Unified Modeling Language (UML) & the Software Development Lifecycle

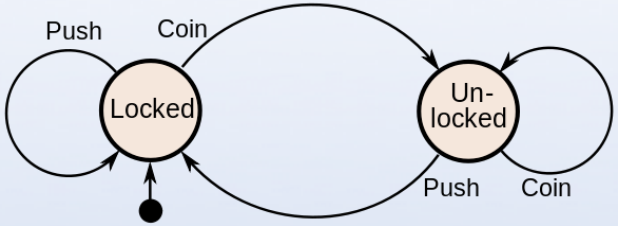
UML Introduction

SDLC models

Software Requirements

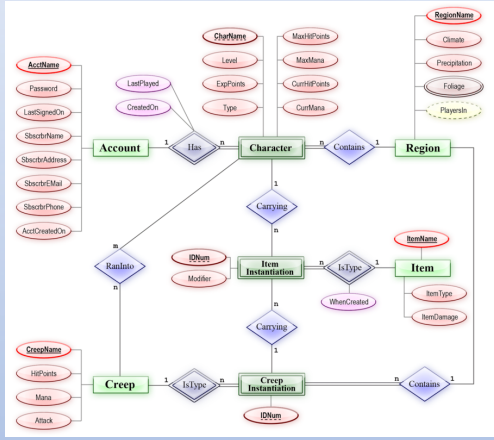
Static and dynamic UML diagrams

Modeling Software



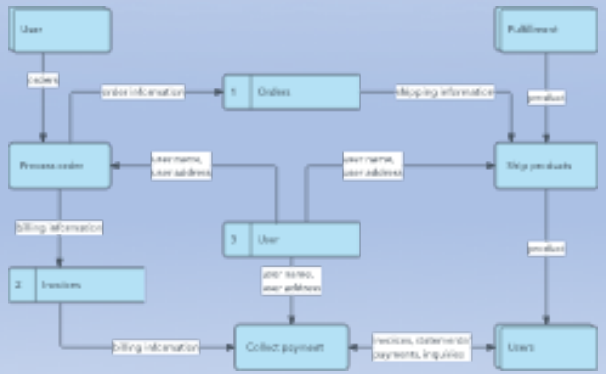
https://en.wikipedia.org/wiki/Finite-state_machine

- Describing a system at a high level of abstraction
  - A model of the system
  - Enables communication, design, assessment of requirements
  - Helps manage software complexity



https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

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

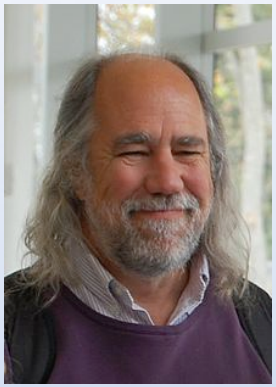


https://www.lucidchart.com/pages/data-flow-diagram

History of UML

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

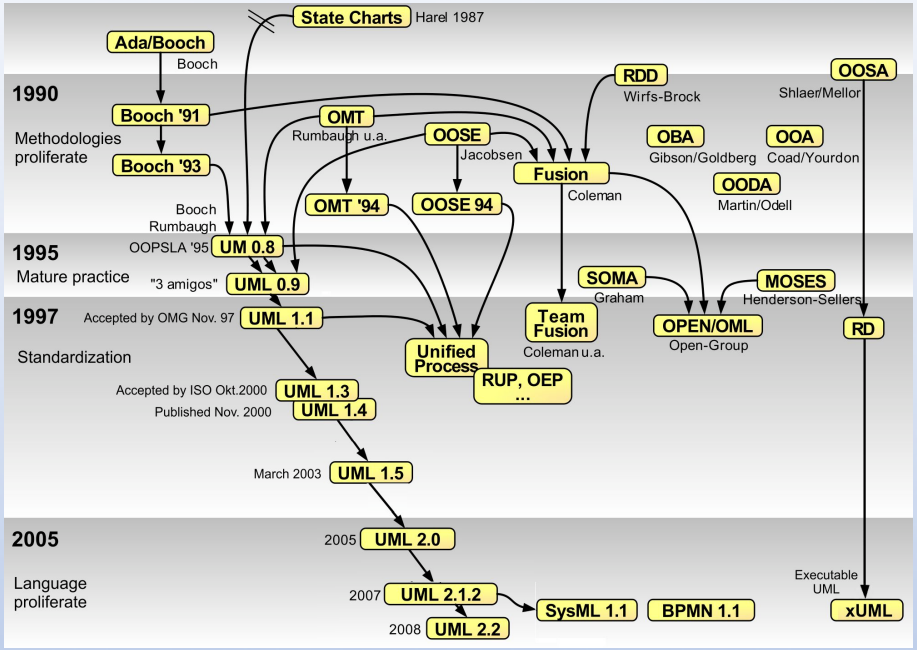
- 1990s
  - Three leading OO notations decide to combine
    - Grady Booch (BOOCH)
    - Jim Rumbaugh (OML: Object Modeling Language)
    - Ivar Jacobson (OOSE: Object Oriented Software Engineering)
  - Why?
    - Natural evolution towards each other
    - Effort to set an industry standard



UML

- Unified Modeling Language
  - Enormous language
- Provides a **common**, **simple**, **graphical** representation of software requirements, design and implementation
- Allows **developers**, **architects**, and **users** to discuss the workings of the software
- UML Guidelines:
  - Nearly everything in UML optional
  - Designed to be extended and customized

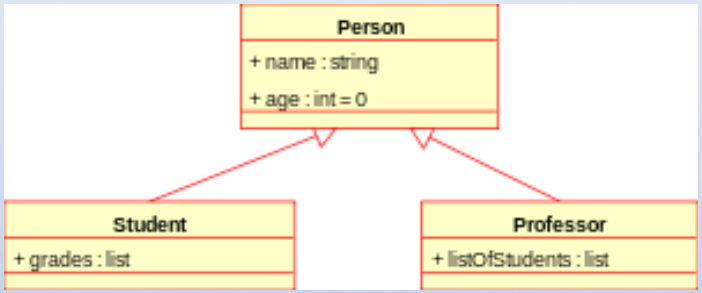
- http://www.omg.org



https://commons.wikimedia.org/wiki/File:OO_Modeling_languages_history.jpg

Model Types 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 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)

Software Development Lifecycle

SDLC Models: Waterfall, Agile

Waterfall Model

- **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

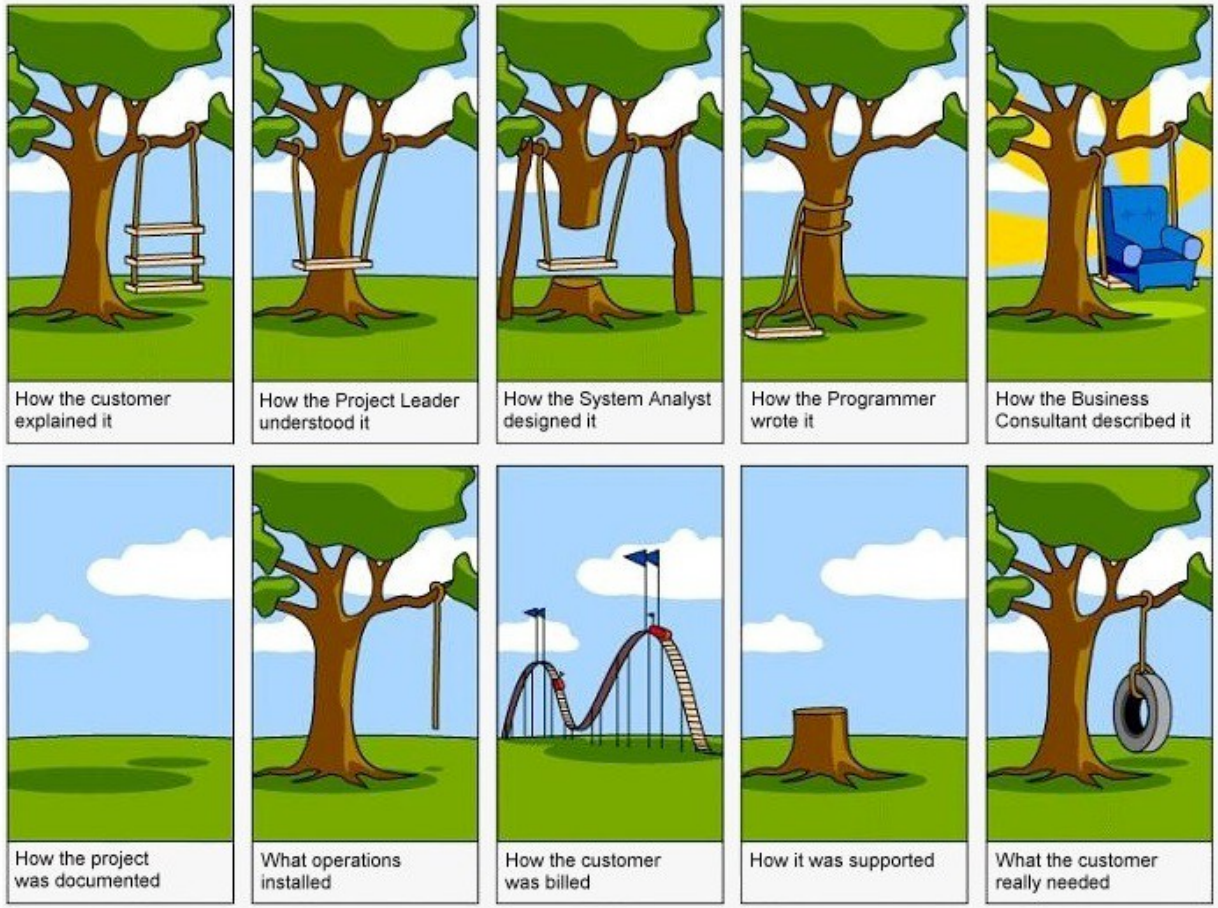
Agile Approaches

- Iterative, incremental; easy adaptation to changing requirements
  - Iterations or sprints: cross-functional team works through all phases to produce a working product
    - timeboxes: usually one to four weeks
    - working product to demo to stakeholders
  - frequent releases, short development cycles, frequent and efficient communication
- Agile software development methods: XP, Scrum
- Scrum: 2-4 week sprints, daily scrums (15 minute standup meetings to plan and track progress), product backlog (ordered list of product requirements, often in form of user stories), sprint backlog (work to be done during sprint), no changes during sprints
- Extreme Programming: iterations typically 1-2 weeks, practices: pair programming ("extreme" code review), test-driven development ("extreme" testing), unit testing of all code, continuous integration, frequently run acceptance tests, small releases
- Daily Standup/Scrum: each team member answers:
  - what did you do yesterday?
  - what will you do today?
  - what is standing in your way?
    - Ex: I need help debugging a problem with ____, I'm having trouble learning ____ and need help with it.

Requirements

Make sure software does what the customer wants

Why are Requirements Important?



https://medium.com/omarelgabrys-blog/requirements-engineering-introduction-part-1-6d49001526d3

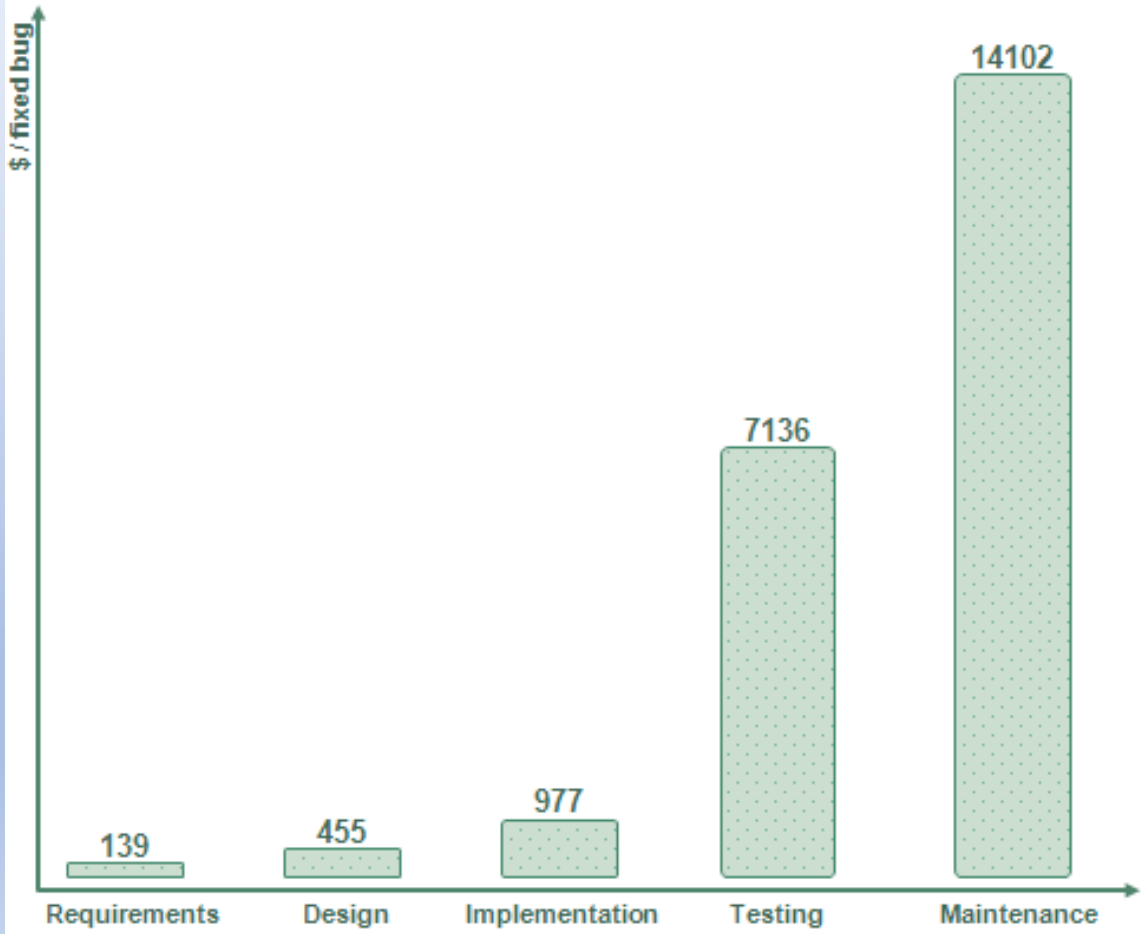
Software Requirements

- Requirements specify what to build
- Functions, performance, qualities and constraints – the "what" to guide the design
- "what" not "how"
- Tell the problem, not the solution
- Reflect system features, not software design
- Goals:
  - understand precisely what is required of software
  - communicate this understanding to all (developers, stakeholders)
  - control production to ensure that system meets specs

Why are Requirements Important?

- 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.
- Roles of requirements
  - customer: show what should be delivered; contractual base
  - managers: scheduling/progress indicator
  - designers: provide a spec to design
  - coders: list range of acceptable implementations/output
  - QA/testers: basis for testing, validation, verification

Identify Problems Early



Classifying Requirements

- Functional requirement: describes behaviors (functions or services) of system – what system should do
  - "The user can search either all databases or a subset."
  - "Once the dog door has opened, it should close automatically if the door isn't already closed."
- Non-functional requirements: constraints on the system, qualities it will have
  - often global
  - ilities: dependability, reusability, portability, performance, security, response time, legal/regulatory requirements
  - Examples: "The system shall not disclose any personal user information."

Functional or Non-Functional?

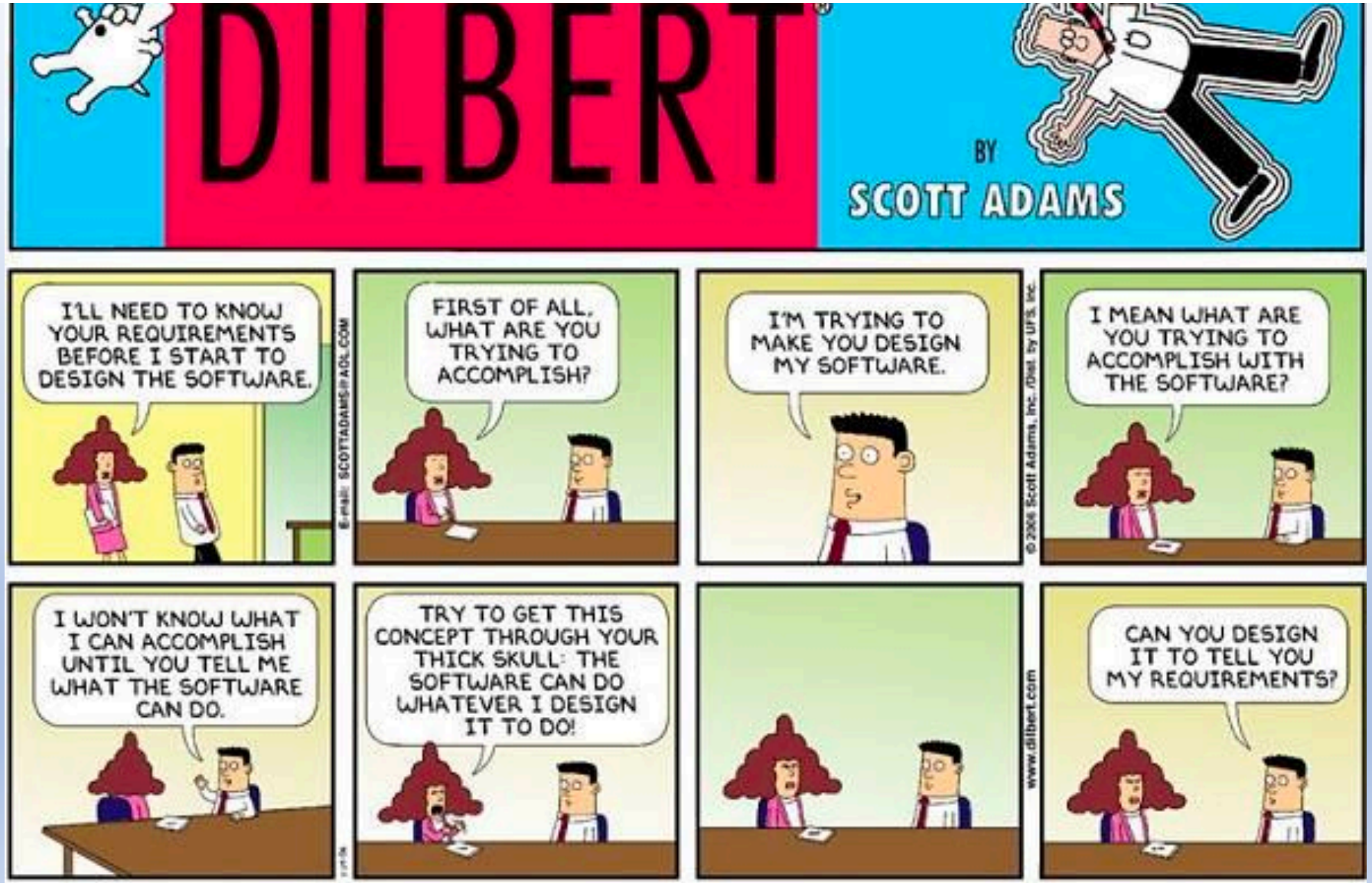
- System will run 24 hours a day, 7 days a week
- An online help system is required
- All code will be written in Java
- The database will be Oracle 8

Gathering Requirements From Users

- The #1 reason that projects succeed is user involvement
  -- Standish Group survey of over 8000 projects

- Easy access to end users is one of the critical success factors in rapid-development [agile] projects.
  -- Steve McConnell

- The most difficult part of requirements gathering is not the act of recording what users want; it is the exploratory activity of helping users figure out what they want.
  -- Steve McConnell


DILBERT
BY
SCOTT ADAMS
I'LL NEED TO KNOW YOUR REQUIREMENTS BEFORE I START TO DESIGN THE SOFTWARE.
FIRST OF ALL, WHAT ARE YOU TRYING TO ACCOMPLISH?
I'M TRYING TO MAKE YOU DESIGN MY SOFTWARE.
I MEAN WHAT ARE YOU TRYING TO ACCOMPLISH WITH THE SOFTWARE?
I WON'T KNOW WHAT I CAN ACCOMPLISH UNTIL YOU TELL ME WHAT THE SOFTWARE CAN DO.
TRY TO GET THIS CONCEPT THROUGH YOUR THICK SKULL: THE SOFTWARE CAN DO WHATEVER I DESIGN IT TO DO!
CAN YOU DESIGN IT TO TELL YOU MY REQUIREMENTS?
www.dilbert.com

How to express requirements?

- Software requirements specification document
- Use cases
- Use case diagrams
- Feature list
- User stories
  - break requirements into chunks for planning purposes
  - intended to be buildable in one iteration

User Story

- Short and simple description of a feature from the POV of user/customer
- Format: As a <type of user>, I want <goal> so that <reason>.
- Examples:
  - As a customer, I want a one-click purchase option so I can save time when buying things.
  - As a legally blind user, I want to zoom in on blocks of text so I can read them.

Use Cases

- **use case**: written description of user's interaction with software to accomplish some goal
  - describes piece of functionality of system
  - defines the interactions between actor and the system
  - Sequence of steps (usually 3-9) describing scenario
  - written from actor's point of view (not system's)
  - focuses on interaction, not internal system activities
  - doesn't describe the GUI in detail
- Benefits:
  - discover, document functional requirements
  - agreement among stakeholders as to system's responsibilities
  - helps provide list of error cases to test

Use Cases vs. Internal Features

Phone software:

- Use cases - Point of view: user
  - call a person
  - receive a call
  - send a message
- Internal function – Point of view: developer
  - transmit/receive data
  - phone book management
  - energy (battery)

Use Cases and Requirements

- Which of these requirements would likely be represented in a use case?
  - Number of simultaneous users will be at most 100.
  - A customer can only have one sales contact.
  - Web site uptime requirement is 99%.
- Answer: None.
  - These are <u>non-functional</u> requirements – use cases wouldn't mention them. User doesn't see them directly.

Actors, Stakeholders and Goals

- What is an actor and a stakeholder in a use case?
- **actor** – anything external that interacts with the system
  - human user, device, another system
  - primary actor: initiates interaction to achieve a goal, aka external initiator
  - supporting/secondary actor: performs subgoals in use case
- **stakeholder** – anyone interested in system
  - supplier, vendor, project manager
  - may not interact with system
- **goal** – action that primary actor wants to accomplish

Use Cases: Different Forms

- List of actors and their goals
- Informal (paragraph describing use)
- Formal: actor, goal, step by step description
- UML use case diagrams

Use Case List

- Can be useful to create a list of table of primary actors and their goals (use cases they start.) A use case diagram can then capture this information.

Actor	Goal
Library Patron	Search for a book
	Check out a book
	Return a book
Librarian	Search for a book
	Check availability
	Request a book from another library

Informal Use Case

- An informal use case is written as a paragraph describing the scenario/interaction
- Example:

➢ <u>Patron Loses a Book</u>

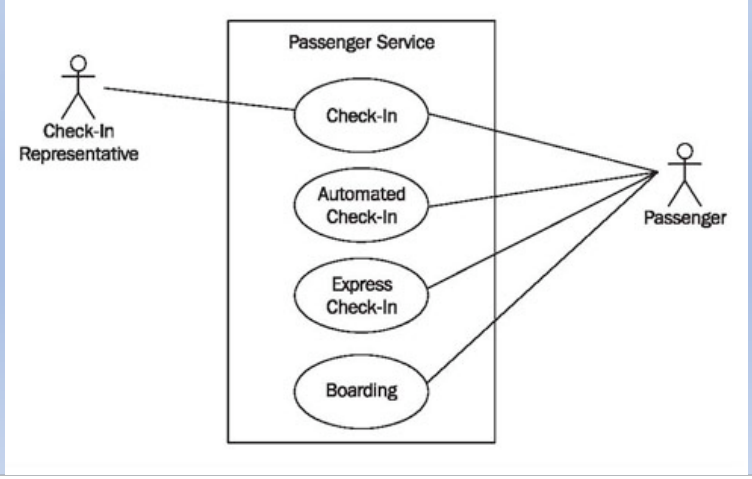
The library patron reports to the librarian that she has lost a book. The librarian prints out the library record and ask patron to speak to the head librarian, who will arrange for the patron to pay a fee. The system will be updated to reflect lost book, and patron's record is updated as well. The head librarian may authorize purchase of a replacement.

Use-Case Diagram

- use case: describes piece of functionality of system
- Helps development team visualize functional requirements
  - interaction of actors (e.g, humans who will interact with system) with system
  - relationship between different use cases
- Diagram usually shows groups of use cases – possibly all, or a collection of use cases with related functionality
- To show use case in use case diagram:
  - Draw oval in middle of diagram and put name of use case in oval (or below)
  - Box represents system boundary (actors are outside it)
  - An actor (system user) is drawn as stick figure to left or right in diagram
  - Lines represent relationships between actors and use cases

Use Case Diagrams

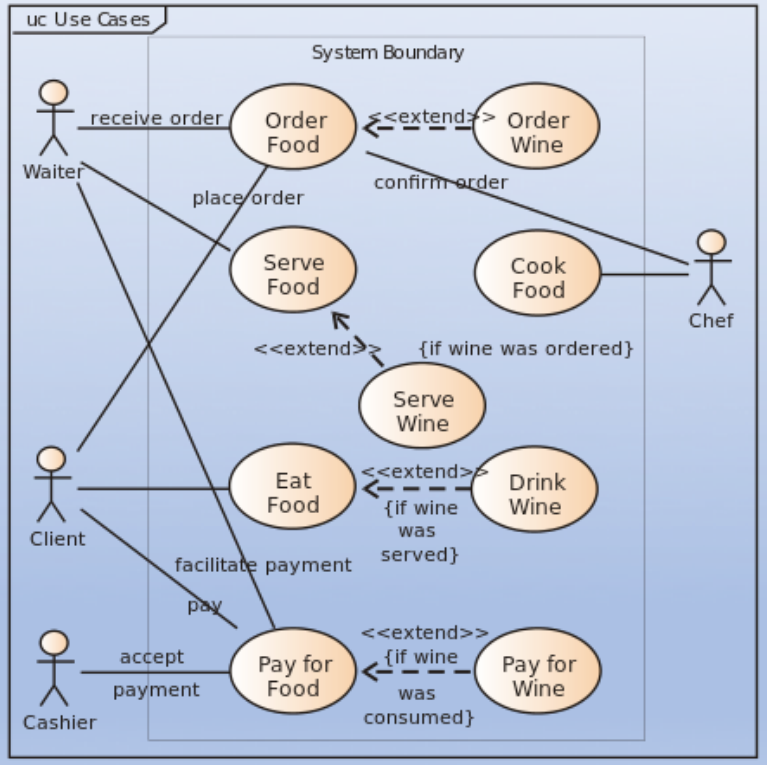
- Use case diagrams capture the requirements of a system from the user's perspective
  - The term *use case* refers to a particular piece of functionality that the system must provide (to a user)
  - Use cases are at a higher level of abstraction than other UML elements
- There will be one or more use-cases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



https://sourcemaking.com/uml/modeling-business-systems/external-view/constructing-use-case-diagrams

Use-Case Diagram

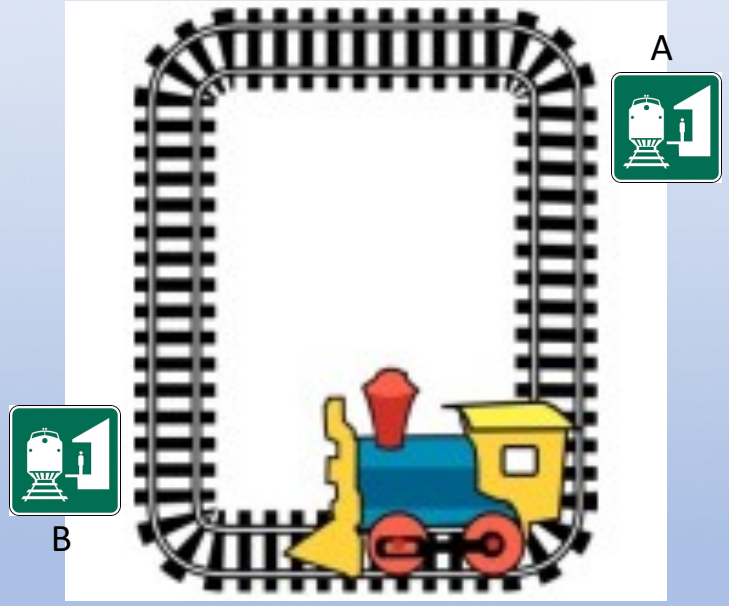
- Indicates high level functions of system
- Also indicates what system doesn't do, by absence of use cases
- Stakeholder can tell from use case diagram if necessary functionality is included in system



https://en.wikipedia.org/wiki/Use_case_diagram

Example Setup: 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



Example: Formal Use Case

- Name: Normal Train Ride
- Actors: Passenger
- Entry Condition: Passenger at station
- Exit Condition: Passenger leaves train
- 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: ??

Formal Use Case

Goal: Patron wishes to reserve a book using the online catalog
Primary actor: Patron
Precondition: Patron is at the login screen
Success end condition: Book is reserved
Failure end condition: Book is not reserved
Trigger: Patron logs into system

1. Patron enters account and password
2. System verifies and logs patron in
3. System presents catalog with search screen
4. Patron enters book title
5. System finds match and presents location choices to patron
6. Patron selects location and reserves book
7. System confirms reservation and re-presents catalog

Formal Use Case, continued

Extensions (error scenarios):

2a. Password is incorrect

    2a1. System returns patron to login screen

    2a2. Patron backs out or tries again

5a. System cannot find book

    5a1. ...

Another Example: Buy a Product

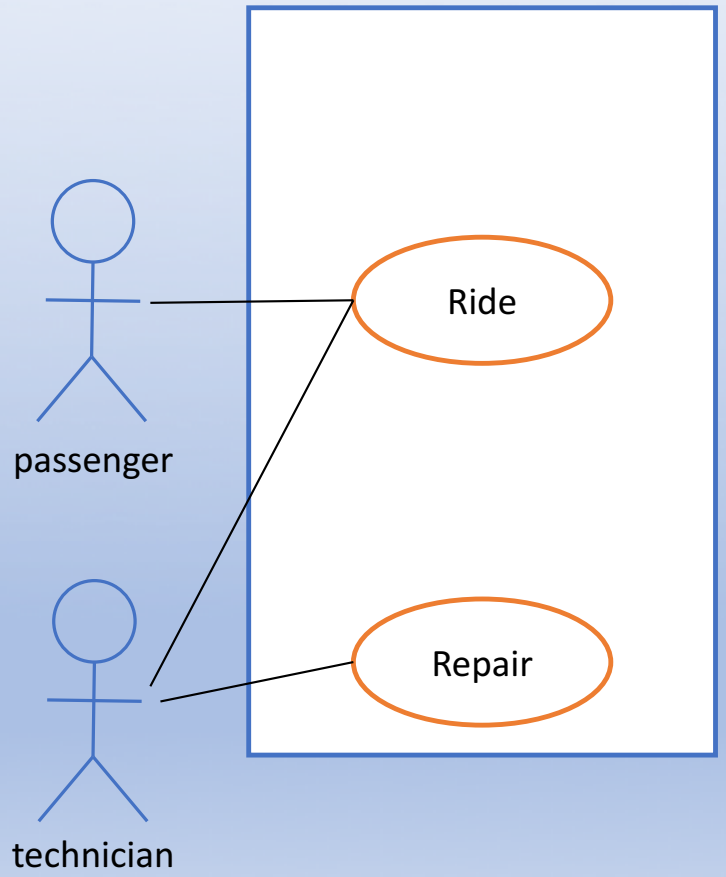
http://ontolog.cim3.net/cgi-bin/wiki.pl?UseCasesSimpleTextExample

1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

- Alternative: Authorization Failure
  - At step 6, system fails to authorize credit purchase
  - Allow customer to re-enter credit card information and re-try
- Alternative: Regular Customer
  - 3a. System displays displays current shipping information, pricing information, and last four digits of credit card information
  - 3b. Customer may accept or override these defaults
  - Return to primary scenario at step 6

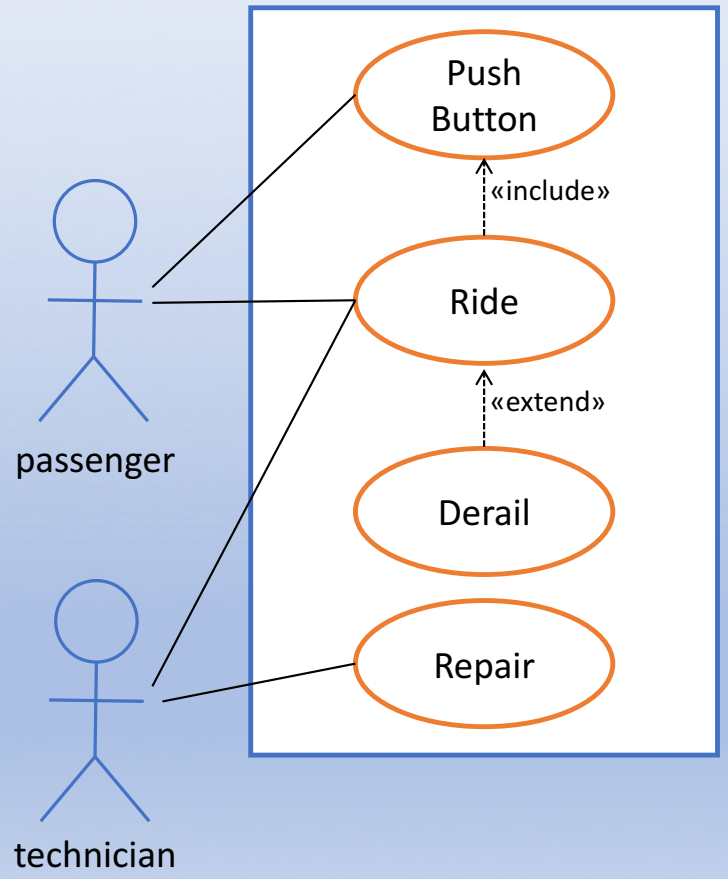
An Example Use Case Diagram

- 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**



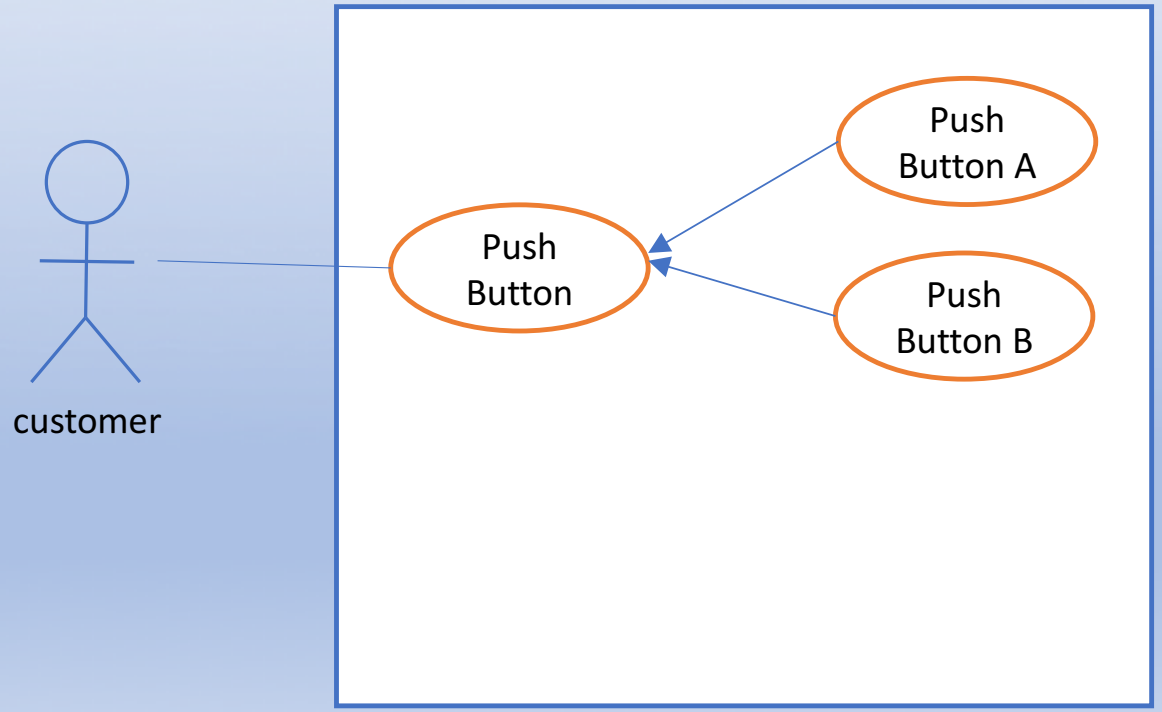
More on Use Case Diagrams

- 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*)



Generalization

- Similar to generalization with classes – Use Case 1 is a type of Use Case 2



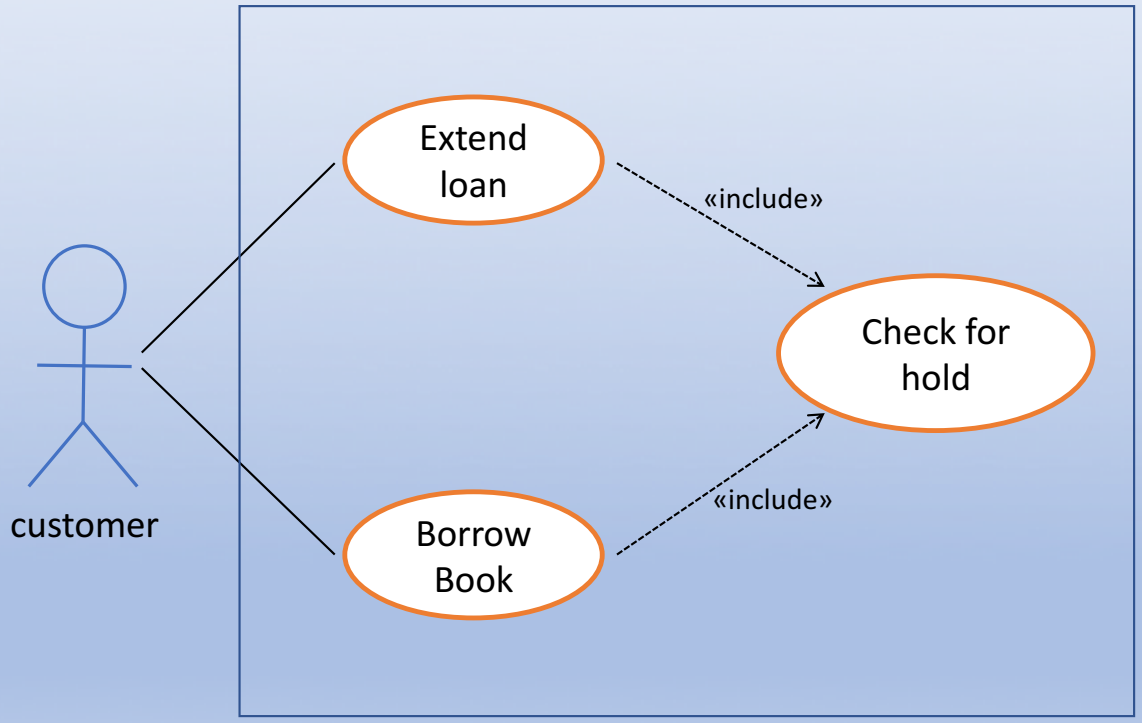
Extensions

- What is an extension?
- **extension:** a possible branch of a use case
  - model optional behavior – adds to "parent" use case's functionality
  - possibly triggered by an error or failure in the process
  - useful for finding edge cases that need to be handled and tested

  - one use case adds behavior to another use case
    - base case is executed
    - depending on what happens during execution, another use case might be executed

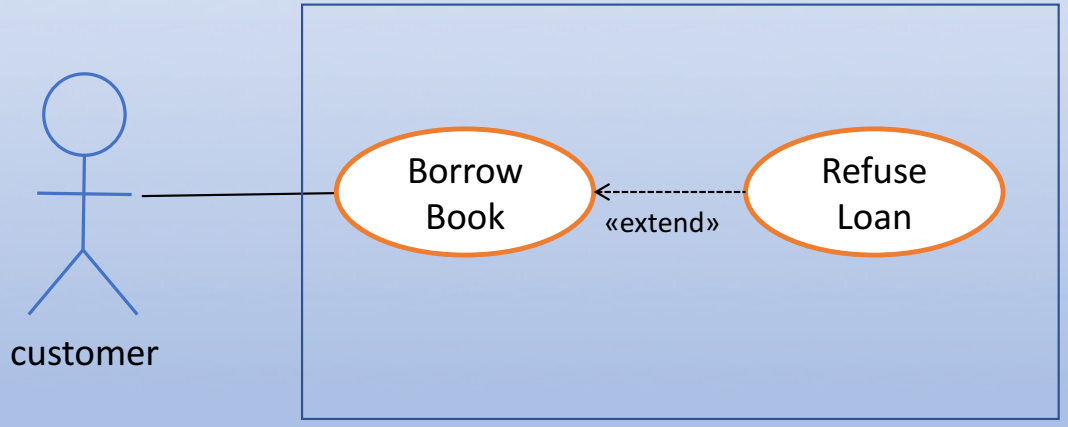
Quickly

- In English, what does this say:



Quickly (again)

- In English, what does this say:



Use Case Exercises

Use Case

Consider an online site that sells a variety of items. Give a brief description of goals that a customer (the actor) may want to initiate with the site.

ACTOR	USE CASES INITIATED
Customer	Place an order Cancel an order Cancel an order item Return an order item

Use Case

- Write a text specification (paragraph) for one of the use cases described on the previous slide.
- <u>Example:</u>
  Place order: The customer places an order for one or more items. The order is entered into the system after the customer pays by credit card. Each item in the order is allocated from existing inventory or from back-ordered inventory if the item is not currently available.

  Cancel Order: Exercise

Exercise: Use Case Diagram

- Draw the use case diagram for the actors and use cases that we identified for the online site order system.
  - System: Represented with a rectangle containing the use cases, and labeled with the system name
  - Actor: Represented with a stick figure
  - Use case: Represented with an oval and labeled with the use case name
  - Association between actor and use case: Represented with a solid line

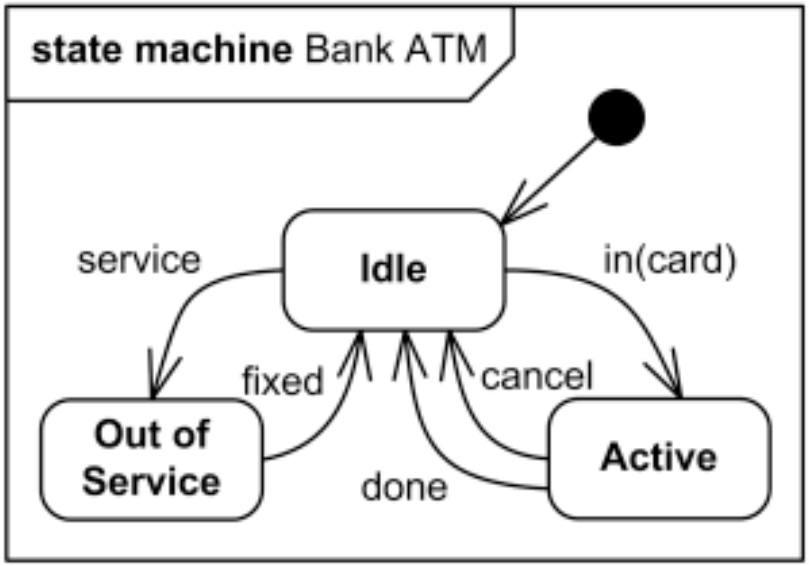
Summary of Use Case Diagrams

- Use case diagram
  - Shows all actors, use cases, relationships
  - Actors are agents that are external to the system (e.g., users)
  - generalization, <<include>>, <<extend>>
- 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

Steps in Creating a Use Case

1. Identify actors and their goals
   - What computers, subsystems, people will drive our system? (Actors)
   - What does each actor need our system to do? (Goals)
   - Exercise: actors/goals for your project
2. Write the success scenario
   - Main success scenario is preferred "happy path"
     - easiest to read and understand – everything else is a complication on this path
   - Capture each actor's intent and responsibility, from trigger to goal delivery
     - Usually each line is numbered
3. List the failure extensions, alternate paths
   - Label with step number and letter

More Modeling with UML



https://www.uml-diagrams.org/state-machine-diagrams.html

State Machine Diagrams

State Machine Diagrams

- Another way of specifying behavioral requirements
  - Built on finite 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
    - how object reacts to sequence of events
  - Protocol state machines show the behavior of a protocol
    - shows how a protocol (e.g., TCP) works
    - show a legal sequence of events
    - no activities specified in states

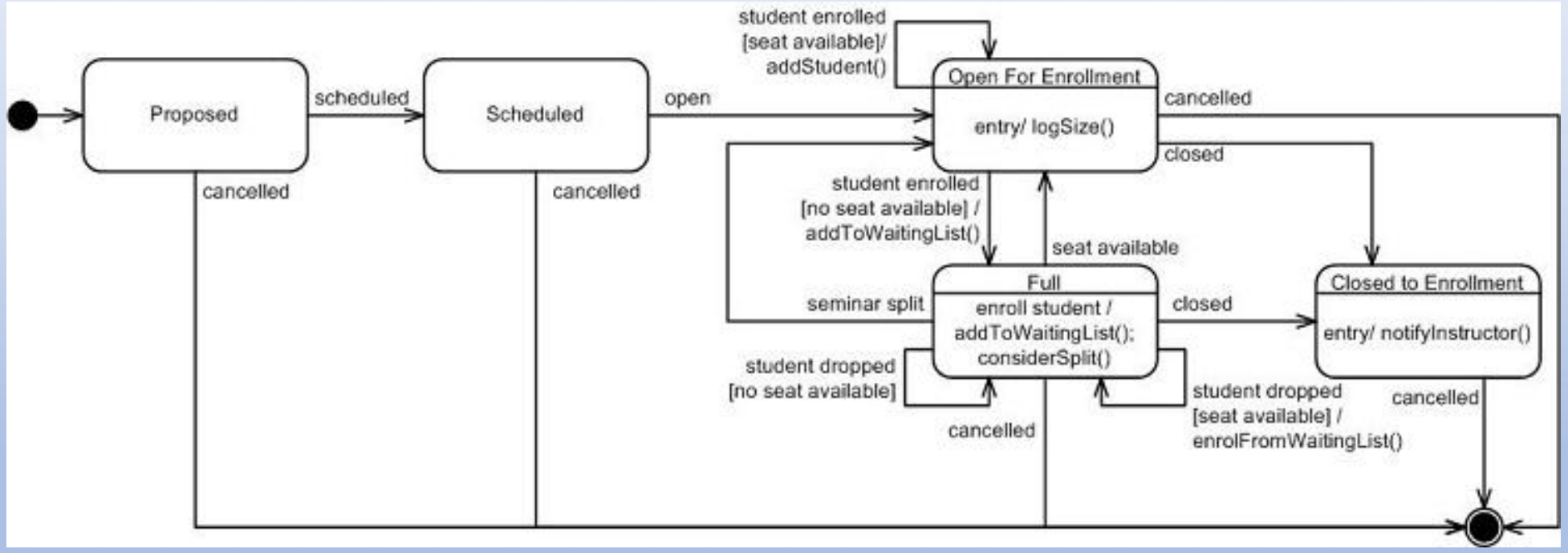
Behavioral State Machine Diagram Components

- 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
    - E.g., *trigger* [*guard*] / *action*
- The **initial state is** represented as a solid black circle

Behavioral State Diagram Concepts & Notation

- initial state (only one): state that a new object is in immediately following its creation
- final state (can be many): a state from which the object doesn't transition – it stops existing
- guarded transition: the guard must be true for the transition to occur
- action: specifies work that is done as object makes a transition
- <u>Transition Label Notation:</u> **event [guard] / action**
- activity: work which is done while an object is in a state
  - <u>Notation:</u> activity listed in state box, usually in second/lower section
    - event / behavior
    - Optionally circumstances/event under which the behavior is invoked is listed, followed by /
    - Reserved event names: entry, do, exit

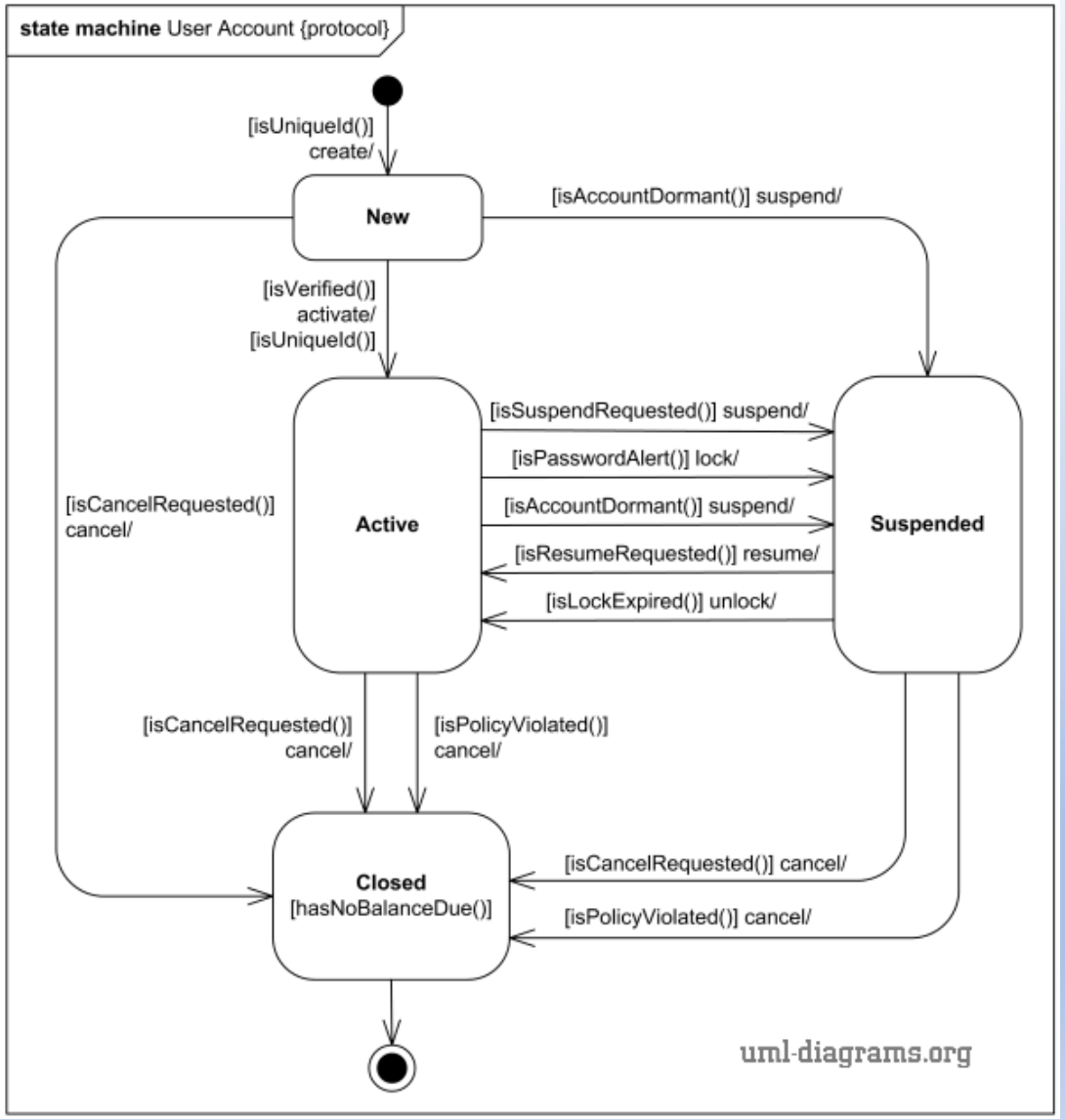
An Example: Seminar class during registration



transition label: event [guard] / action

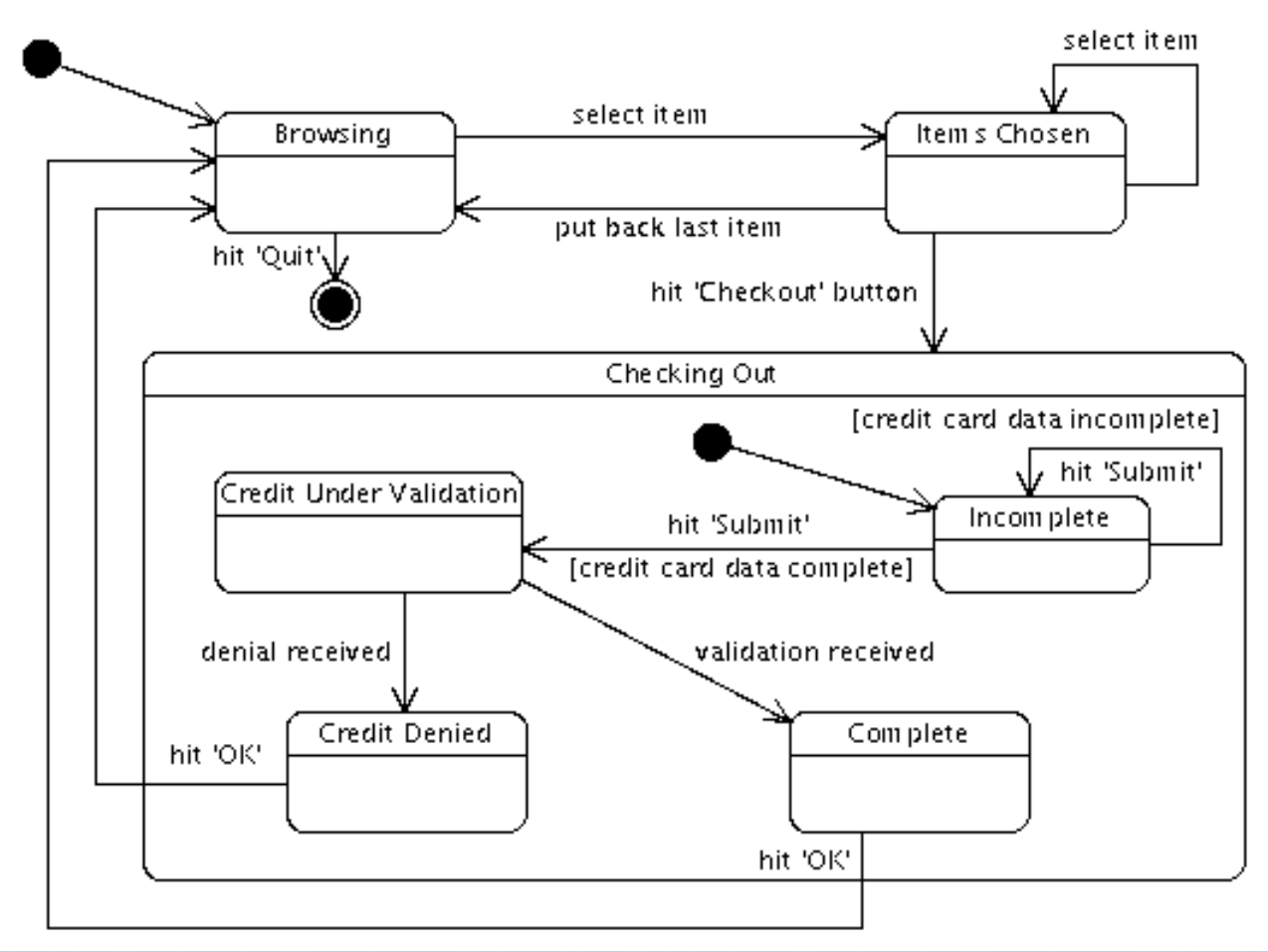
http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm

An Example: Protocol machine



transition label:
[precondition] event / [postcondition]

Another Example



Statechart Exercise

- Draw the statechart diagram that represents the states of a bank account
- 3 states: open, closed, overdrawn
- Indicate initial, final states
- Label transition inputs/events: open, close, deposit(amt), withdraw(amt)...
- Use transition guards, as appropriate: [amt <= balance], ...

Class Diagrams

UML Class Diagram

- Models the **static relationships** between the components of a system
  - Describes the **classes** (in the OO sense)
  - their fields and methods
  - connections between classes
- Classes represent concepts within a system
  - Typically named using **nouns**
- A single class represents one or more objects in the system at runtime

Class Diagram

- Each box is a class
  - Name of class
    - <<interface>> over interface's name
    - italics for abstract class' name
  - Attributes (optional)
    - Visibility, type
  - Operations/methods (optional)
    - don't include inherited 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 Attributes

- attributes (fields, instance variables)
  - visibility name : type [count] = default-value
  - visibility:
    - + public
    - # protected
    - - private
    - ~ package/default
- Example:

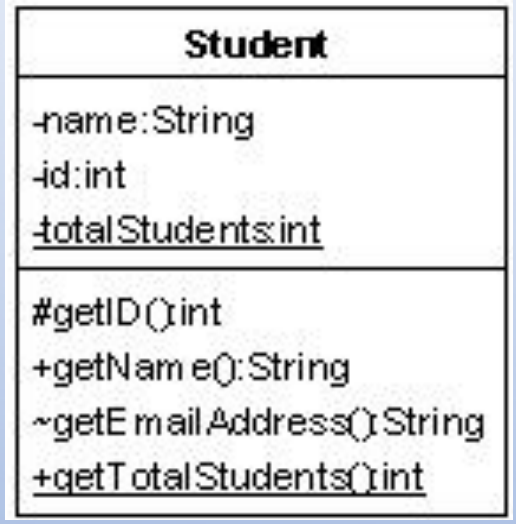
  - balance : double = 0.0

Rectangle
- width : int - height : int
+ Rectangle(width: int, height: int) + distance(r: Rectangle) : double

Class Operations/Methods

- Operations
  - visibility name (parameters) : return-type
  - visibility:
    - + public
    - # protected
    - - private
    - ~ package/default
  - <u>underline static methods</u>
  - parameter types listed as (name : type)
  - omit return type when void
- Example:

  + distance(p1: Point, p2: Point) : double



Relationships

- generalization:
  - inheritance
    - solid arrow with closed arrowhead
  - interface implementation
    - dotted arrow with closed arrowhead

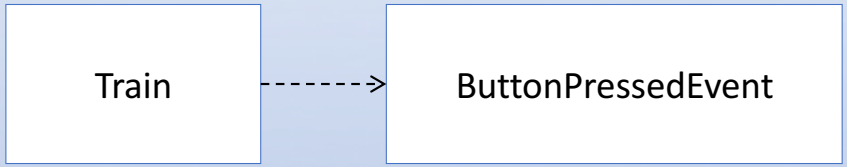
- associational relationships
  - dependency: "using temporarily" relationship
    - weak relationship – e.g., class A is type of member function argument for class B
    - dotted, possibly directional arrow
  - association: "uses" relationship
    - ability of an instance to send a message to an instance of the other class
    - solid line between the classes
    - bidirectional unless arrow present
  - aggregation: "has a" relationship, whole-part relationship
    - connected by solid line, with unfilled diamond at whole/aggregate class
  - composition: stronger form of aggregation – the "part" can't exist without the "whole"
    - same as aggregation, except the diamond is filled

Associational Relationships

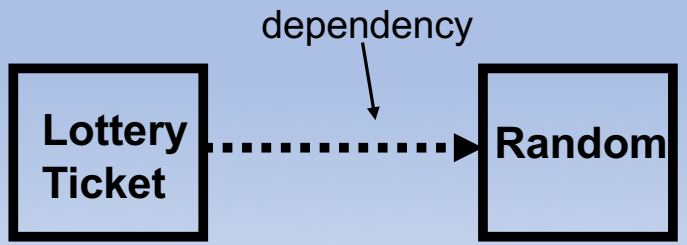
- Multiplicity (how many are used)
  - * 0 or more
  - 1 1 exactly
  - 2..4 2, 3 or 4
  - 2..* 2 or more
- Can name the relationship

Dependency

- Dependency is the weakest relationship
  - E.g., class A **uses** class B
  - Used if A:
    - has a method with parameter of type B
    - has a temporary variable of type B
  - Depicted by a dotted arrow

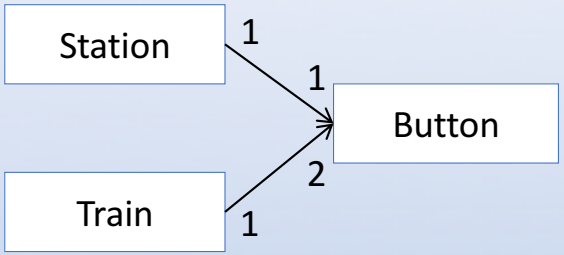


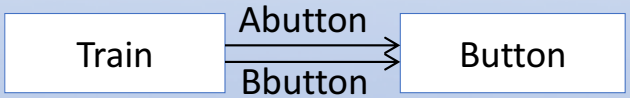

```
public class Train {
    ...
    protected void addStop(ButtonPressedEvent stop){
        // update nextStop
        ...
    }
    ...
}
```

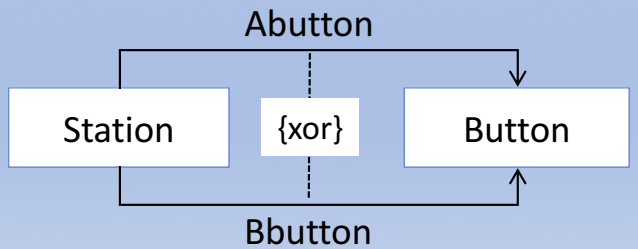


Association

- Indicates a stronger relationship
  - Instances can send message to instances of other class
  - Often one-way
  - Object of class A has to know about object of type B
  - Specific types of association: aggregation, composition
- Use number labels to indicate multiplicity
  - Use * to indicate arbitrary cardinality
- You can also explicitly name the associations
- And make them conditional

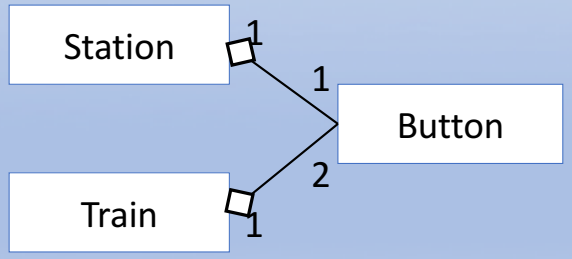


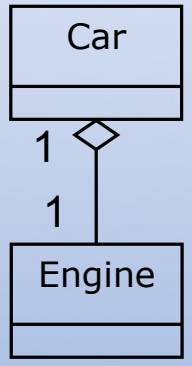




Aggregation

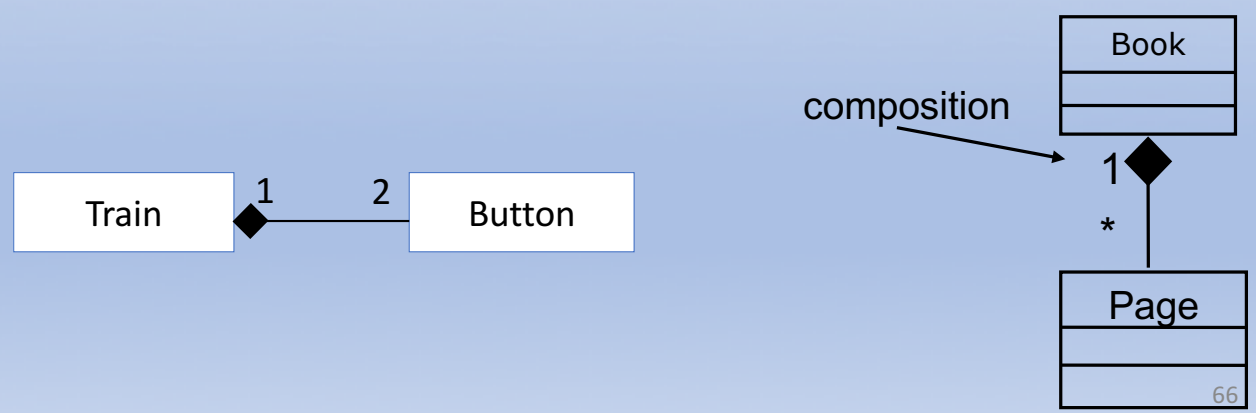
- Indicates a strong association: "is part of" relationship
  - E.g., Class A **owns a** Class B
  - A contains a reference to object of type B
  - Imagine the buttons were “members” of the train/station classes (just a different design, really)





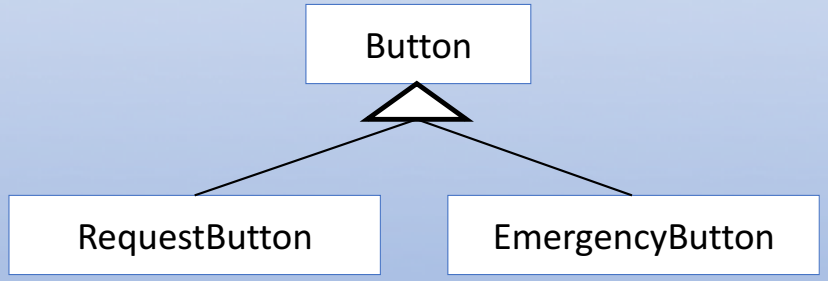
Composition

- 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++, <u>aggregation is usually defined by pointers/references</u>, while <u>composition is defined by containing instances</u>
    - So in composition: when an A object is destroyed, the B object it contains will also be destroyed
  - In Java, composition is often indicative of the inner class style relationship



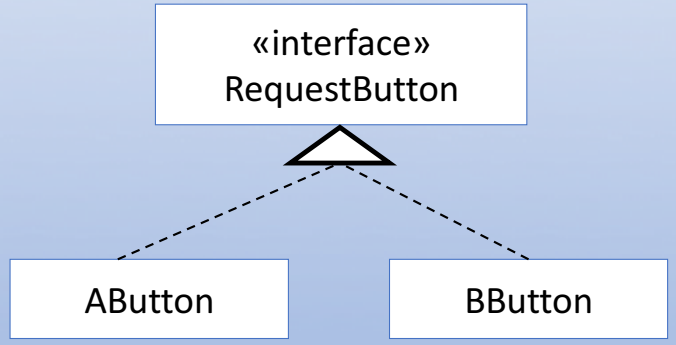
Generalization

- 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



Realization

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



Operations in Class Diagrams

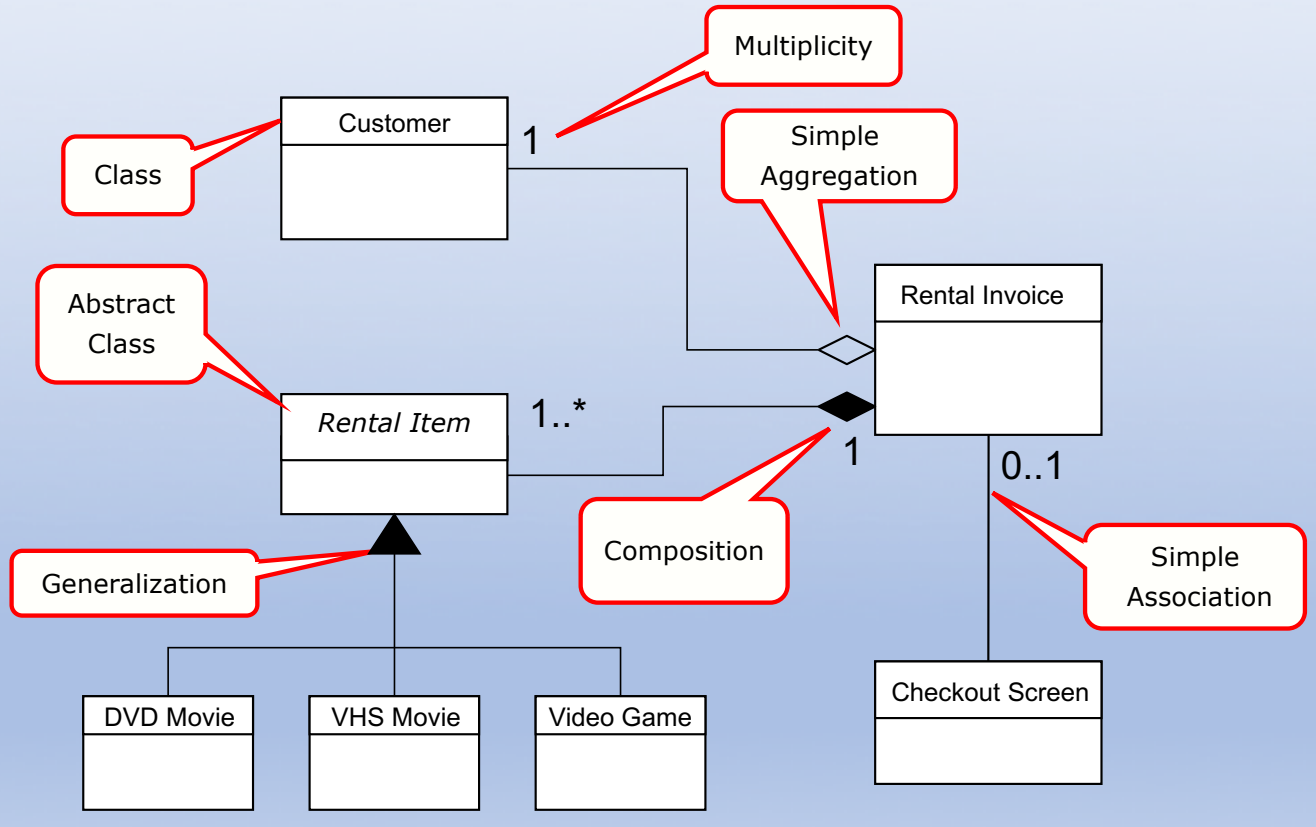
- Operation descriptions include
  - Visibility
    - Public +
    - Private -
    - Protected #
    - Package ~
  - Parameter list
    - Name
    - Type
    - Multiplicity
  - Polymorphism
  - Abstract operations (italics)

```
Train

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

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

Class Diagram Example



Sequence Diagrams

“The key question to keep asking is, Are you spending your time on the right things? Because time is all you have. ”
— Randy Pausch, The Last Lecture

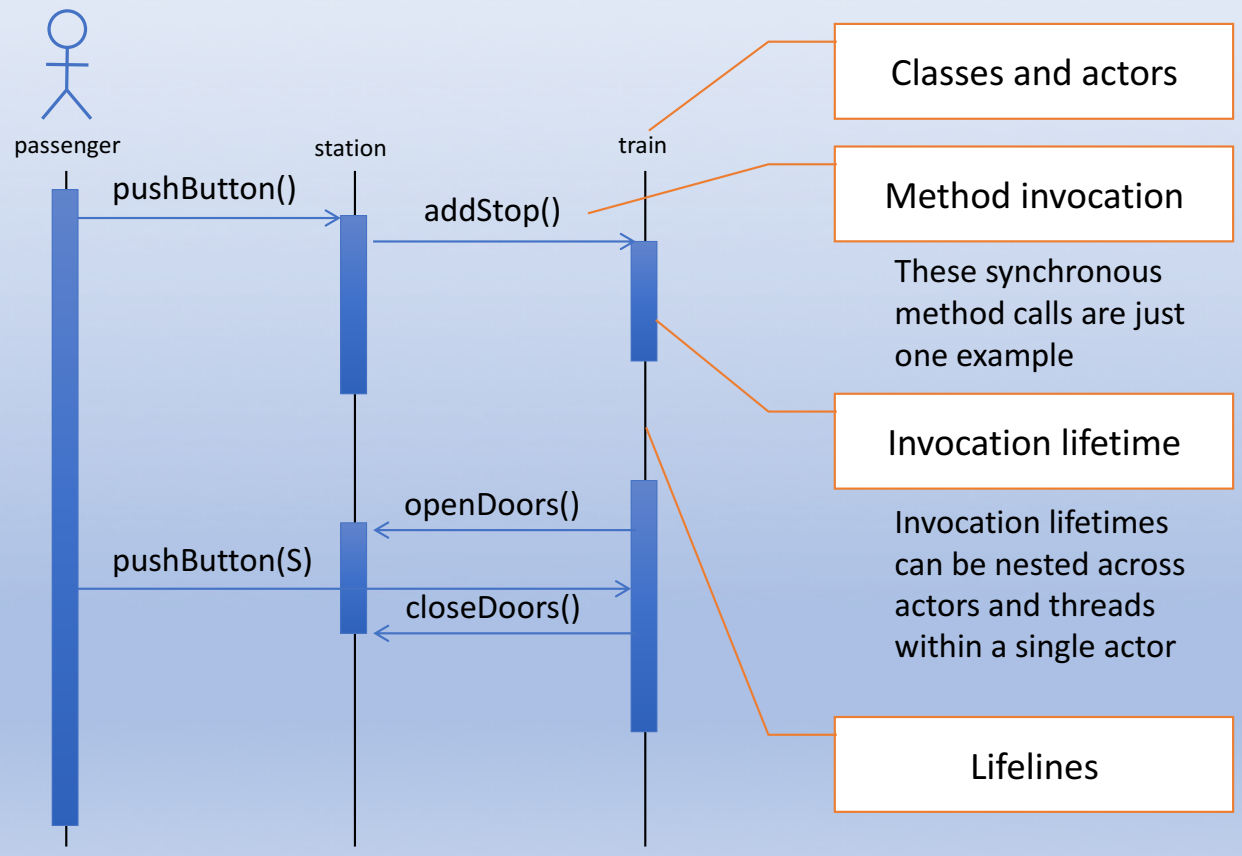
Interaction Diagrams

- Focus on **communication** between elements
  - **Sequence diagrams: emphasize time ordering**
    - **flow of messages, events between objects**
  - Communication diagrams
  - Interaction overview diagrams
  - Timing diagrams

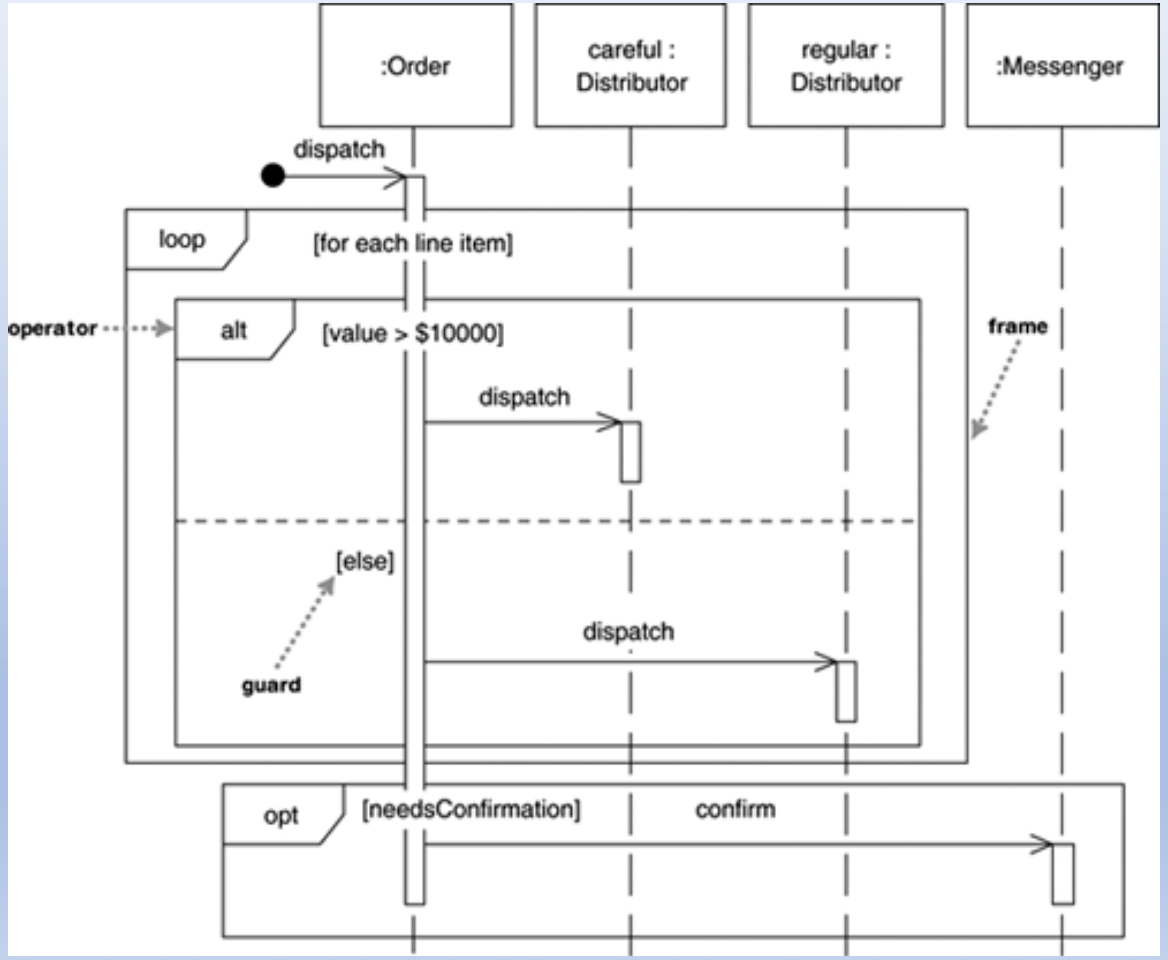
UML Sequence Diagrams

- Sequence diagrams show a time-based view of messages between objects
- Think of it as a **table**:
  - Columns are classes, objects 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

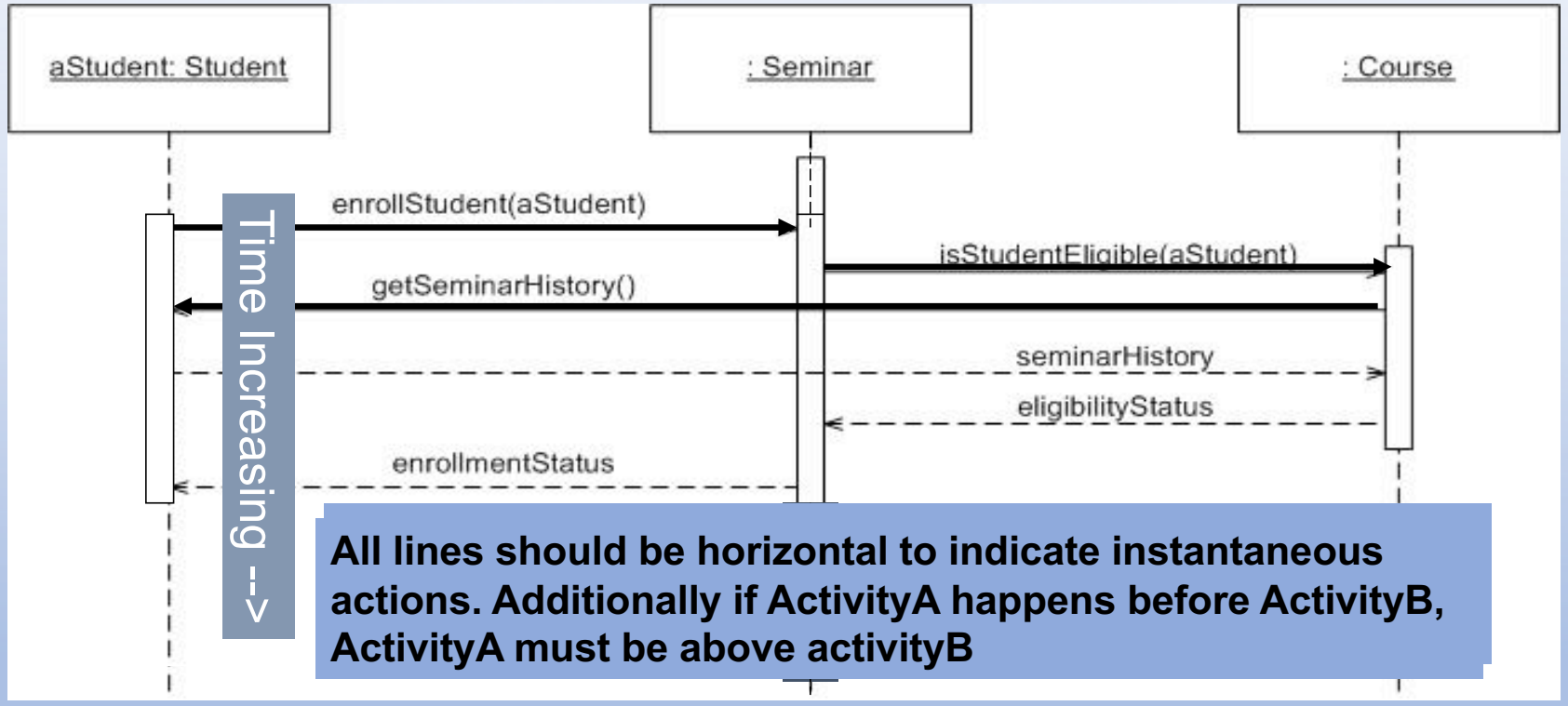
Example Sequence Diagram



Loops and Alternatives

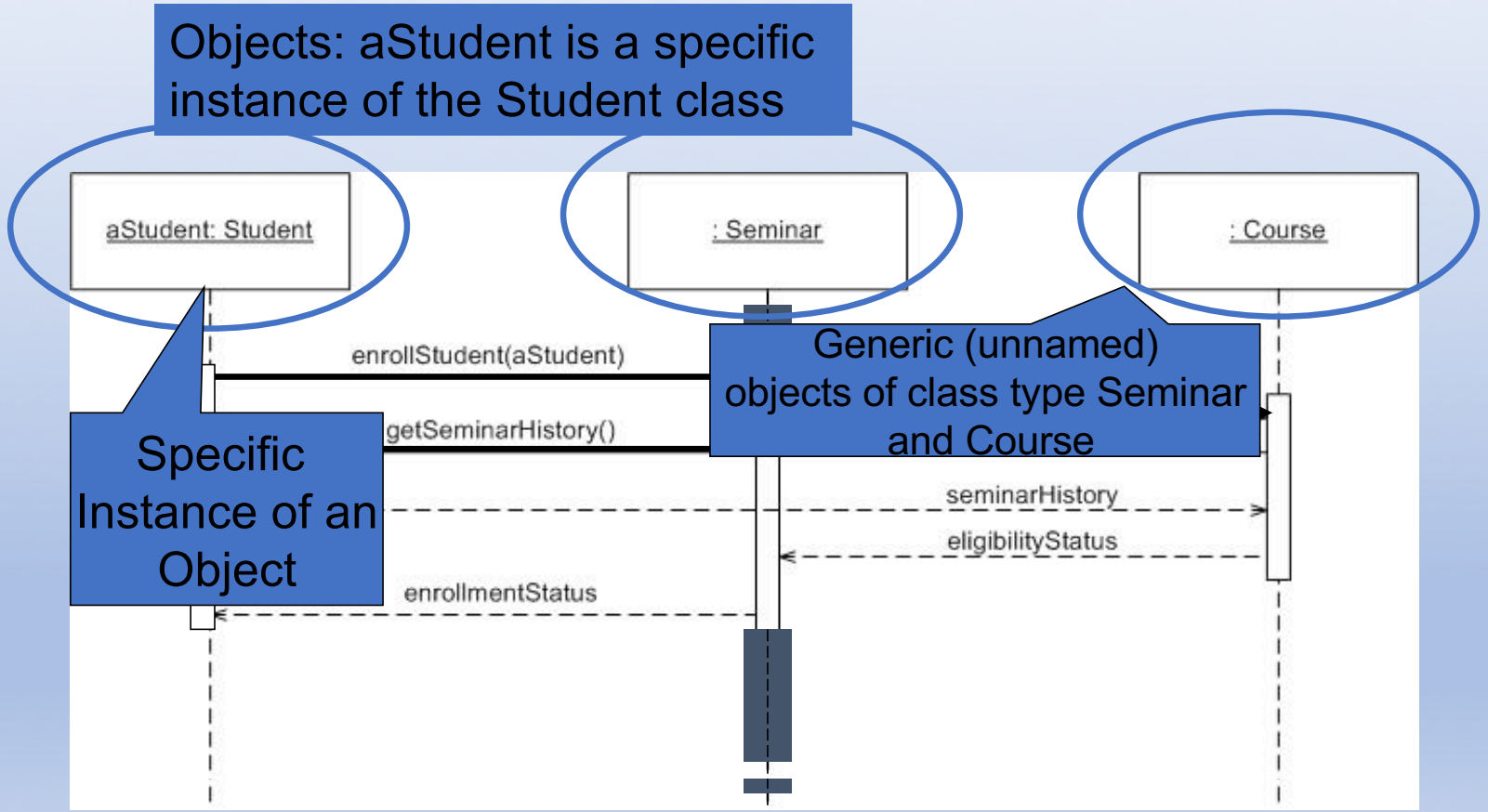


Sequence Diagram



All lines should be horizontal to indicate instantaneous actions. Additionally if ActivityA happens before ActivityB, ActivityA must be above activityB

Components

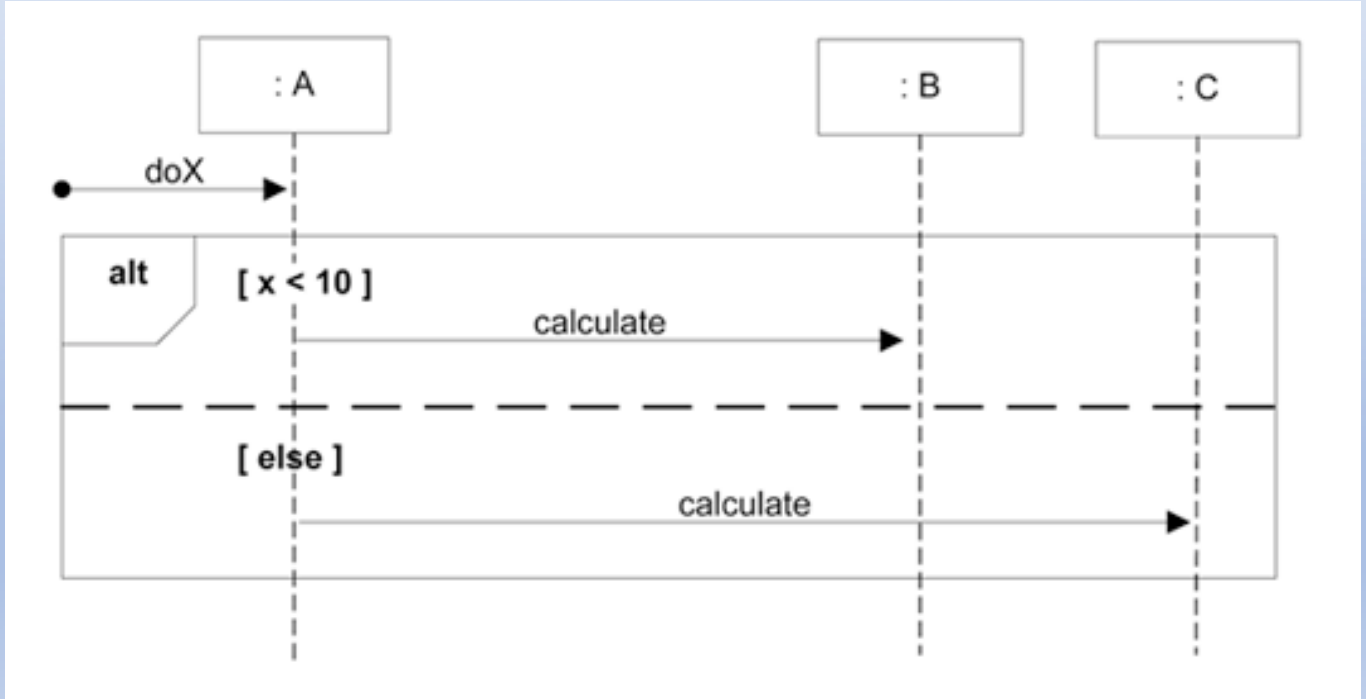

Objects: aStudent is a specific instance of the Student class
aStudent: Student
: Seminar
: Course
enrollStudent(aStudent)
getSeminarHistory()
seminarHistory
eligibilityStatus
enrollmentStatus
Specific Instance of an Object
Generic (unnamed) objects of class type Seminar and Course

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

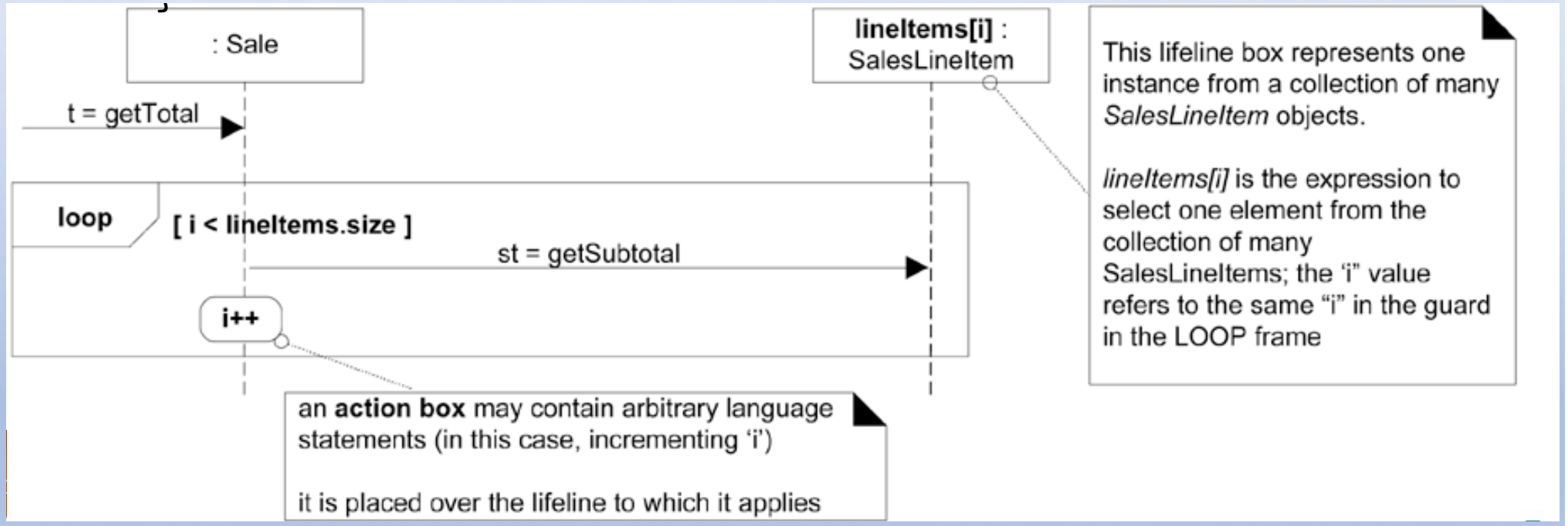
- Write pseudocode that has the same meaning as the following sequence diagram



Answer

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

Another One…


: Sale
lineItems[i] :
SalesLineItem
t = getTotal
loop
[i < lineItems.size]
st = getSubtotal
i++
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 subtotal = null;

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

Sequence Diagram Exercise

- A customer is at a print shop. The customer requests that a file be printed at a printer station. The station determines the status of the printer; if the printer is ready, it prints the file. If not, the file is added to the print queue.

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 readings and recommended books, and not too difficult to understand