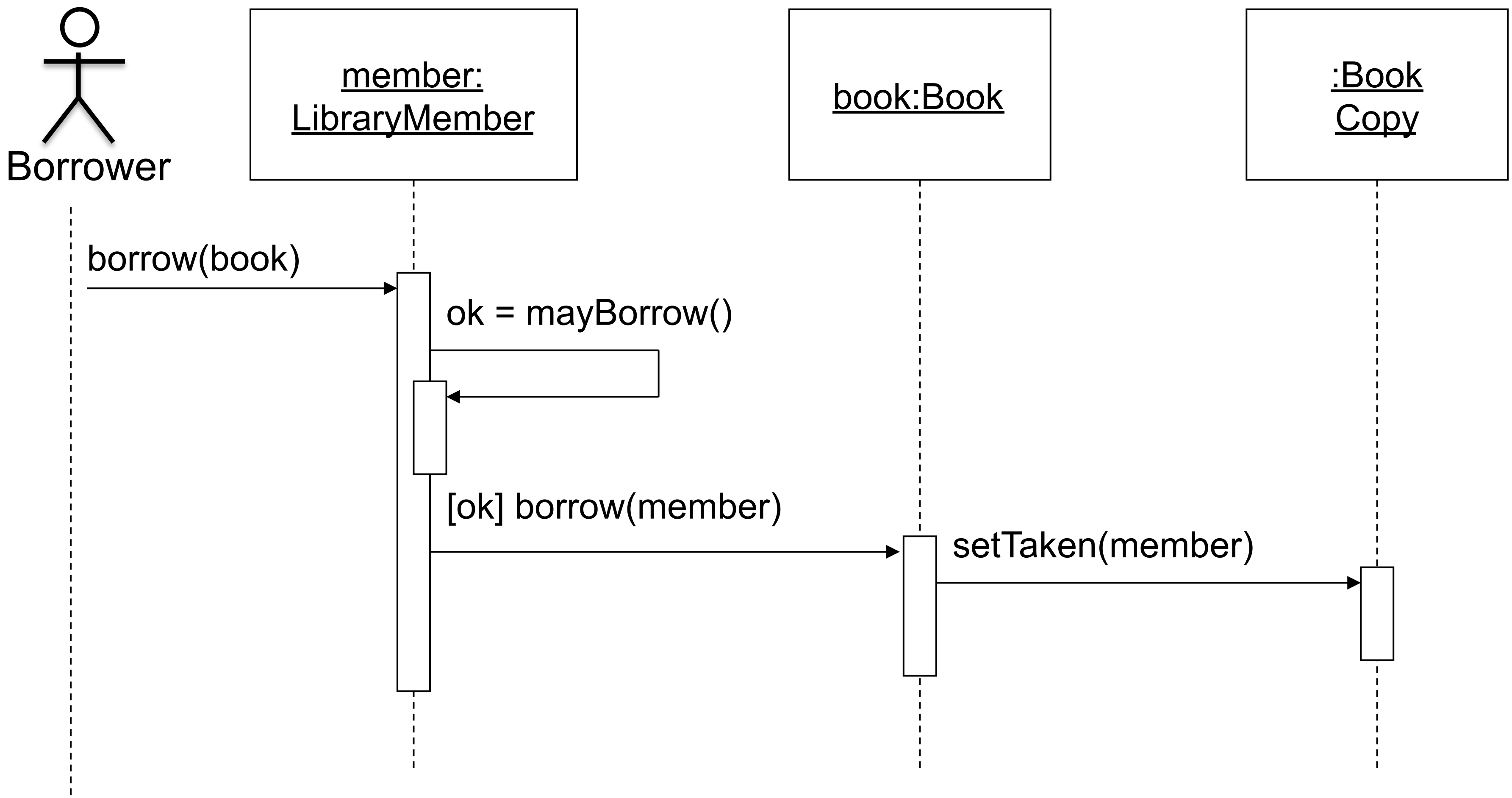
Sequence Diagrams

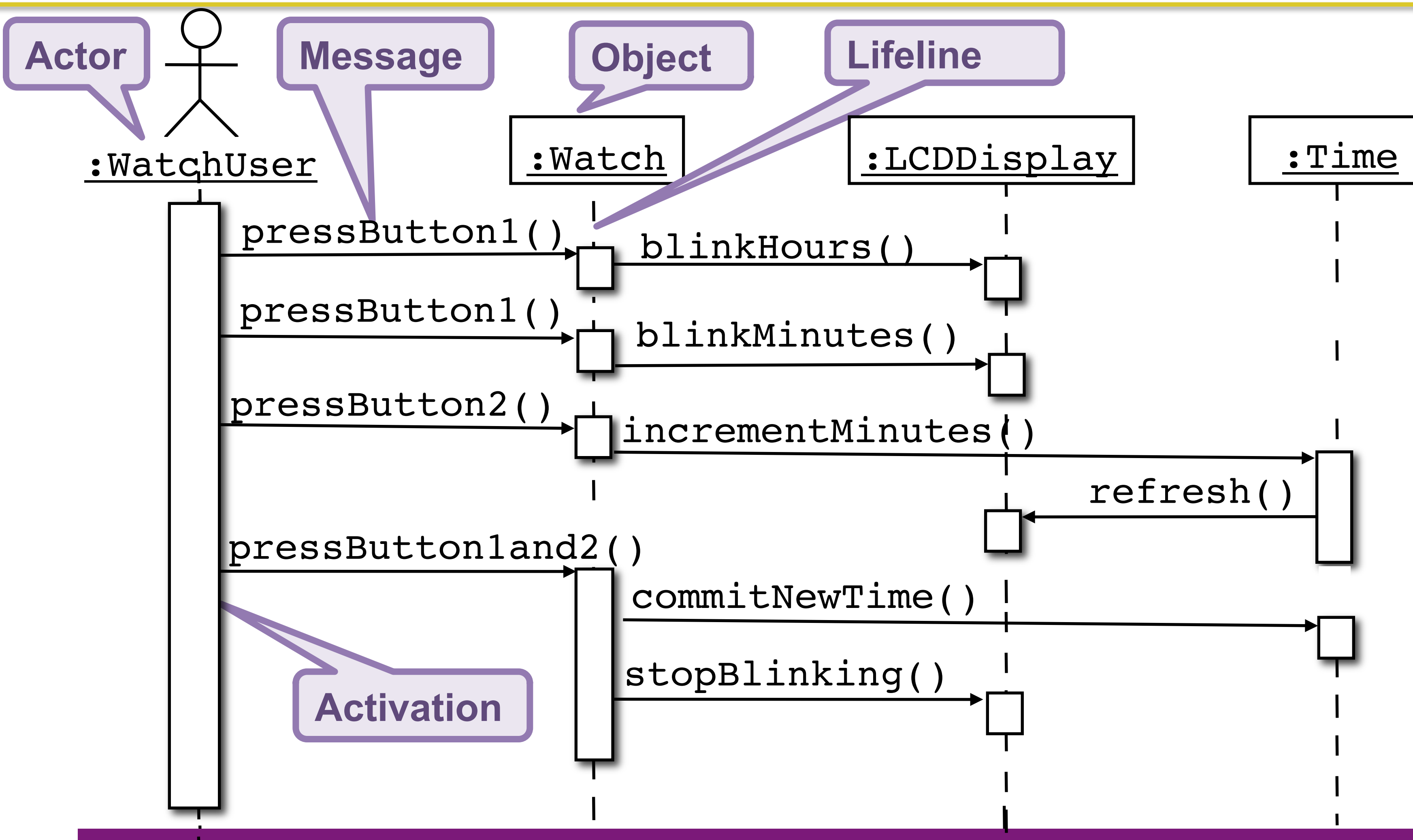
CSCI-4448 - Boese

University of Colorado Boulder

Overview

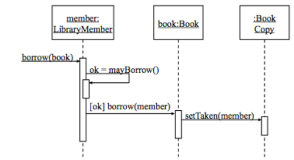

Borrower
member:
LibraryMember
book:Book
:Book
Copy
borrow(book)
ok = mayBorrow()
[ok] borrow(member)
setTaken(member)

Sequence diagram



Sequence diagrams represent the behavior of a system as messages ("**interactions**") between *different objects*

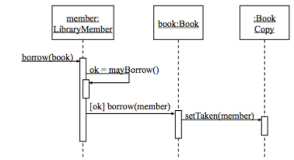
Sequence Diagrams - Definition



Sequence Diagrams

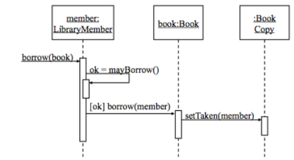
- Models the dynamic behavior and system interactions for a <u>single use case scenario</u> executing in the system
- Models the assignment of responsibility
- Interaction diagram

Activity Diagrams - Why



- Why?
  - Assist in understanding how a system (a use case) actually works
  - Verify that a use case description can be supported by the existing classes
  - Identify responsibilities/operations and assign them to classes
  - The structure of the sequence diagram helps us to determine how decentralized the system is

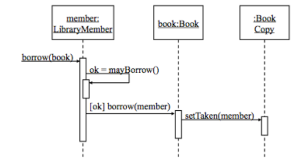
Why not just code it?



- Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?
  - A good sequence diagram is still a bit above the level of the real code (not EVERY line of code is drawn on diagram)
  - SDs are language-agnostic
  - Non-coders can do sequence diagrams
  - Easier to do sequence diagrams as a team
  - Can see many objects/classes at a time on same page (visual bandwidth)

Details

Sequence Diagrams



Sequence Diagrams

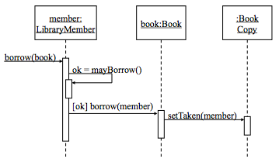
- Model the behavior and interactions of use cases scenarios.
- Reference your use cases and class diagrams
- Heuristic for finding participating objects:
  - A event always has a sender and a receiver
  - Find them for each event => These are the objects participating in the use case.

Heuristics for Sequence Diagrams

- Layout:
  1st column: Should be the actor of the use case
  2nd column: Should be a boundary object
  3rd column: Should be the control object that manages the rest of the use case

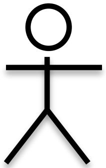
- Creation of objects:
  - Create control objects at beginning of event flow
  - The control objects create the boundary objects
- Access of objects:
  - Entity objects can be accessed by control and boundary objects
  - Entity objects should not access boundary or control objects.

Sequence Diagrams



Sequence Diagrams

- Main Components



  - **Actor:** stick figure
  - **Objects/Classes**: *Rectangle*

book:Book

  - **Life-Lines**: *Dotted vertical lines*
  - **Messages**: *Horizontal arrows*
  - **Conditions**: *In brackets*

[ok] borrow(member)

  - **Iterations**: *
  - **Activation Boxes**: *Vertical rectangles*
  - **Return Values**: *Dotted lines*
  - **Time**: *[runs top to bottom]*

An Example

- Flow of events in “Get SeatPosition” use case :

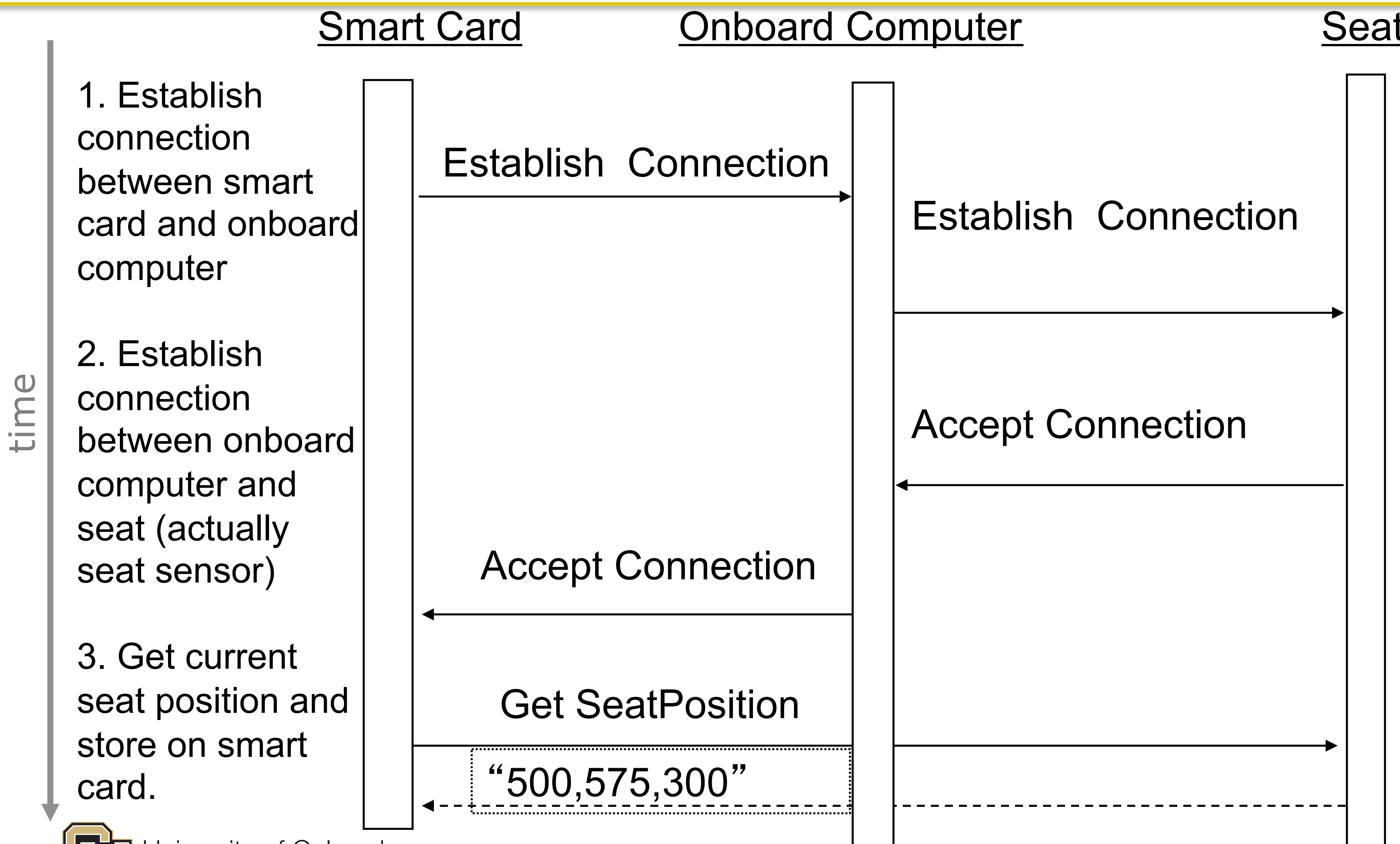
    1. Establish connection between smart card and onboard computer

    2. Establish connection between onboard computer and sensor for seat

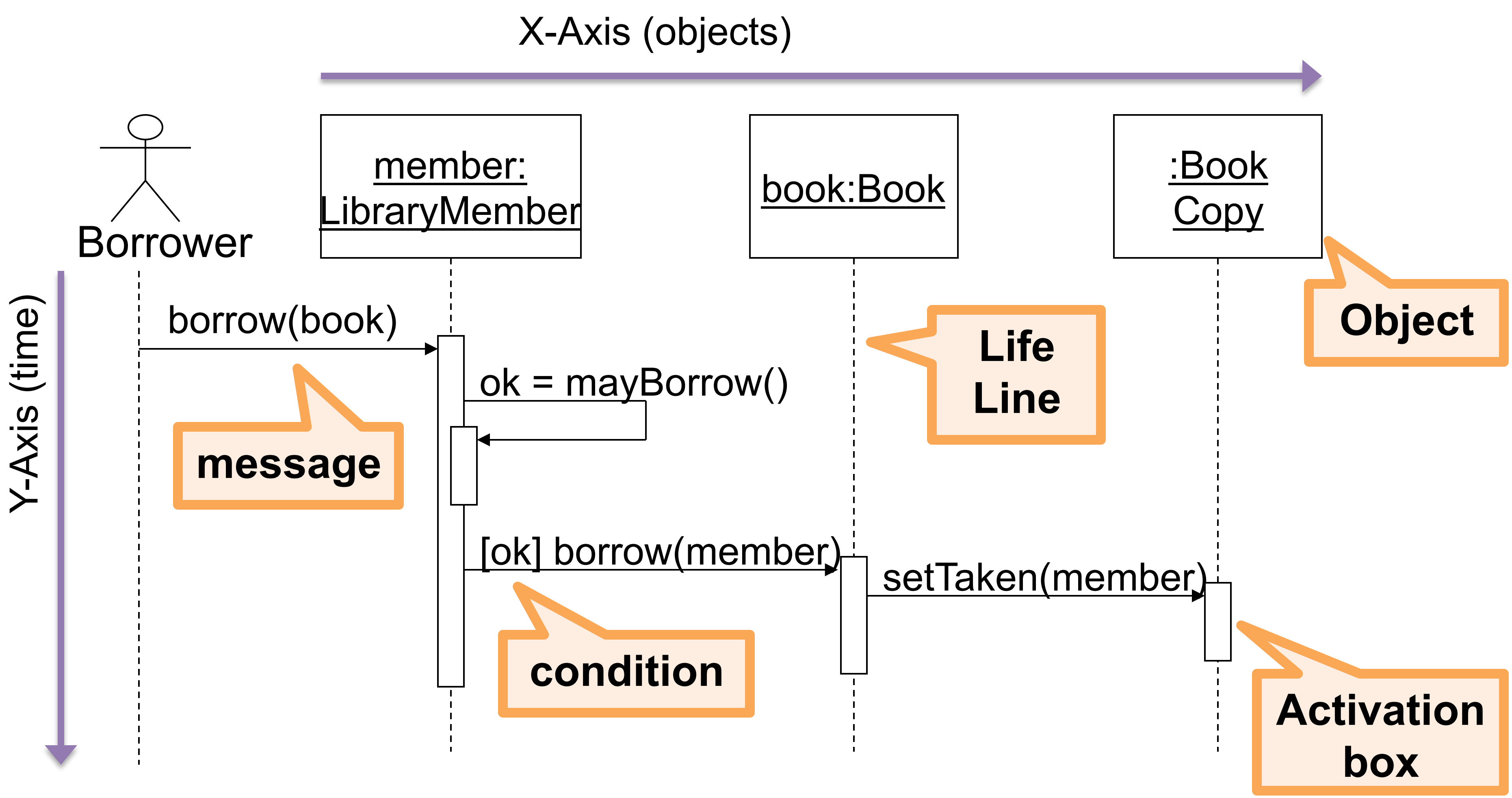
    3. Get current seat position and store on smart card

- Where are the objects?

Sequence Diagram for “Get SeatPosition”



A Sequence Diagram



Object

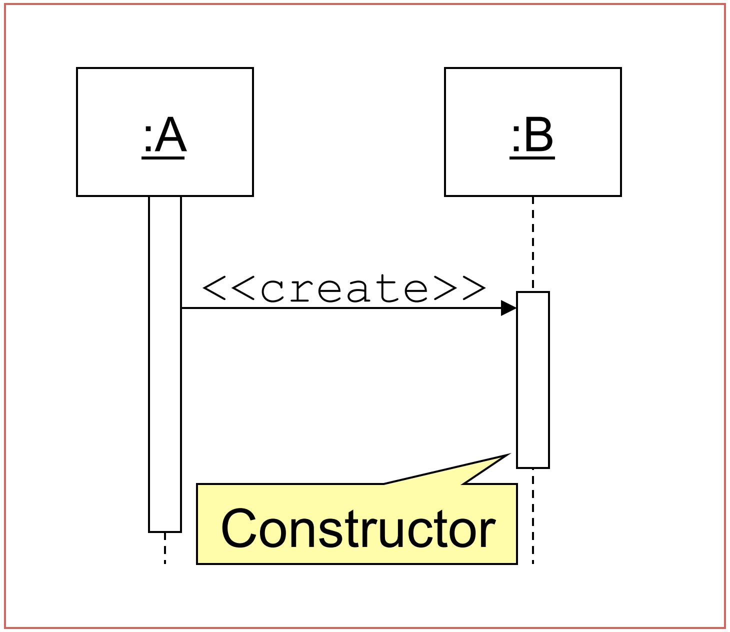
Objects

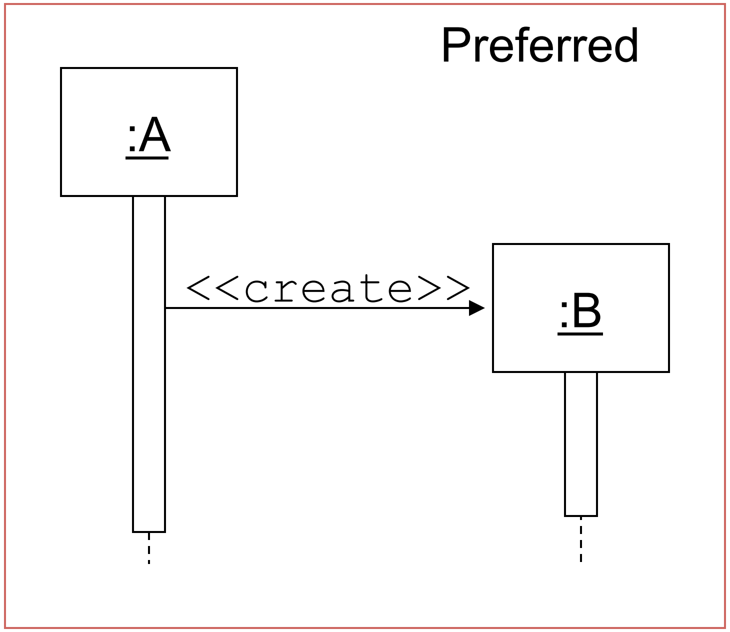
myBirthday
:Date

- Naming:
  - syntax: [instanceName]:className
  - Name classes consistently with your class diagram (same classes).
  - Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.
- "Black Box": interactions with the objects and messages sent input/output

Object Creation

- An object may create another object via a <<create>> message.

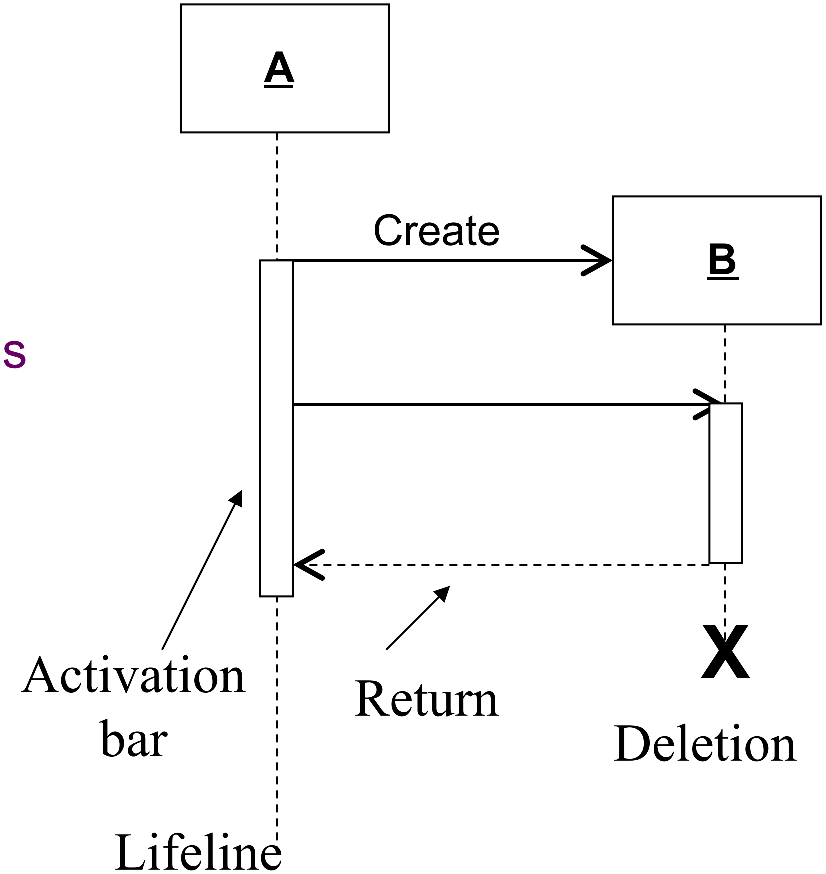




Sequence Diagrams – Object Life Spans

Object Life

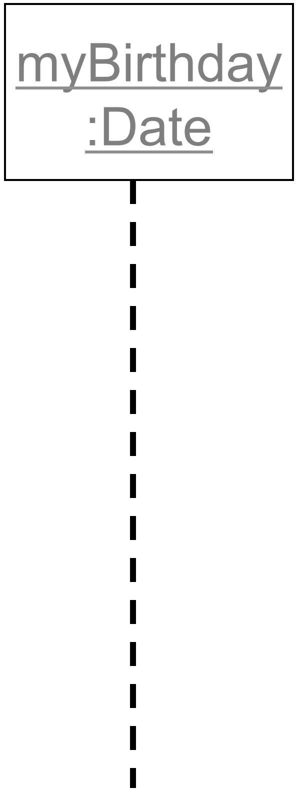
- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular box
  - Place on the lifeline where object is activated.
  - The time required by the receiver object to process the message.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an ‘X’ on lifeline
  - Object’s life ends at that point



Object

Life-Line

- Dotted-Line
- Represents the object’s life during the interaction



Messages

borrow(book)

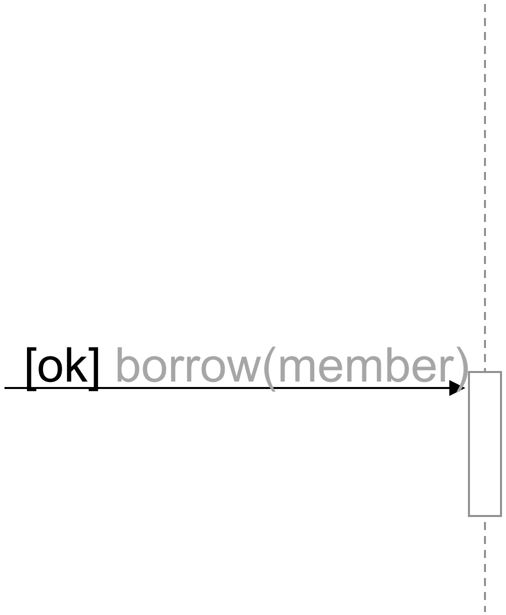
Messages

- Arrow between the life lines of two objects.
  - Self calls are also allowed
- Labeled at least with the message name.
  - Arguments and control information
    - *Conditions*
    - *Iteration*
- An interaction between two objects is performed as a <u>message</u> sent from one object to another

Messages - Condition

Message Condition

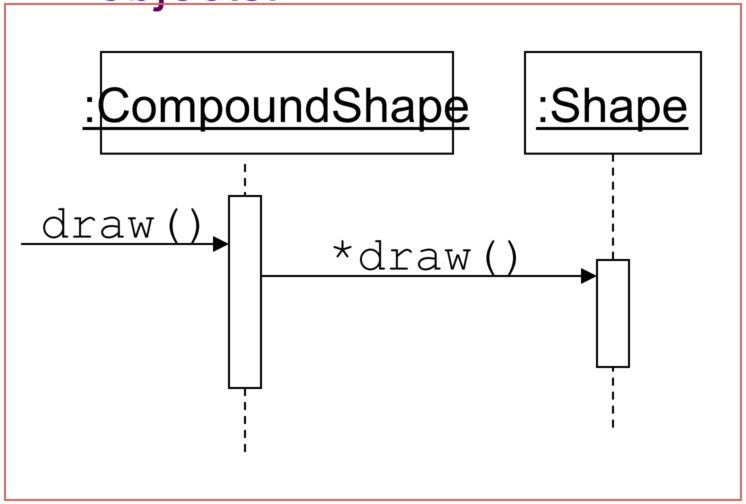
- In brackets before message detail
- The message is sent only if the condition is true

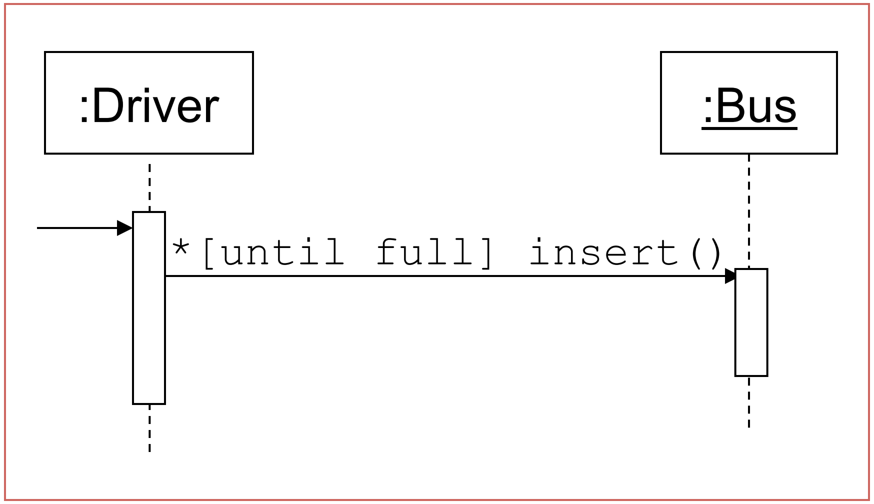


Messages - Iteration

Message Iteration

- Designate:
  - * message-label
  - *[expression] message-label
- The message is sent many times to possibly multiple receiver objects.

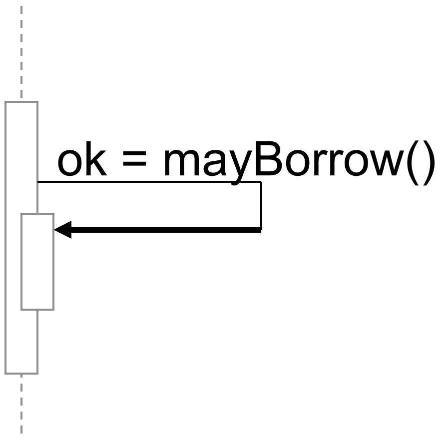




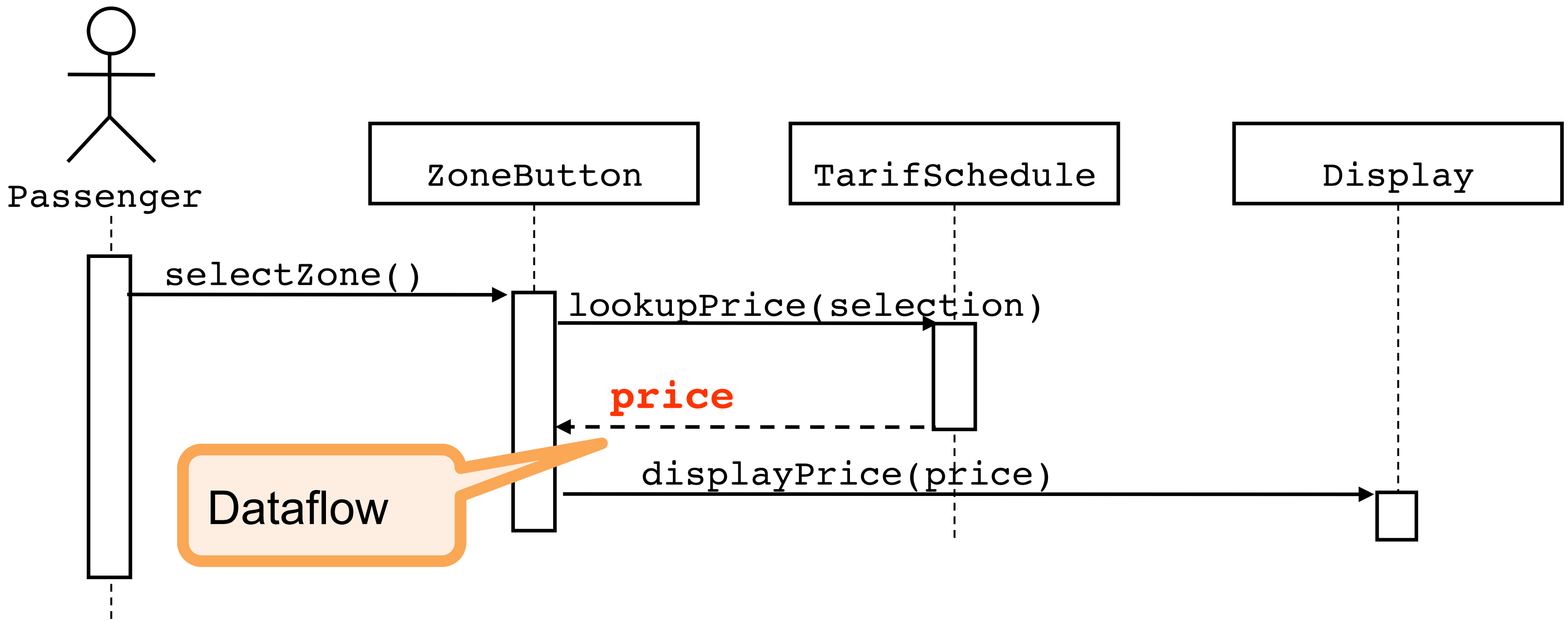
Return Values

Return Values

- Dashed arrow with a label indicating the return value.
  - Don't model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
  - Prefer modeling return values as part of a method invocation, e.g. `ok = isValid()`



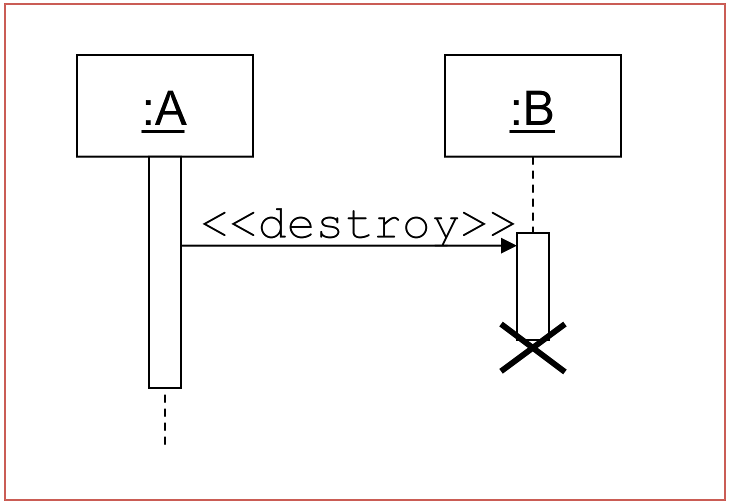
Model the Flow of Data



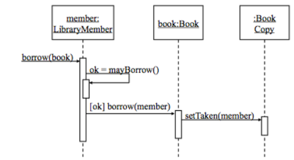
- The source of an arrow indicates the activation which sent the message
- Horizontal dashed arrows indicate data flow, for example return results from a message

Object Destruction

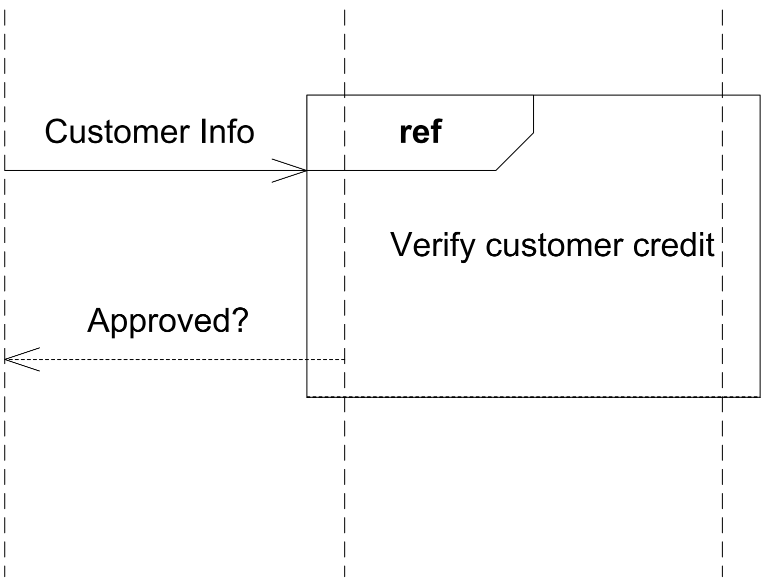
- An object may destroy another object via a `<<destroy>>` message.
  - An object may destroy itself.
  - *Avoid modeling object destruction unless memory management is critical.*

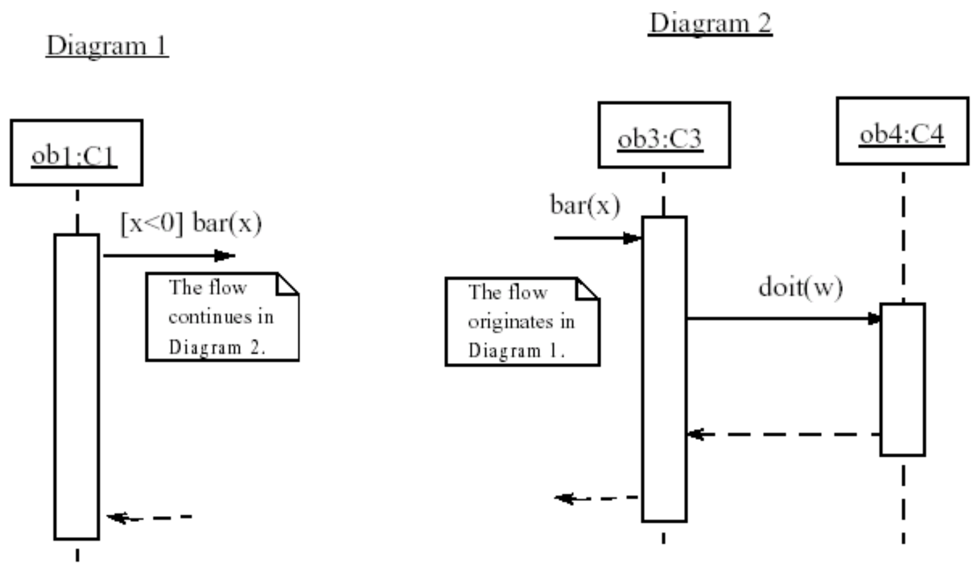


linking sequence diagrams



- if one sequence diagram is too large or refers to another diagram, indicate it with either:
  - an unfinished arrow and comment
  - a "ref" frame that names the other diagram



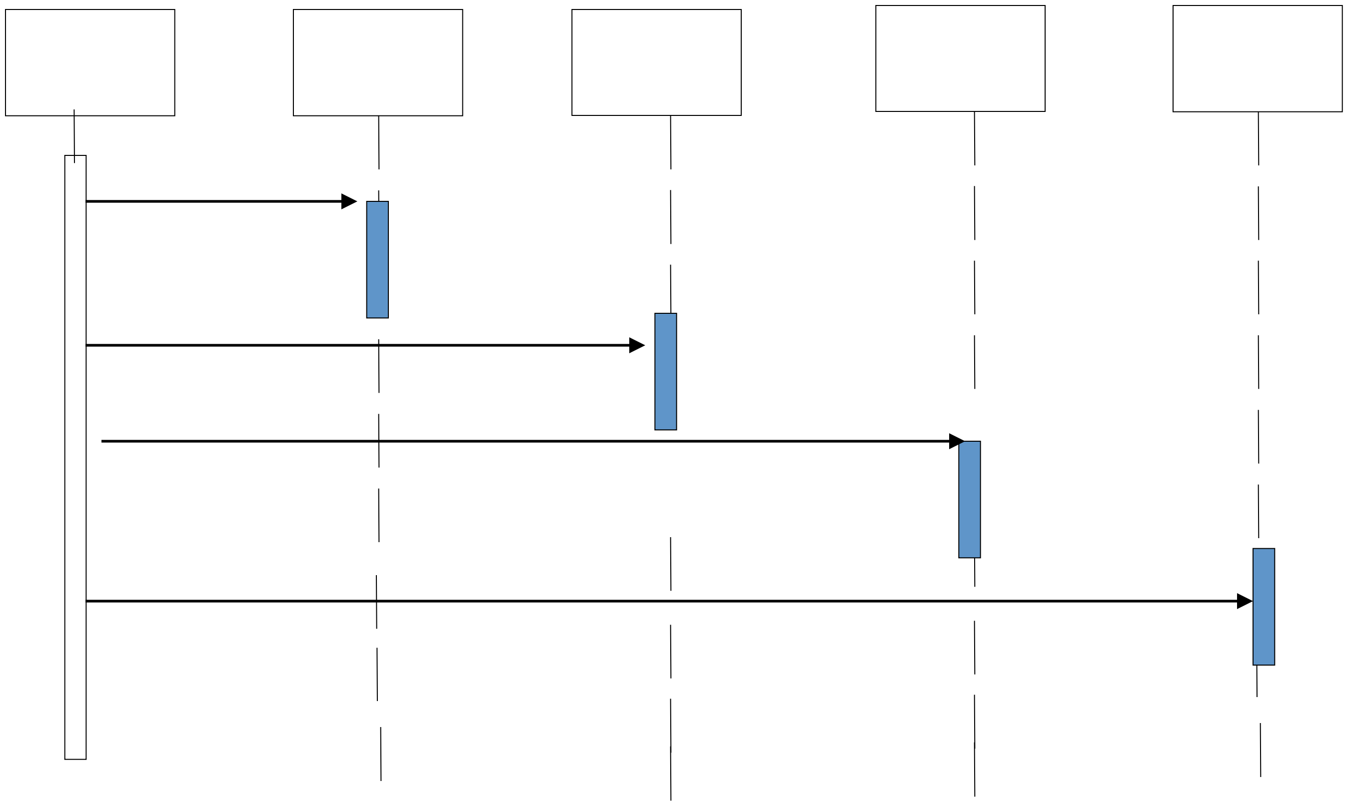


What else can we get out of Sequence Diagrams?

- Sequence diagrams are derived from use cases
- The structure of the sequence diagram helps us to determine how **decentralized** the system is
- We distinguish two structures for sequence diagrams
  - Fork Diagrams
  - Stair Diagrams

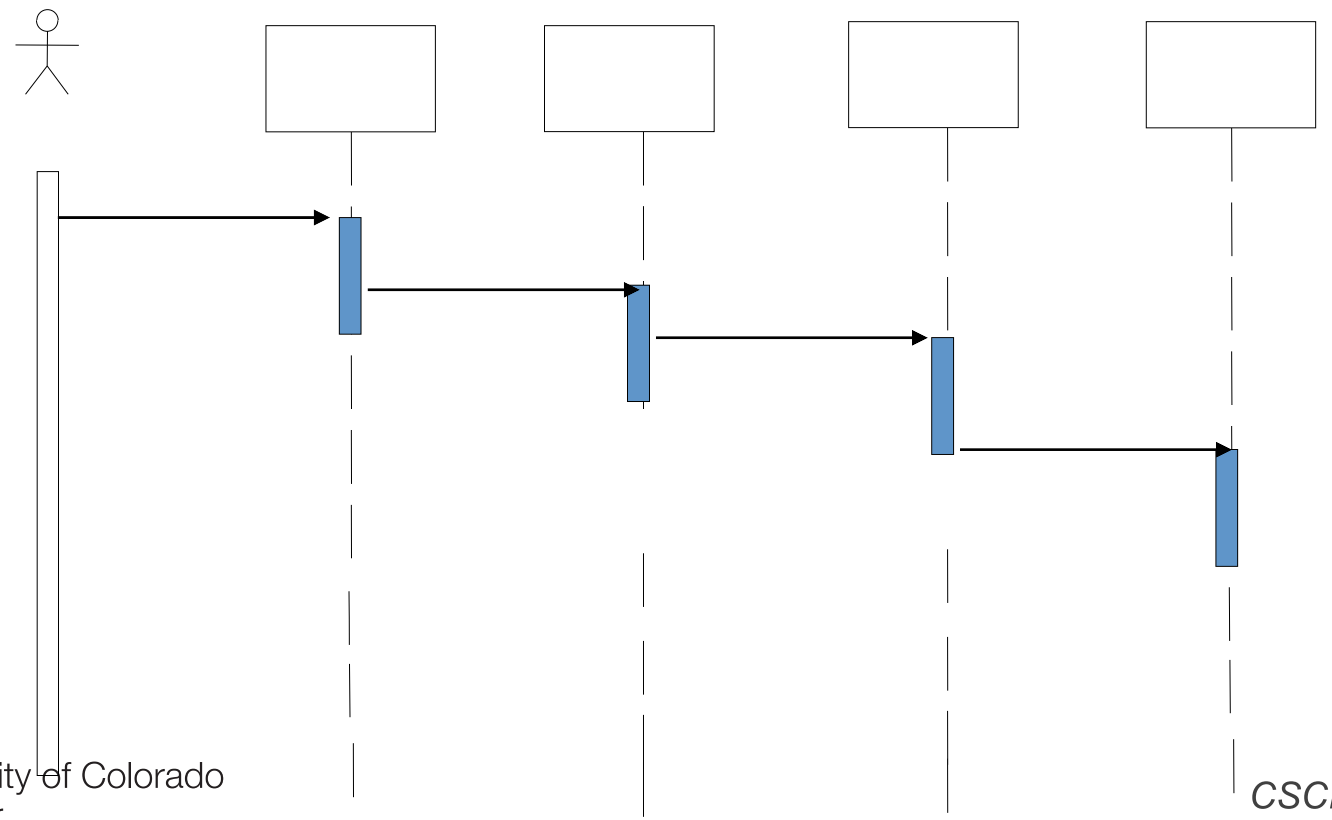
Fork Diagram

- Much of the dynamic behavior is placed in a single object, usually the control object. It knows all the other objects and often uses them for direct questions and commands.



Stair Diagram

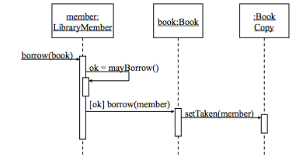
- The dynamic behavior is distributed.
- Each object delegates some responsibility to other objects.
- Each object knows only a few of the other objects and knows which objects can help with a specific behavior.



Fork or Stair?

- Which of these diagram types should be chosen?
- Object-oriented fans claim that the stair structure is better
  - The more the responsibility is spread out, the better
- However, this is not always true. Better heuristics:
- Decentralized control structure
  - The operations have a strong connection
  - The operations will always be performed in the same order
- Centralized control structure (better support of change)
  - The operations can *change order*
  - New operations can be inserted as a result of new requirements

Sequence Diagrams - How

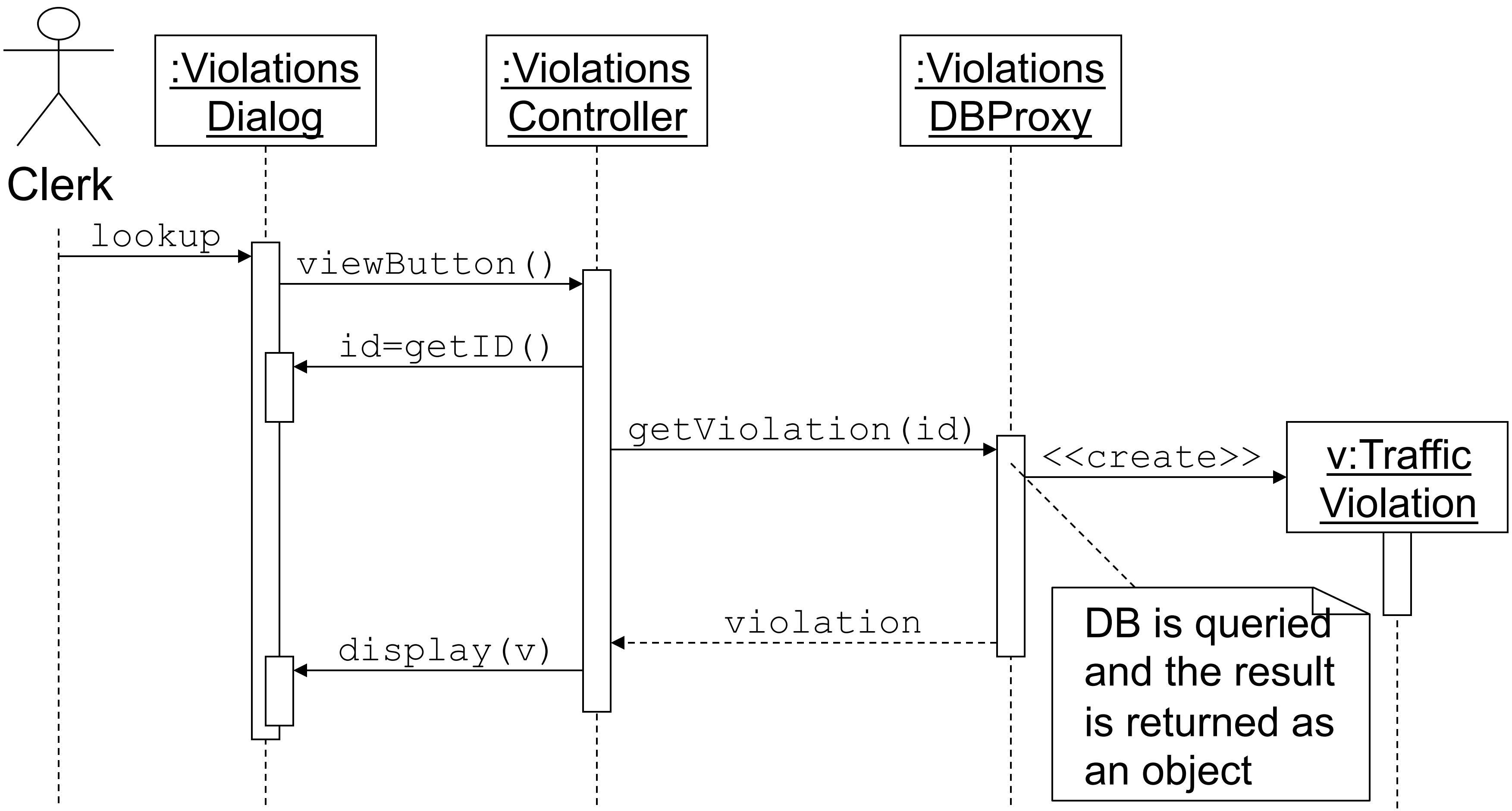


General technique to build an sequence diagram from a use case.

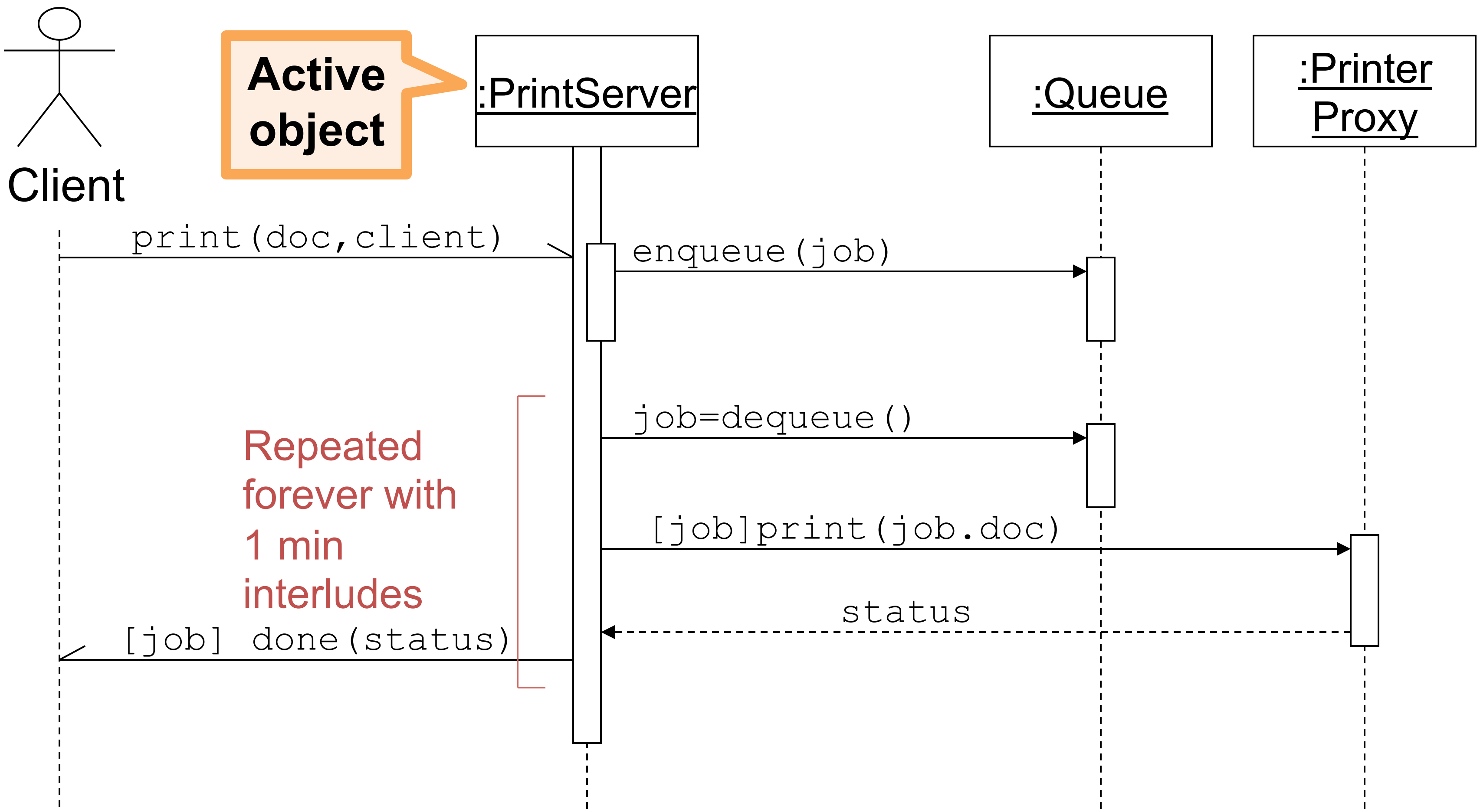
- Draw System as black box on right side
- For each actor that directly operates on the System, draw a stick figure and a lifeline.
- For each System events that each actor generates in use case, draw a message.
- Optionally, include use case text to left of diagram

More Examples

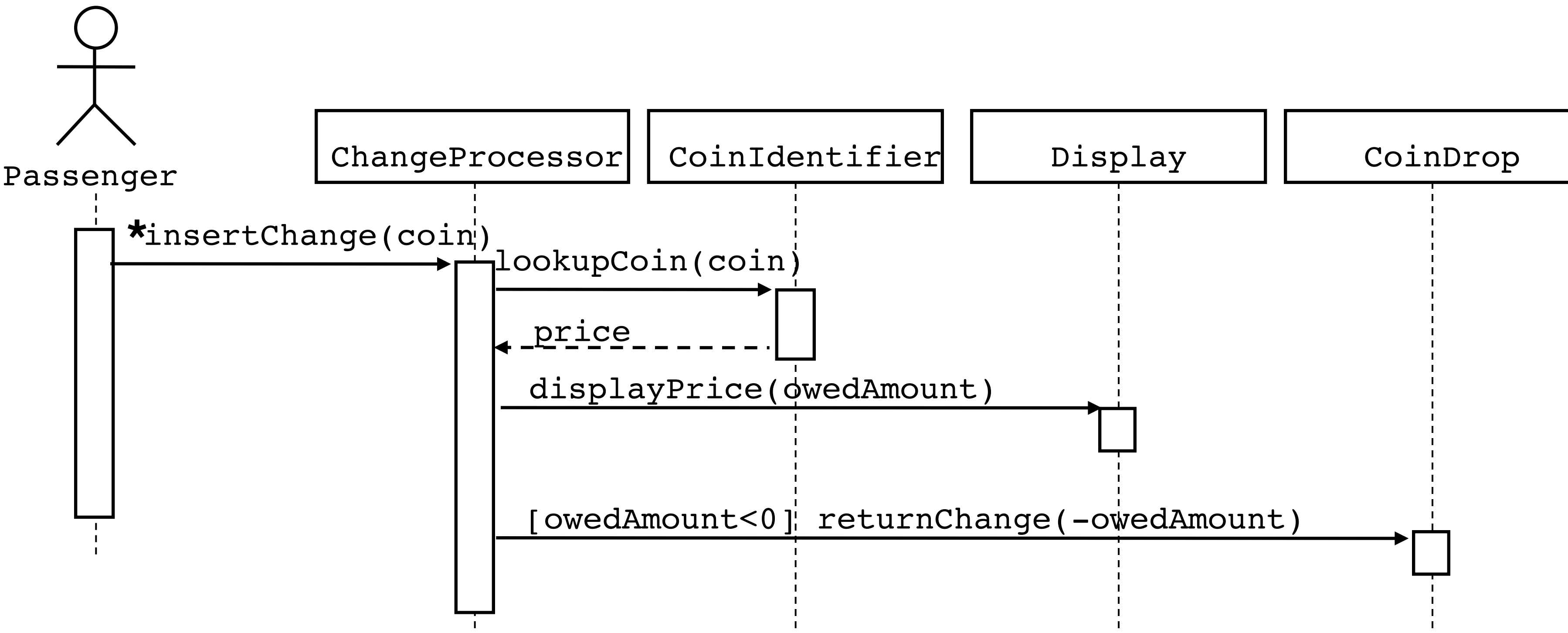
Example – Lookup Traffic Violation



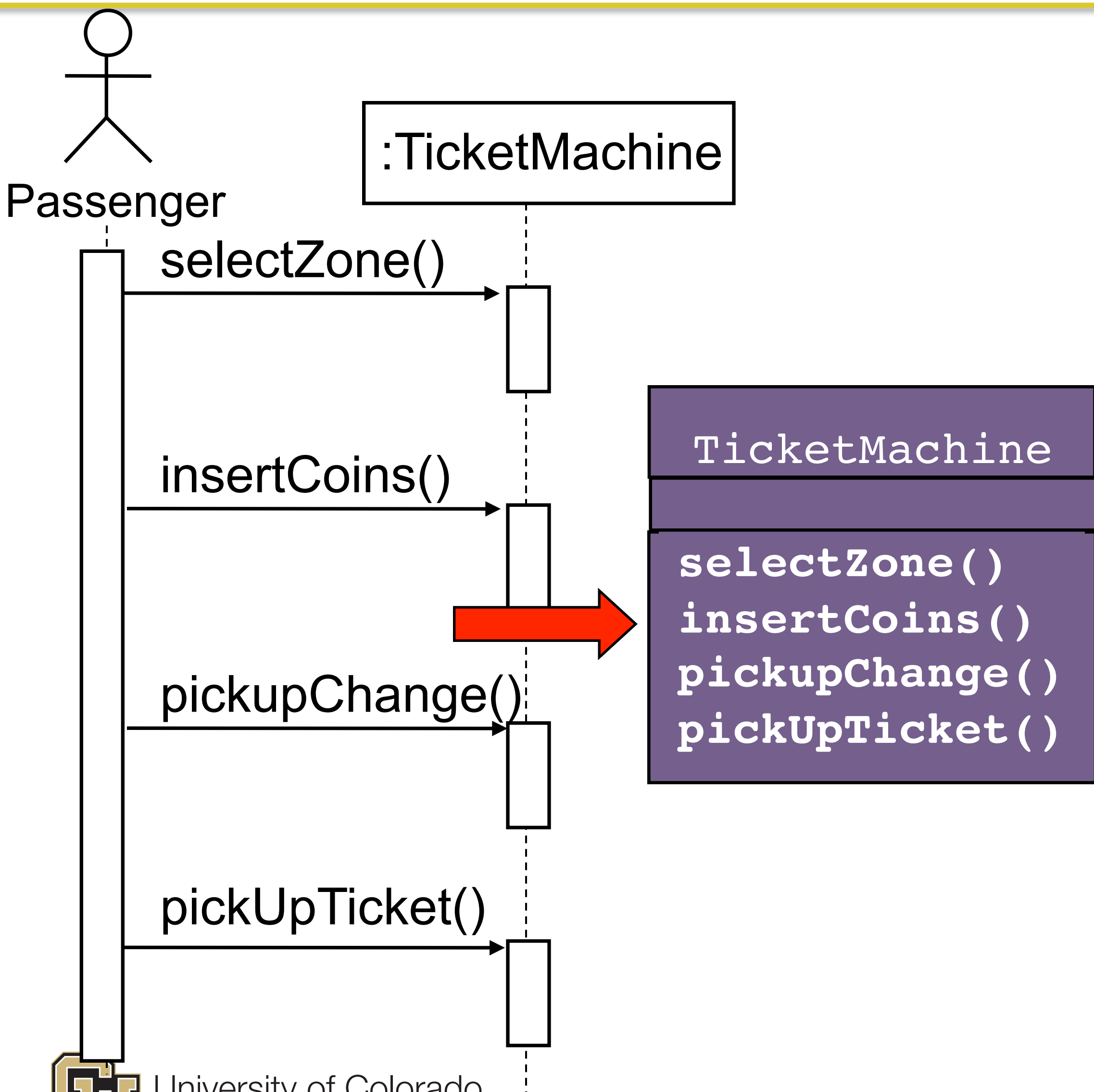
Example – Printing a Document



Sequence Diagrams: Iteration & Condition



Use Cases – Sequence D. – Class D.



- Used during analysis
  - To refine use case descriptions
  - Find additional objects ("participating objects")
- Used during system design
  - Refine subsystem interfaces