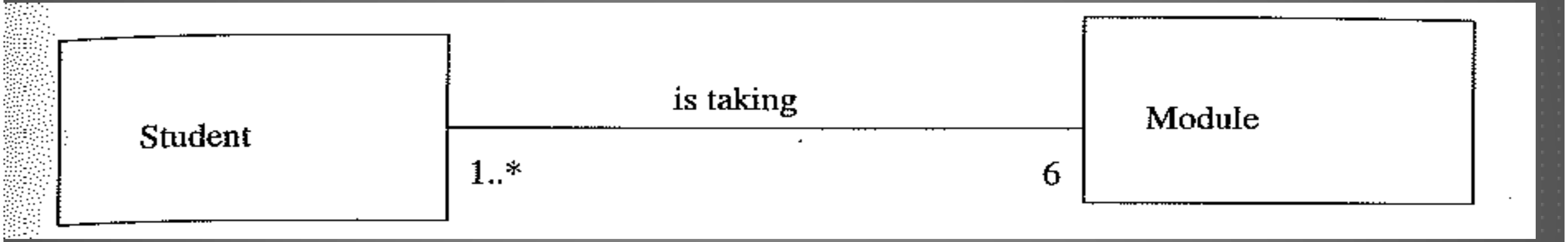
THINK PAIR SHARE

- Write code interpreting the diagram

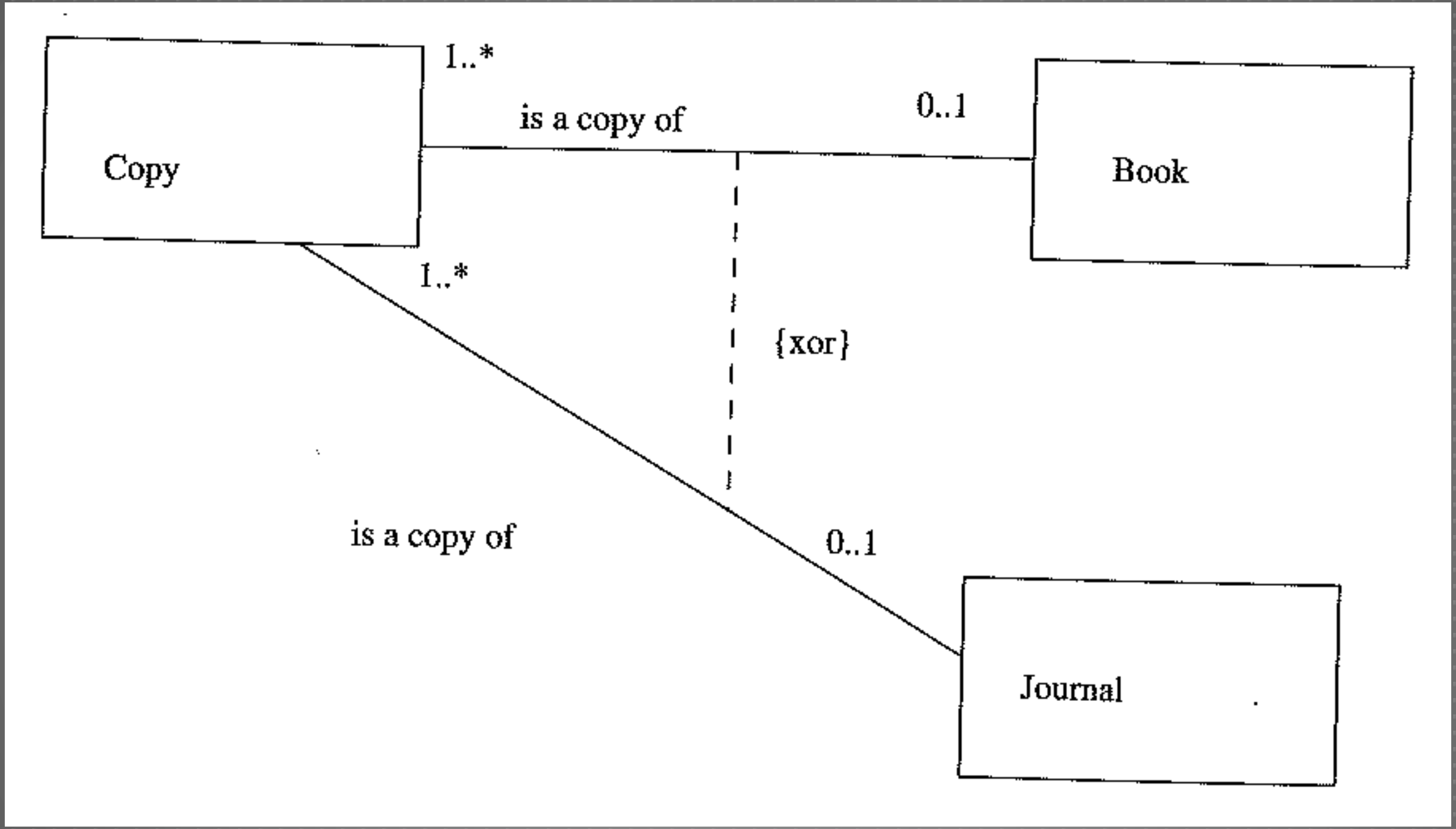


THINK PAIR SHARE

- Write code interpreting the diagram



THINK PAIR SHARE



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…

RECAP

- UML class diagrams and sequence diagrams are notations for expressing low-level design in OO programs.
    - Practice interpreting UML class and sequence diagrams
    - Practice reverse-engineering UML diagrams from code

PREVIEW

- Lab Sections
  - New Application Proposal Overview (Part A)
- Read "On the criteria…"
- Do the lab tutorial on UML as a practice at home.

QUESTIONS?