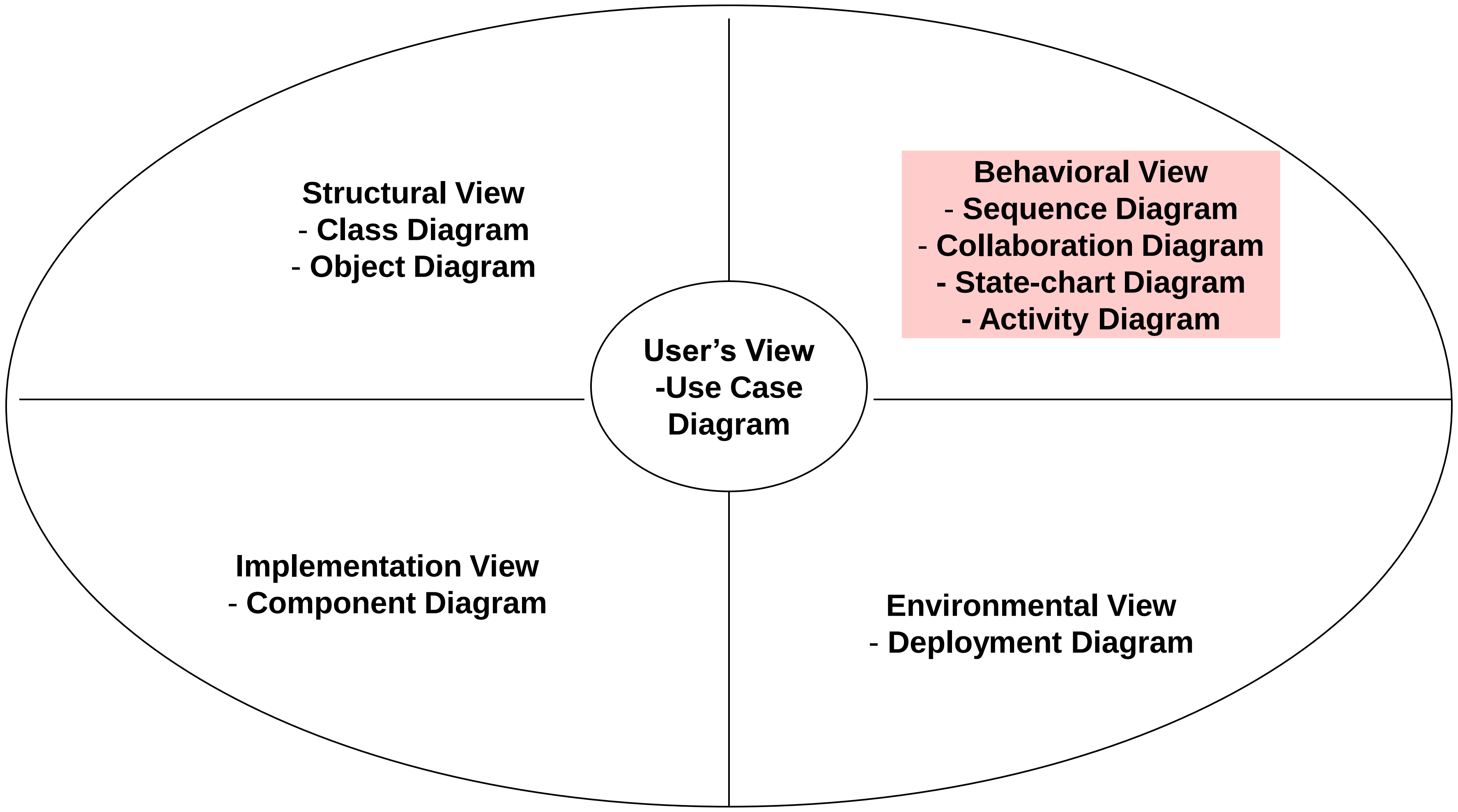
This image cannot currently be displayed.

Chapter 4

Behavioral View

UML Diagrams



Diagrams and views in UML

Types of Diagrams

- UML diagrams can be divided broadly into two categories
  - **Structure Diagrams –** Used to model the static structure of a system, for example- class diagram, object diagram, deployment diagram etc.
  - **Behavior diagram –** Used to model the dynamic change in the system over time. They are used to model and construct the functionality of a system. So, a behavior diagram simply guides us through the functionality of the system using Use case diagrams, Interaction diagrams, Activity diagrams and State Chart diagrams.

State Chart Diagram

State Chart Diagram

- A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.
- A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.
- Based on the work of David Harel [1990]
- Model how the state of an object changes in its lifetime
- Based on Finite State Machine (FSM) formalism
- FSM is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of *states* at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a *transition.*

State Chart Diagram

- At any given point in time, the system or object is in a certain state. The states are specific to a component/object of a system.
  - Being in a state means that it will behave in a *specific way* in response to any events that occur.
- Some events will cause the system to change state.
  - In the new state, the system will behave in a different way to events.
- A state diagram is a directed graph where the nodes are states and the arcs are transitions.

State Chart Diagram Cont…

- State chart avoids the problem of state explosion of FSM

Finite State Machines (FSM) are widely used for modeling system behavior, but they often suffer from **state explosion**—a problem where the number of states grows exponentially as system complexity increases, making it difficult to manage or design.

State Charts address this problem by introducing advanced features to compactly represent complex systems, including:

Hierarchy:
State charts allow states to be organized hierarchically. Substates (nested states) reduce redundancy by encapsulating common behavior in a parent state. This avoids creating multiple flat states for similar transitions. Hierarchical model of a system, represents composite nested states

Concurrency:
State charts can model parallel behaviors using orthogonal regions. Instead of representing parallel states individually, state charts combine them in a modular way.

State Chart Diagram Cont…

Events and Actions:
State charts explicitly manage events, actions, and conditions, reducing the need for complex and separate transitions for every possible scenario.

History Mechanism:
A history state in state charts lets a system remember the last active substate, eliminating the need to track it manually in a flat FSM.

These features collectively help simplify complex systems and avoid the **state explosion problem** that traditional FSMs face.

Purpose of Statechart Diagrams

- Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.
- Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.
- Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

How to Draw a Statechart Diagram?

- Before drawing a Statechart diagram we should clarify the following points −
  - Identify the important objects to be analyzed.
  - Identify the states.
  - Identify the events.

States

- At any given point in time, the system is in one state.
- It will remain in this state until an event occurs that causes it to change state.
- A state is represented by a rounded rectangle containing the name of the state.
- Special states:
  - A black circle represents the *start state*
  - A circle with a ring around it represents an *end state*

States

- **Initial State:** This shows the starting point or first activity of the flow. Denoted by a solid circle. This is also called as a "pseudo state," where the state has no variables describing it further and no activities.
- **Final State:** The end of the state diagram is shown by a bull's eye symbol, also called a final state. A final state is another example of a pseudo state because it does not have any variable or action described.

Transitions

- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.

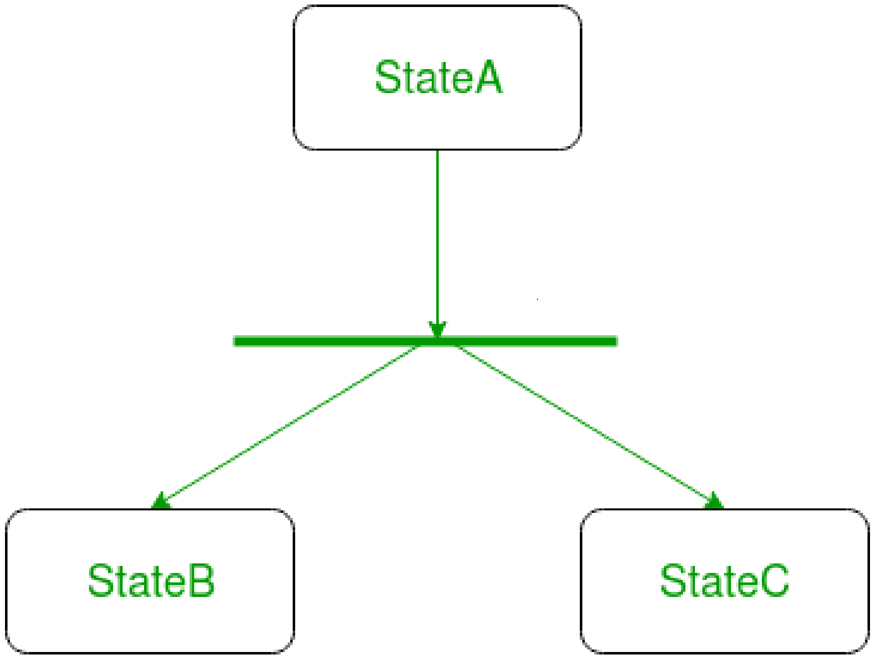
- The label on each transition is the event that causes the change of state.

Transitions

- An arrow indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow, separated by a slash. Transitions that occur because the state completed an activity are called "**triggerless**" transitions. If an event has to occur after the completion of some event or action, the event or action is called the guard condition. The transition takes place after the guard condition occurs. This guard condition/event/action is depicted by square brackets around the description of the event/action (in other words, in the form of a Boolean expression).

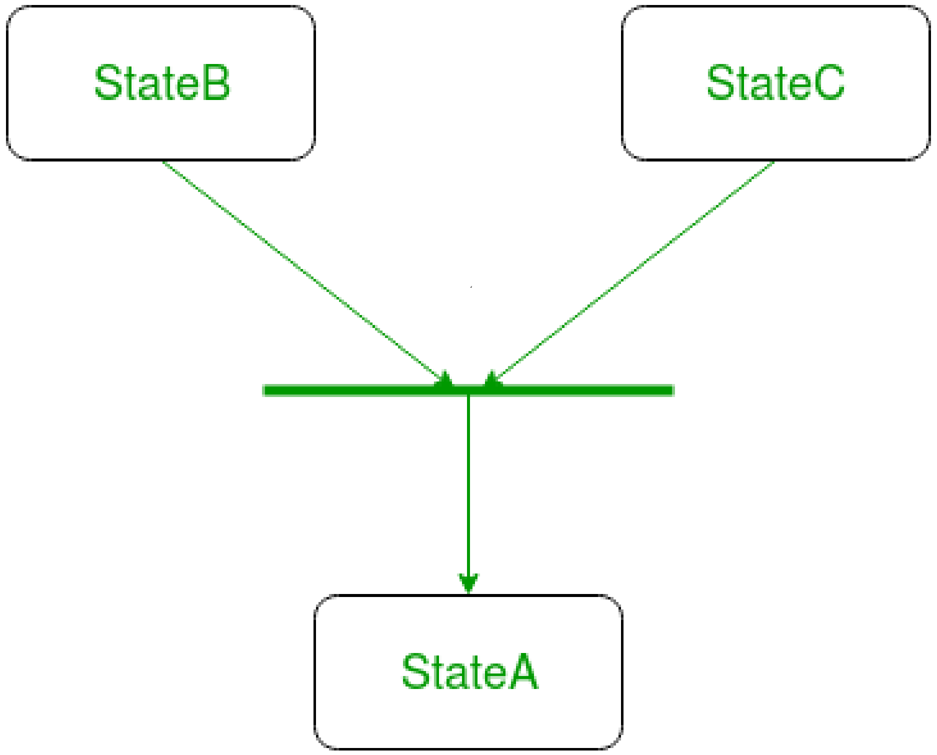
Fork

- We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



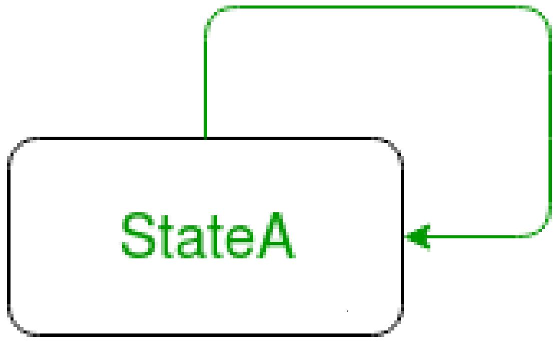
Join

- We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



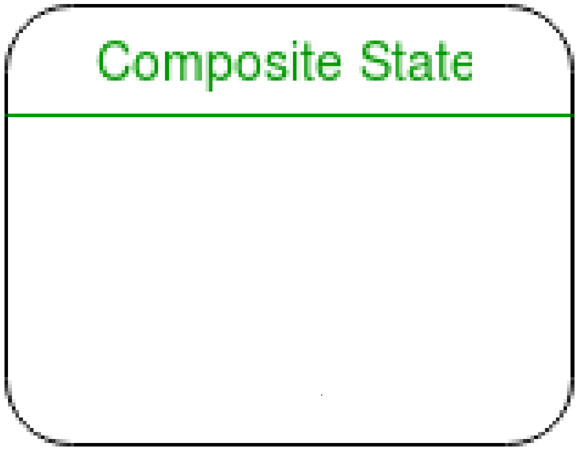
Self transition

- We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



Composite state

- We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.



History States

- A flow may require that the object go into a trance, or wait state, and on the occurrence of a certain event, go back to the state it was in when it went into a wait state - its last active state. This is shown in a State diagram with the help of a letter H enclosed within a circle.
- For example:
- A process has substates: **Processing**, **Paused**, and **Terminated**.
- If the system goes into a "wait" state from **Paused**, the history state allows it to return to **Paused** when resumed.

Event and Action

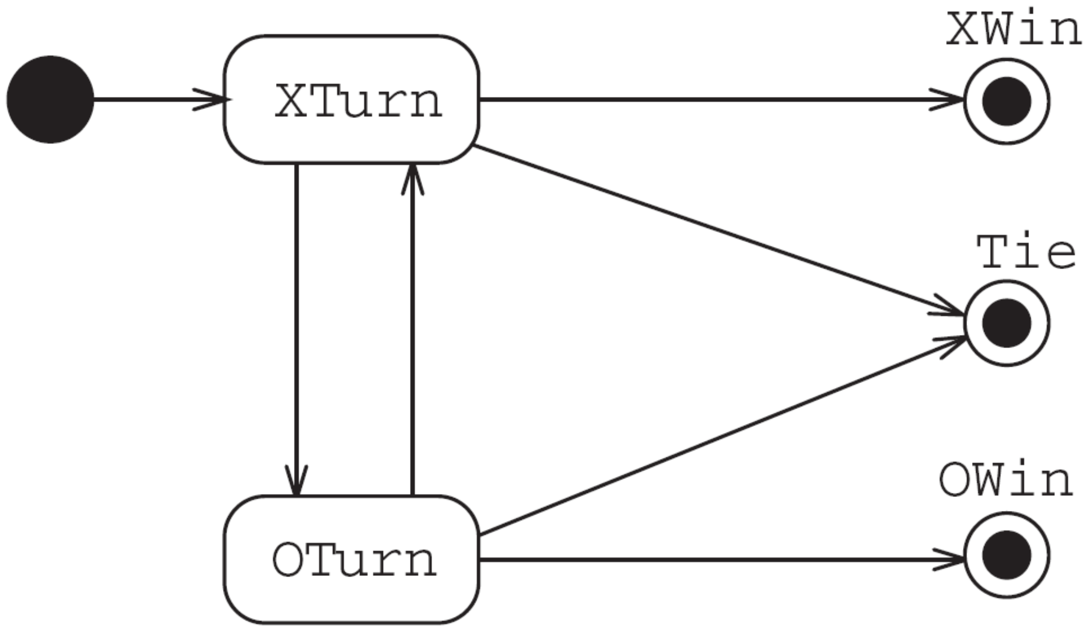
- A trigger that causes a transition to occur is called as an event or action. Every transition need not occur due to the occurrence of an event or action directly related to the state that transitioned from one state to another. As described above, an event/action is written above a transition that it causes

Signal

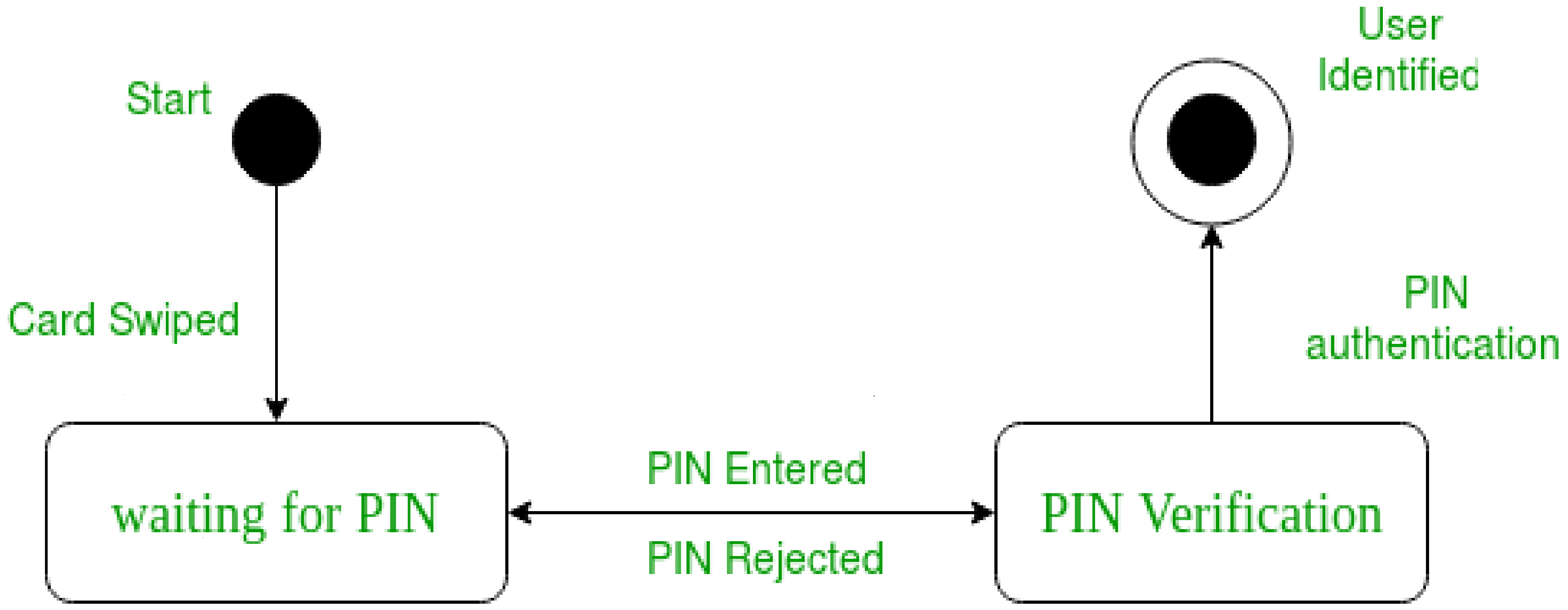
- When an event causes a message/trigger to be sent to a state, that causes the transition; then, that message sent by the event is called a signal. Represented as a class with the <<Signal>> icon above the action/event.

State diagrams – an example

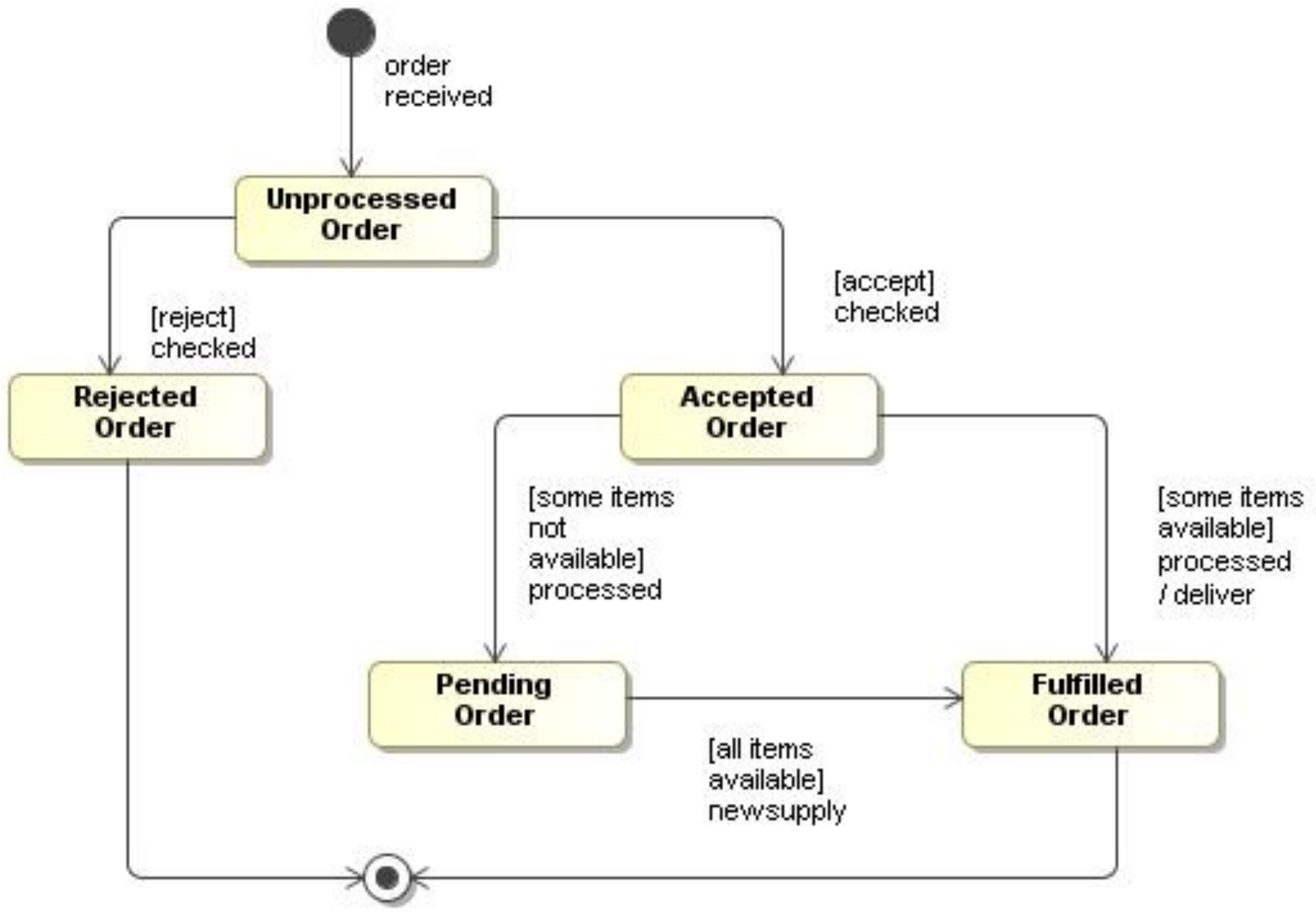
tic-tac-toe game (also called noughts and crosses)



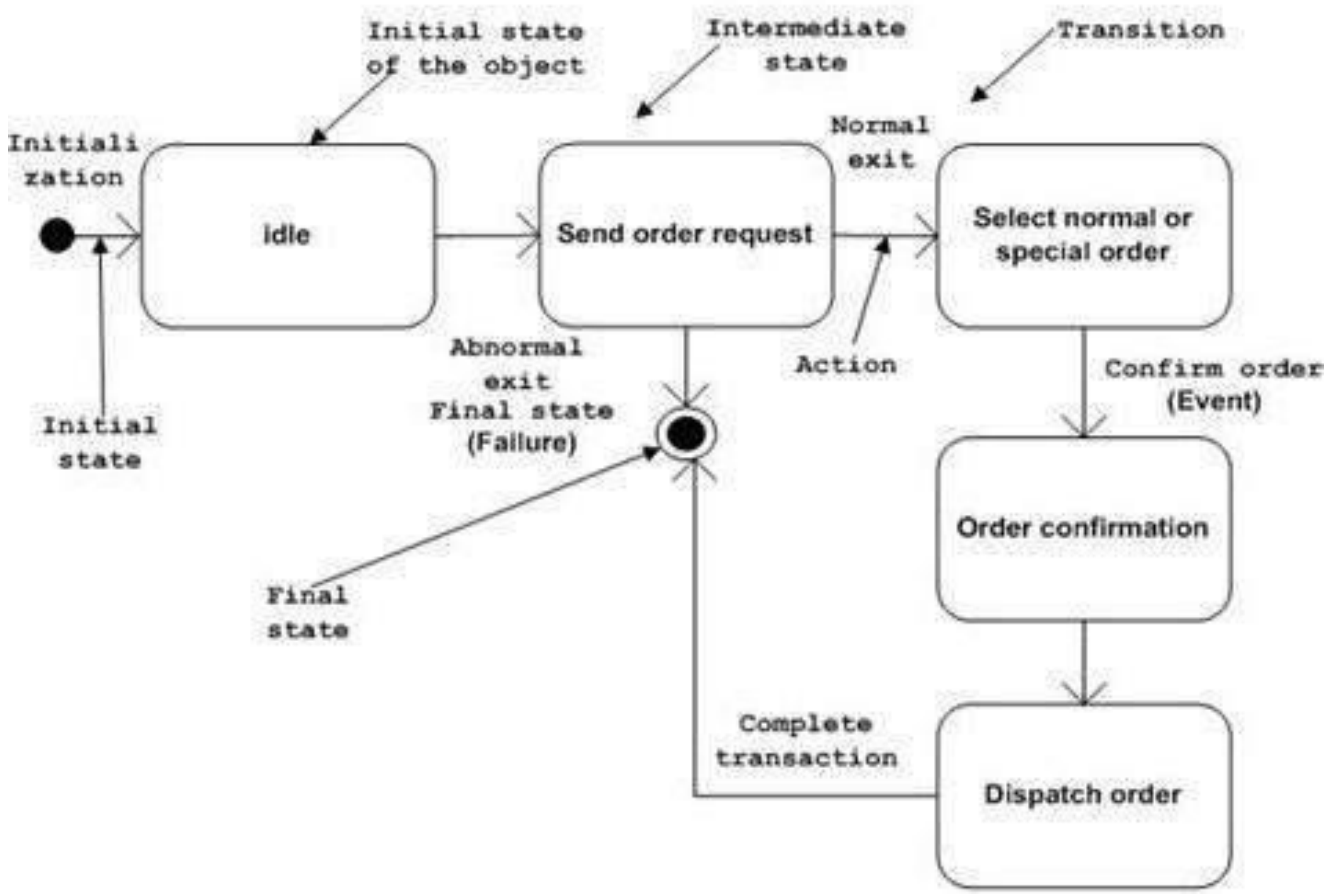
A state diagram for user verification



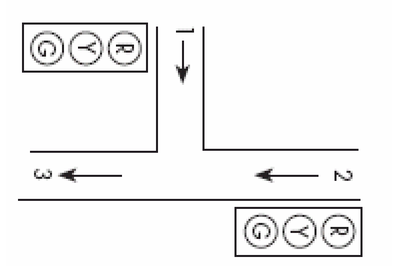
State Chart Diagram for Order Processing

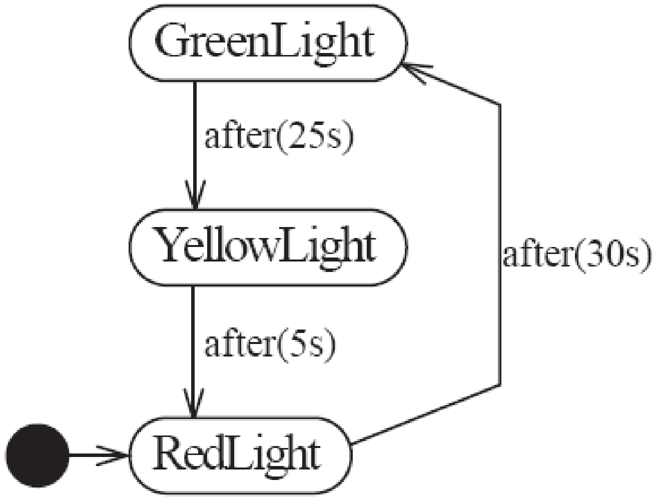


Statechart diagram of an order management system

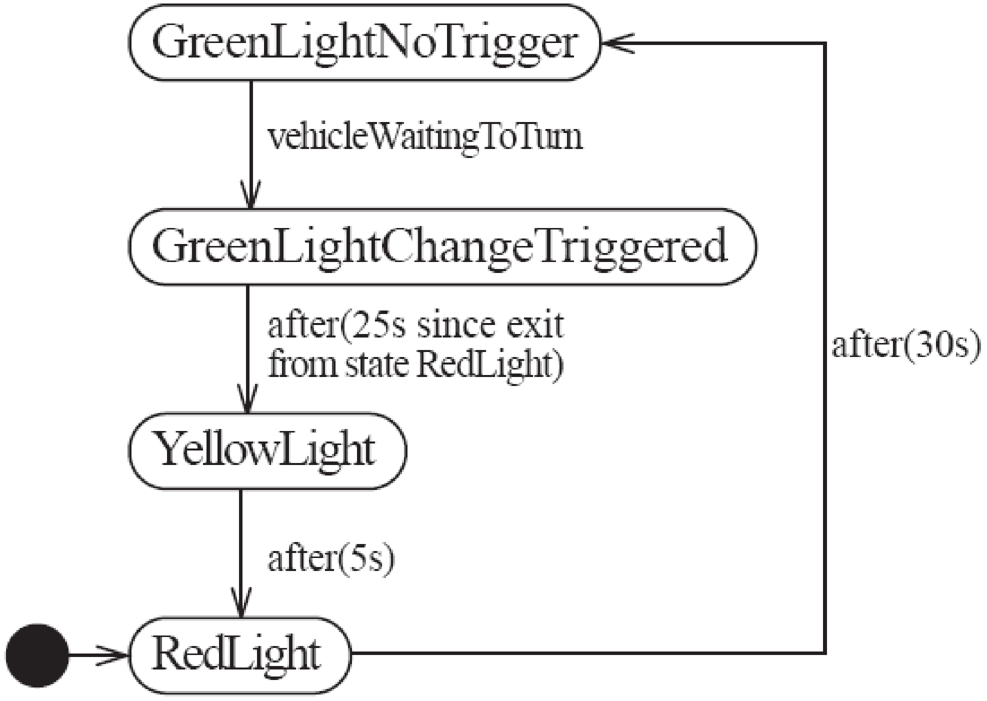


State diagrams – an example of transitions with time-outs and conditions



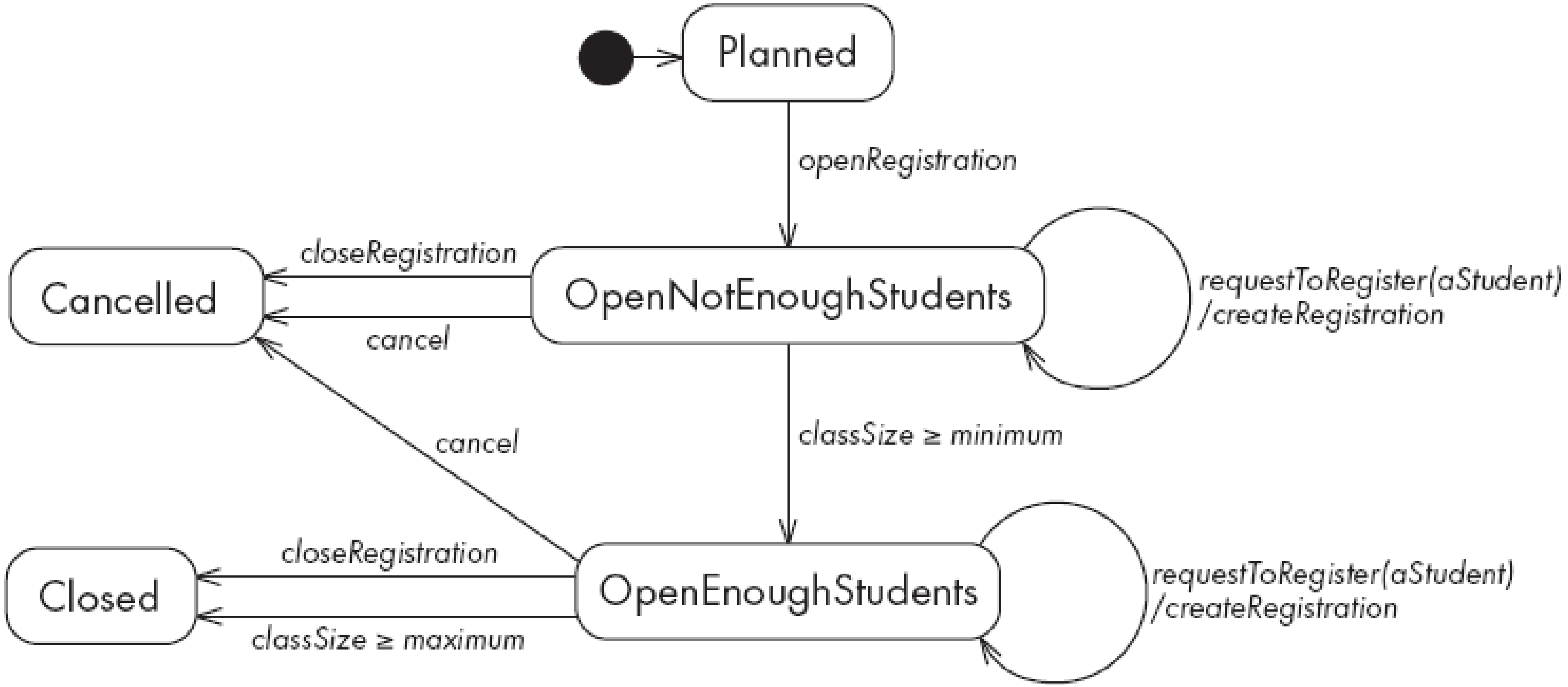


(a)



(b)

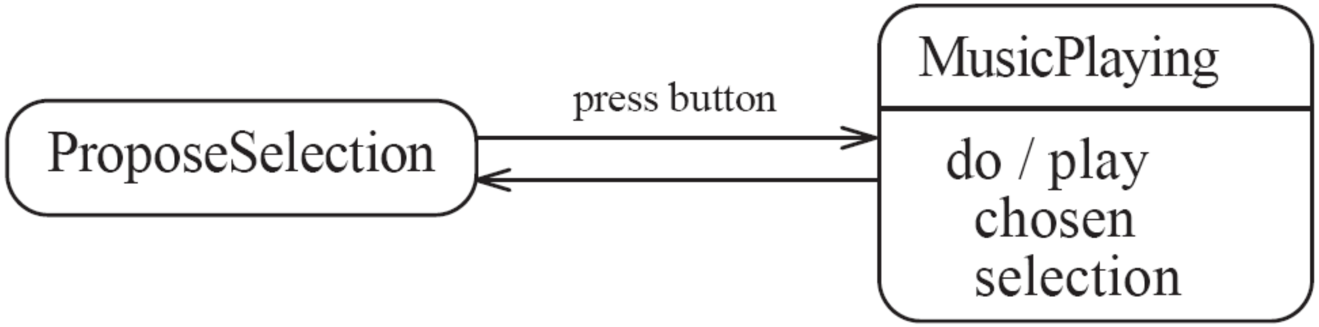
State diagrams – an example with conditional transitions



Activities in state diagrams

- An *activity* is something that takes place while the system is *in* a state.
  - It takes a period of time.
  - The system may take a transition out of the state in response to completion of the activity,
  - Some other outgoing transition may result in:
    - The interruption of the activity, and
    - An early exit from the state.

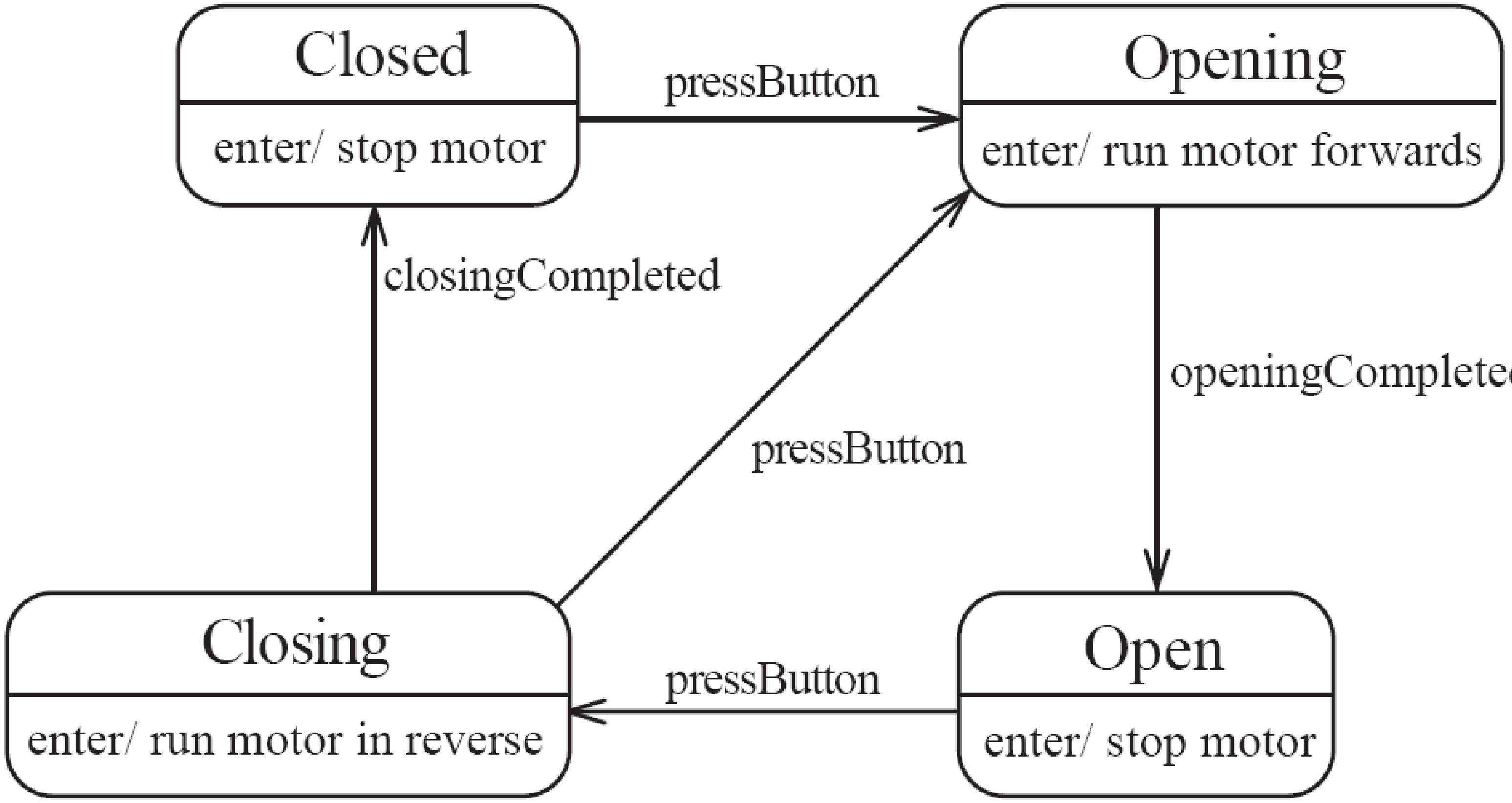
State diagram – an example with activity



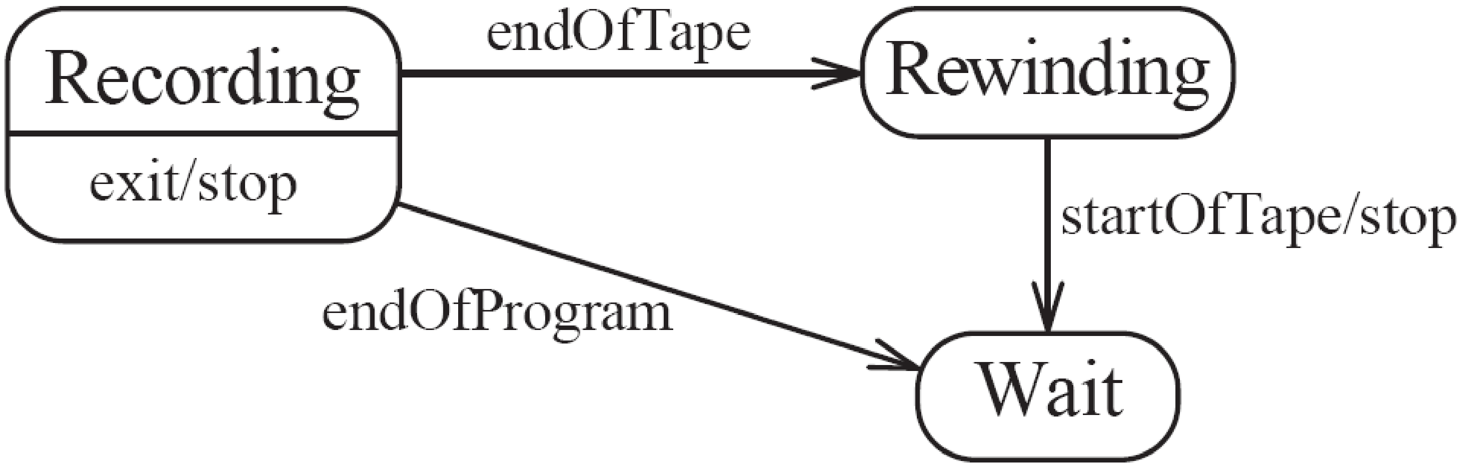
Actions in state diagrams

- An *action* is something that takes place effectively *instantaneously*
  - When a particular transition is taken place,
  - Upon entry into a particular state, or
  - Upon exit from a particular state
- An action should consume no noticeable amount of time

State diagram – an example with actions

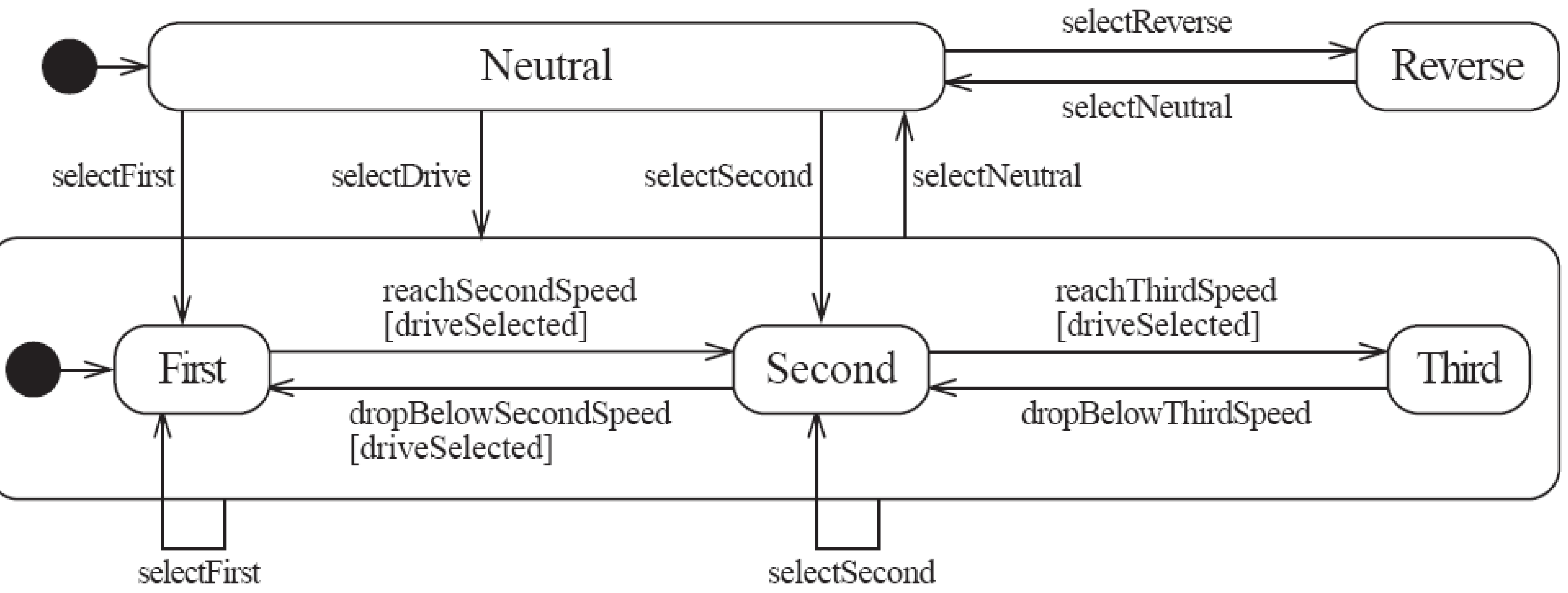


State diagrams – another example

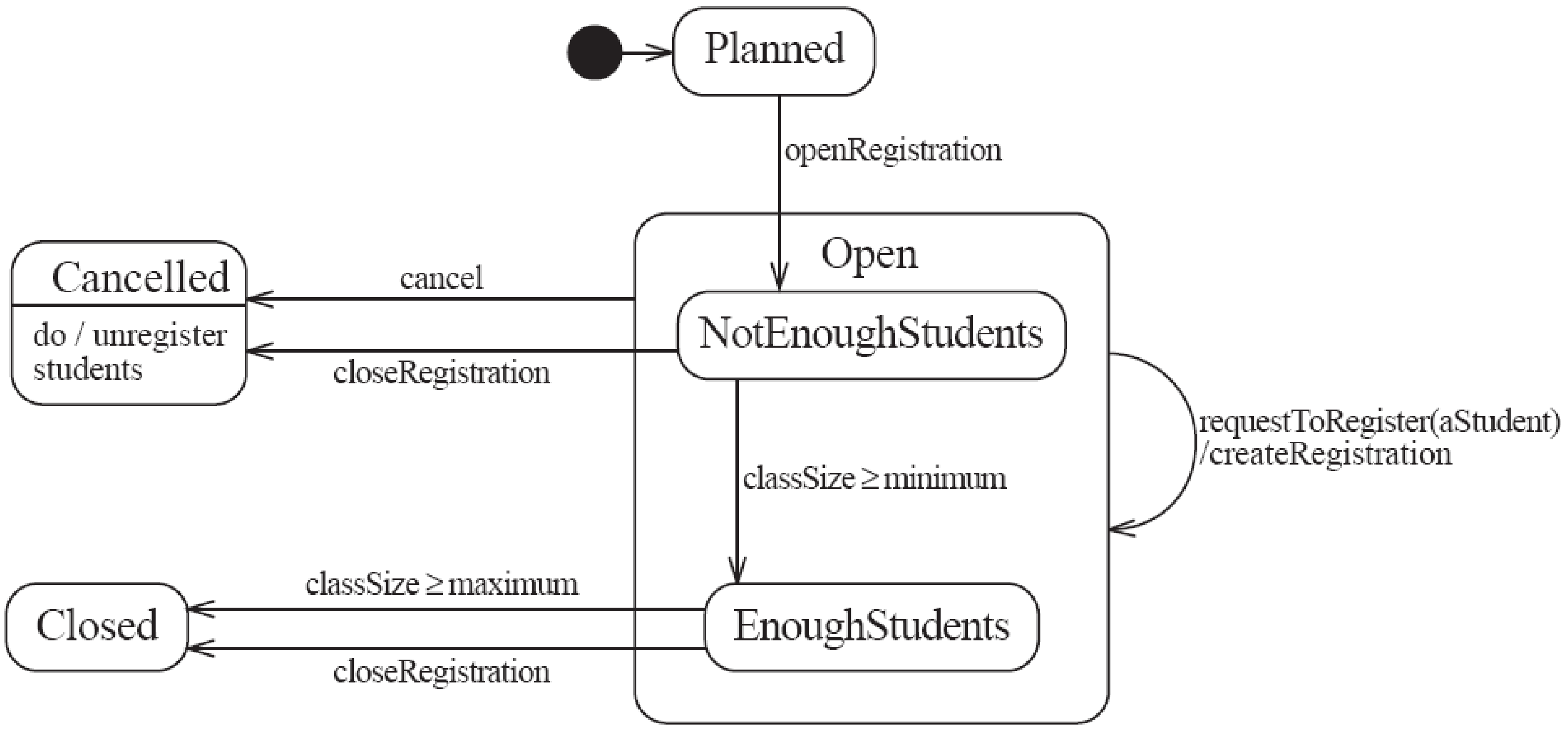


Nested substates and guard conditions

- A state diagram can be nested inside a state.
  - The states of the inner diagram are called *substates*.



State diagram – an example with substates



Difference between state diagram and flowchart

- The basic purpose of a **state diagram** is to portray various changes in state of the class and not the processes or commands causing the changes. However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

This image cannot currently be displayed.

Interaction Diagram

Interaction Diagram

- Interaction diagrams are used to model the *dynamic* aspects of a software system
- They help you to *visualize* how the system runs.
- Models how groups of objects *collaborate* to realize some behavior
- An interaction diagram is often built from a use case and a class diagram
- The objective is to show how a set of objects accomplish the required *interactions* with an actor.

Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called an *interaction*.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.

Elements found in interaction diagrams

- Instances of classes
  - Shown as *boxes* with the class and object identifier underlined
- Actors
  - Use the *stick-person* symbol as in use case diagrams
- Messages
  - Shown as *arrows* from actor to object, or from object to object

Creating interaction diagrams

- You should develop a class diagram and a use case model before starting to create an interaction diagram.
  - There are two kinds of interaction diagrams:
    - *Sequence diagrams*
    - *Communication or collaboration diagrams*

This image cannot currently be displayed.

Sequence Diagram

Sequence Diagram

- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
- Shows interaction among objects as a two dimensional chart
- Objects are shown as boxes at top
- If object created during execution then shown at appropriate place
- Objects existence are shown as dashed lines (lifeline)
- lifeline becomes a broad box, called an activation box during the live activation period.

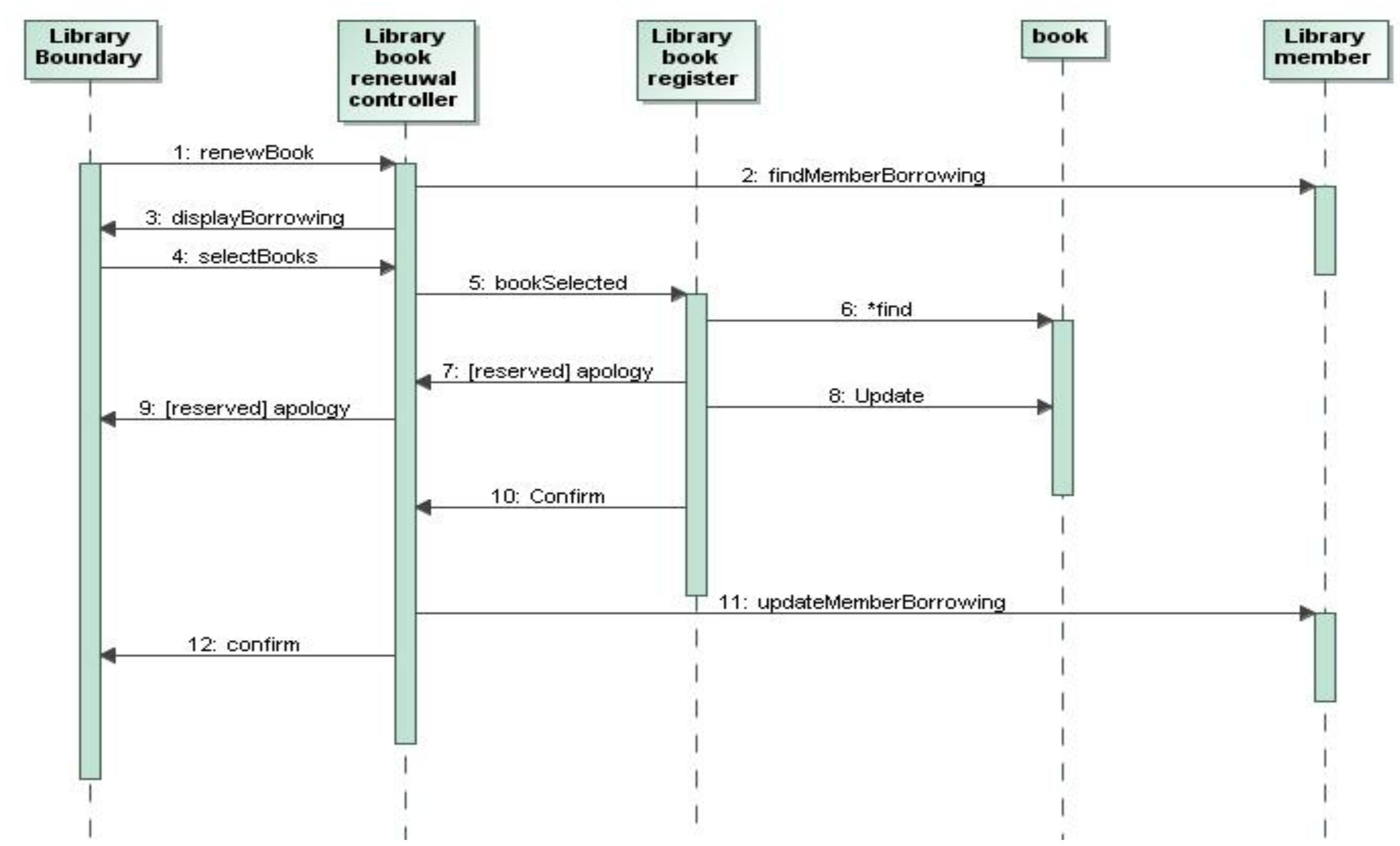
Sequence Diagram Cont…

- An actor that initiates the interaction is often shown on the left.
- The vertical dimension represents time.
- Messages are shown as arrows
- Each message labelled with corresponding message name
- Each message can be labelled with some control information
- Two types of control information

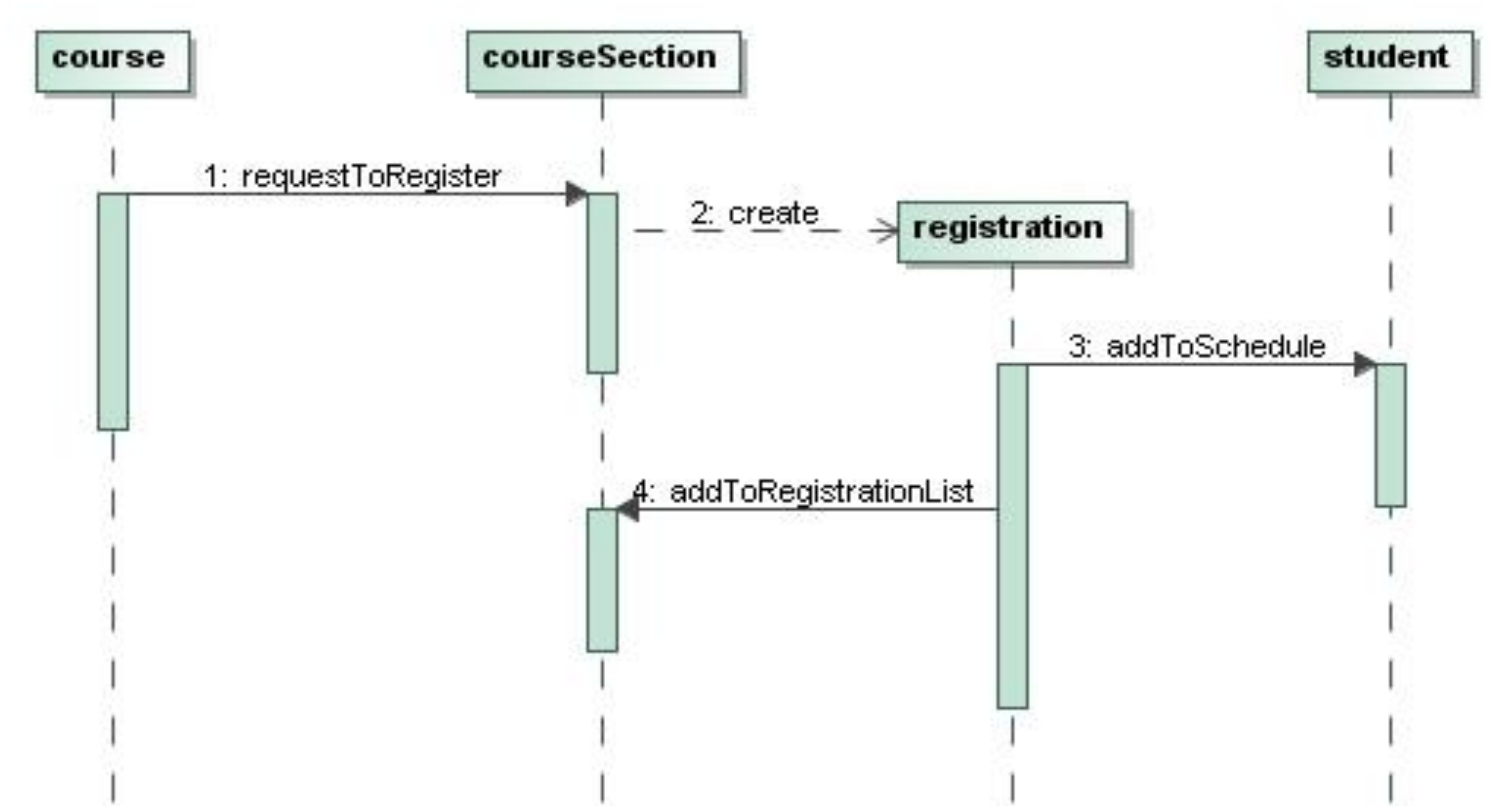
– condition ([])

– iteration (*)

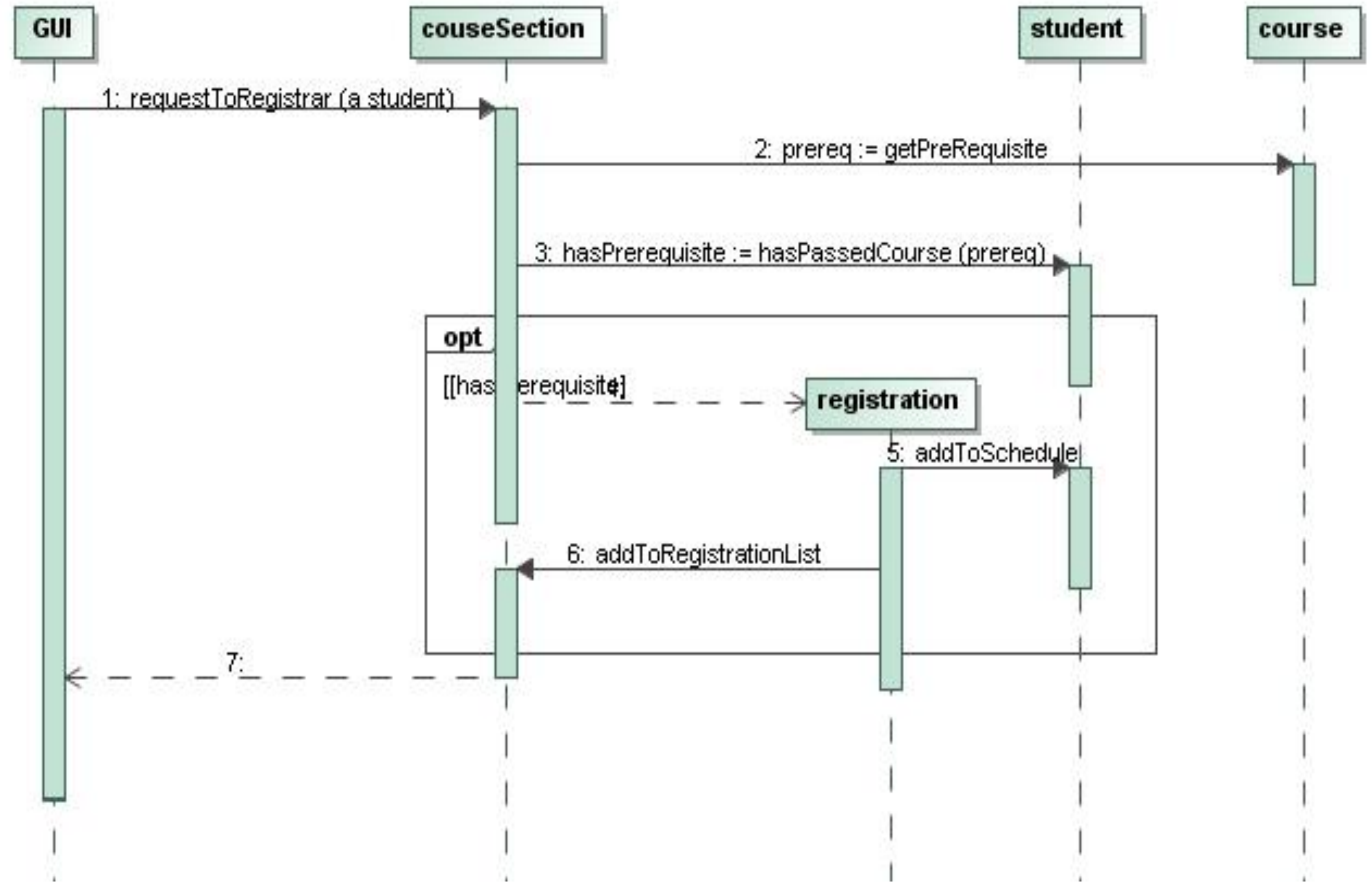
An Example of Sequence Diagram



Another Example of Sequence Diagram(version 0)



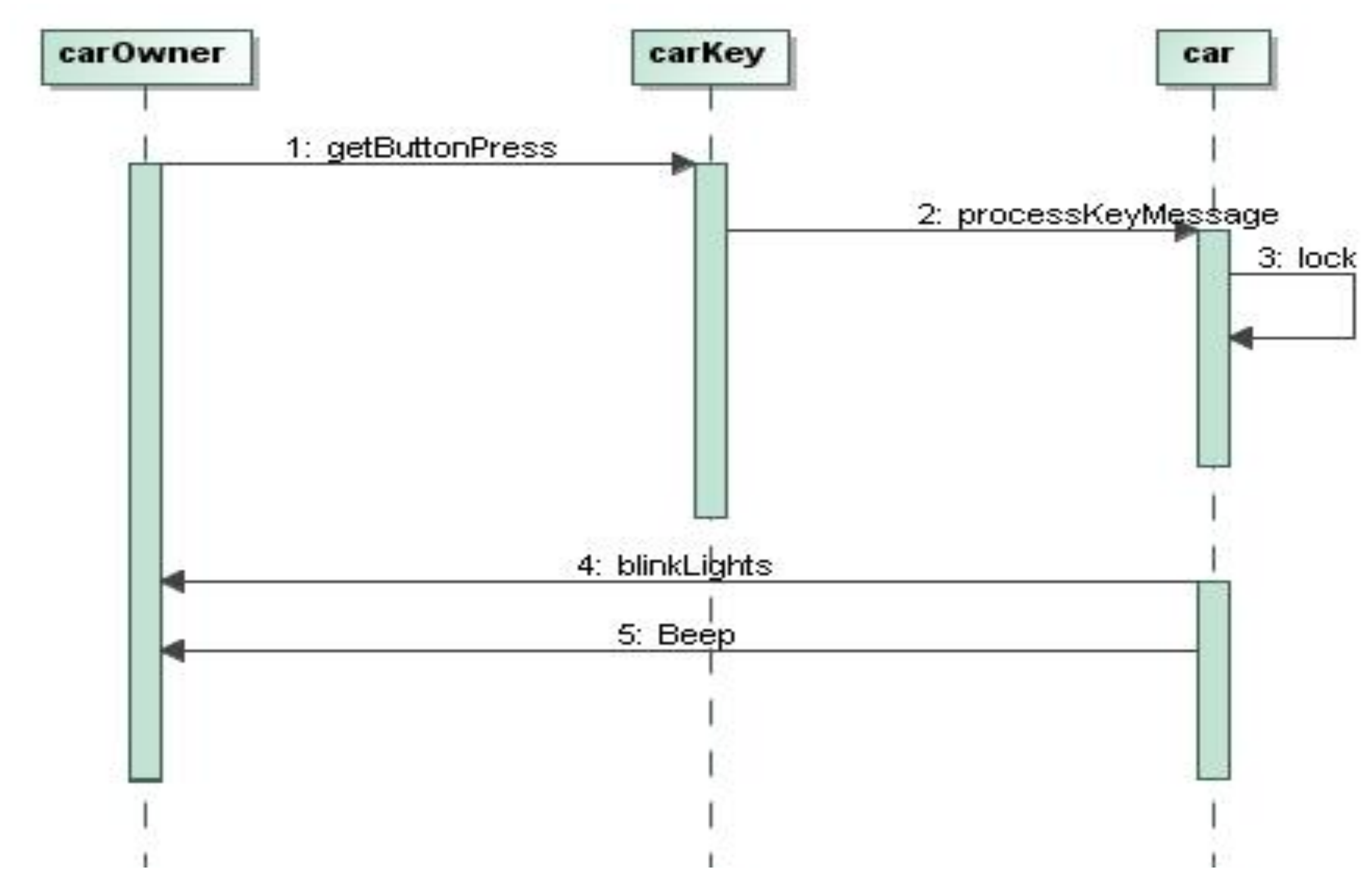
Another Example of Sequence Diagram (version 1)


GUI
couseSection
student
course
1: requestToRegistrar (a student)
2: prereq := getPreRequisite
3: hasPrerequisite := hasPassedCourse (prereq)
opt
[[hasPrerequisite]]
registration
5: addToSchedule
6: addToRegistrationList
7:

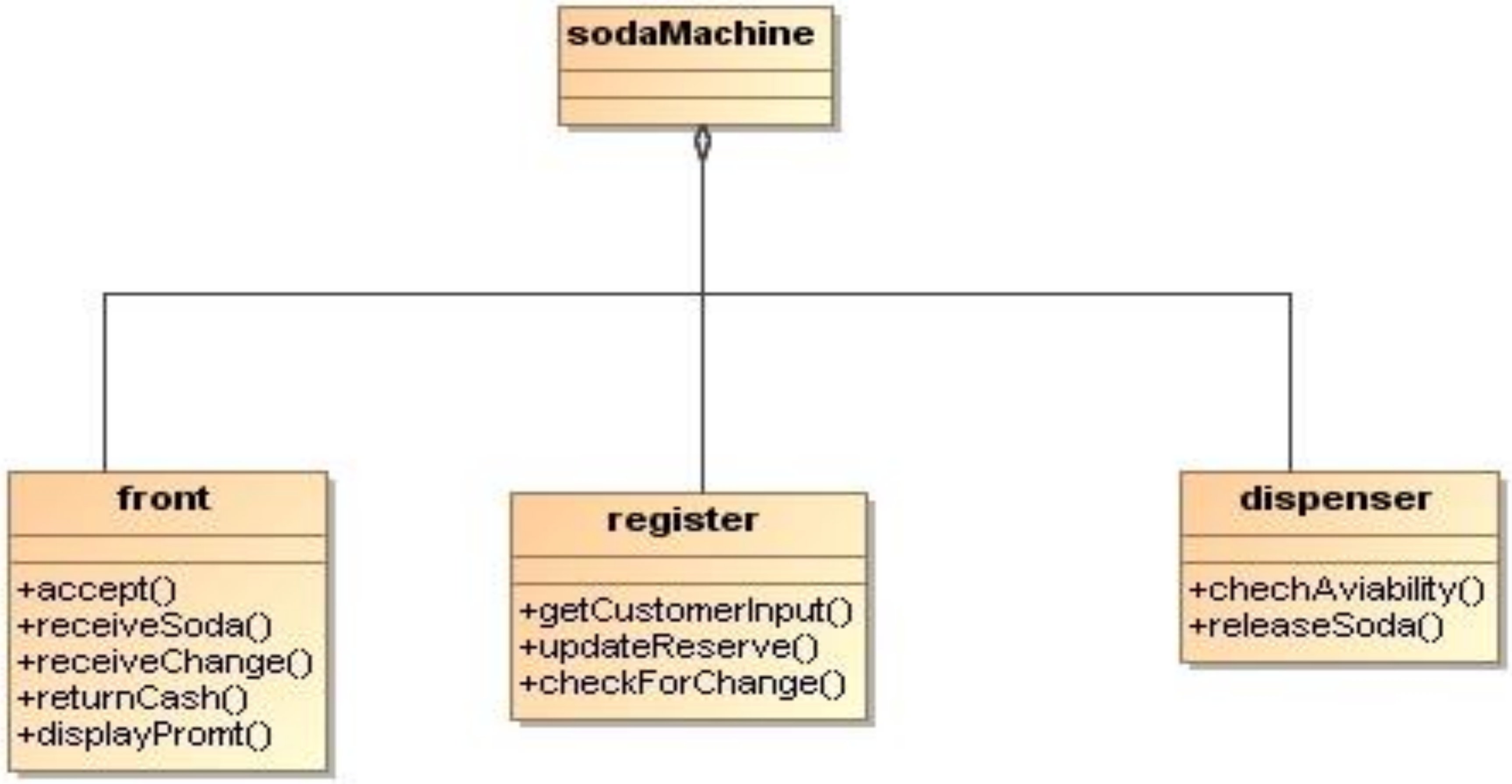
Another Example (car locking)

- By pressing a car key we can lock and unlock a car. When we press the lock button the car locks itself, then blinks its lights and beeps to let you know that it’s finished locking.

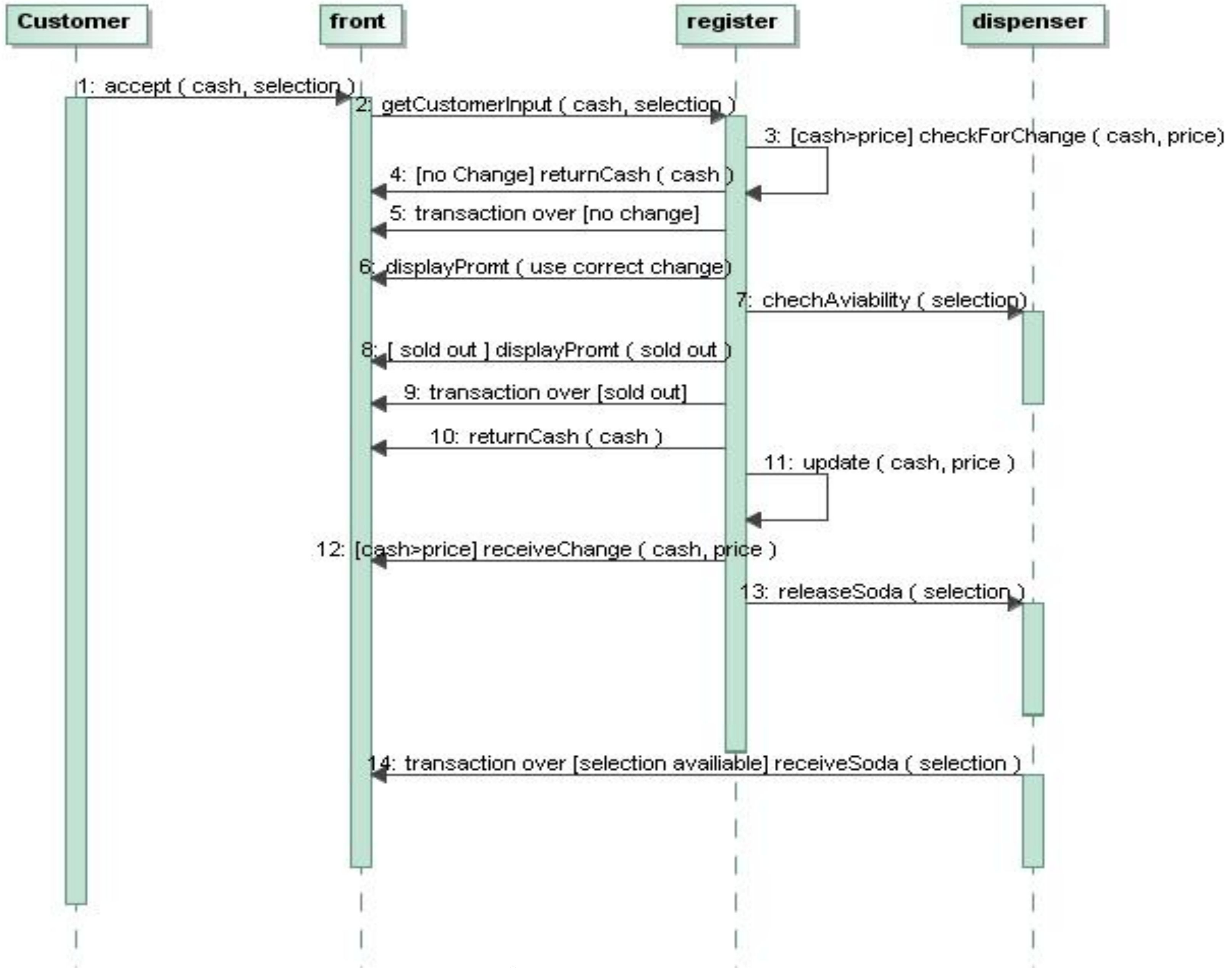
Sequence Diagram



Class Diagram of Soda Machine



Sequence Diagram of Soda Machine


Customer
front
register
dispenser
1: accept (cash, selection)
2: getCustomerInput (cash, selection)
3: [cash>price] checkForChange (cash, price)
4: [no Change] returnCash (cash)
5: transaction over [no change]
6: displayPromt (use correct change)
7: chechAviability (selection)
8: [sold out] displayPromt (sold out)
9: transaction over [sold out]
10: returnCash (cash)
11: update (cash, price)
12: [cash>price] receiveChange (cash, price)
13: releaseSoda (selection)
14: transaction over [selection availiable] receiveSoda (selection)

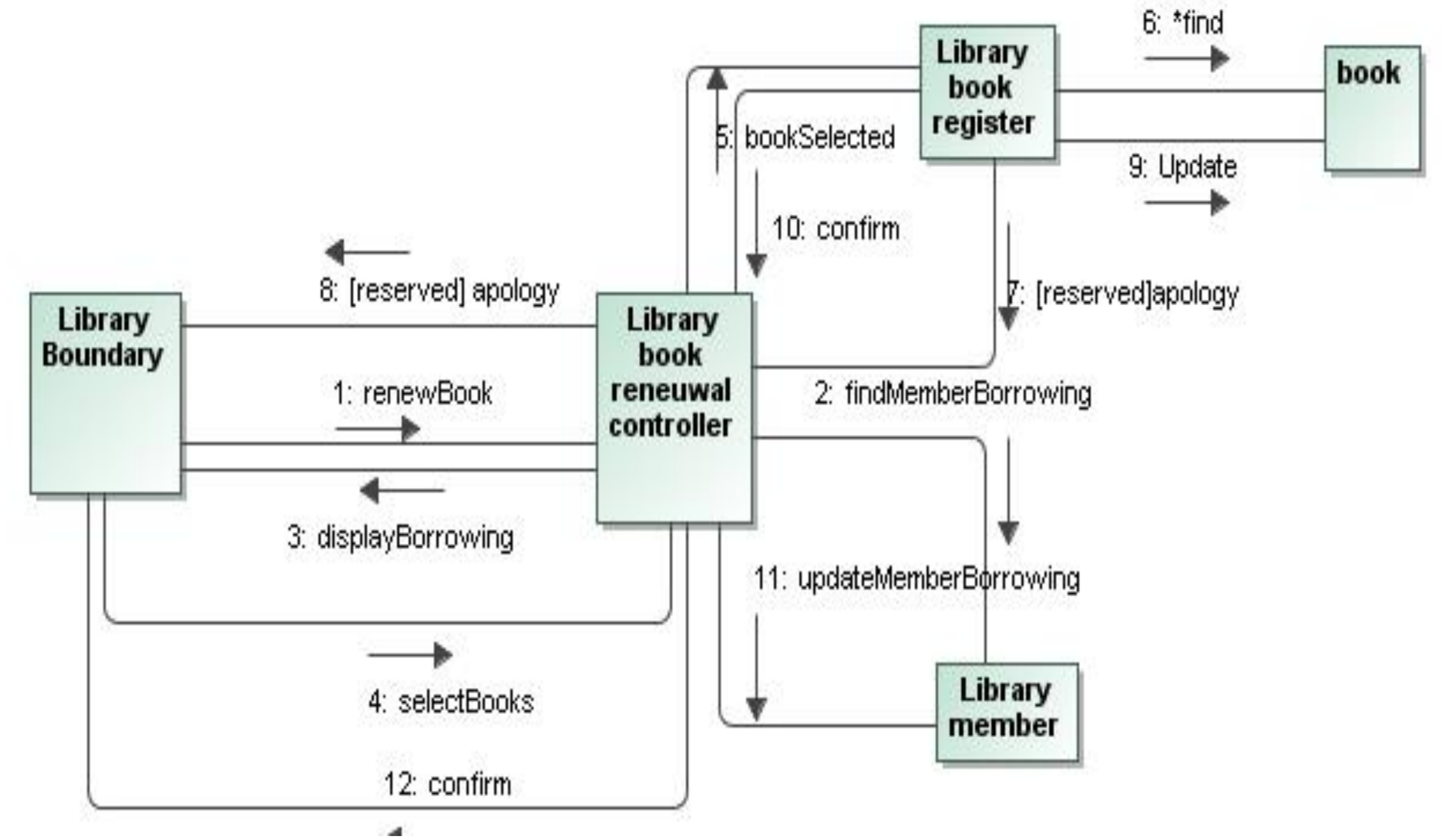
This image cannot currently be displayed.

Collaboration Diagram

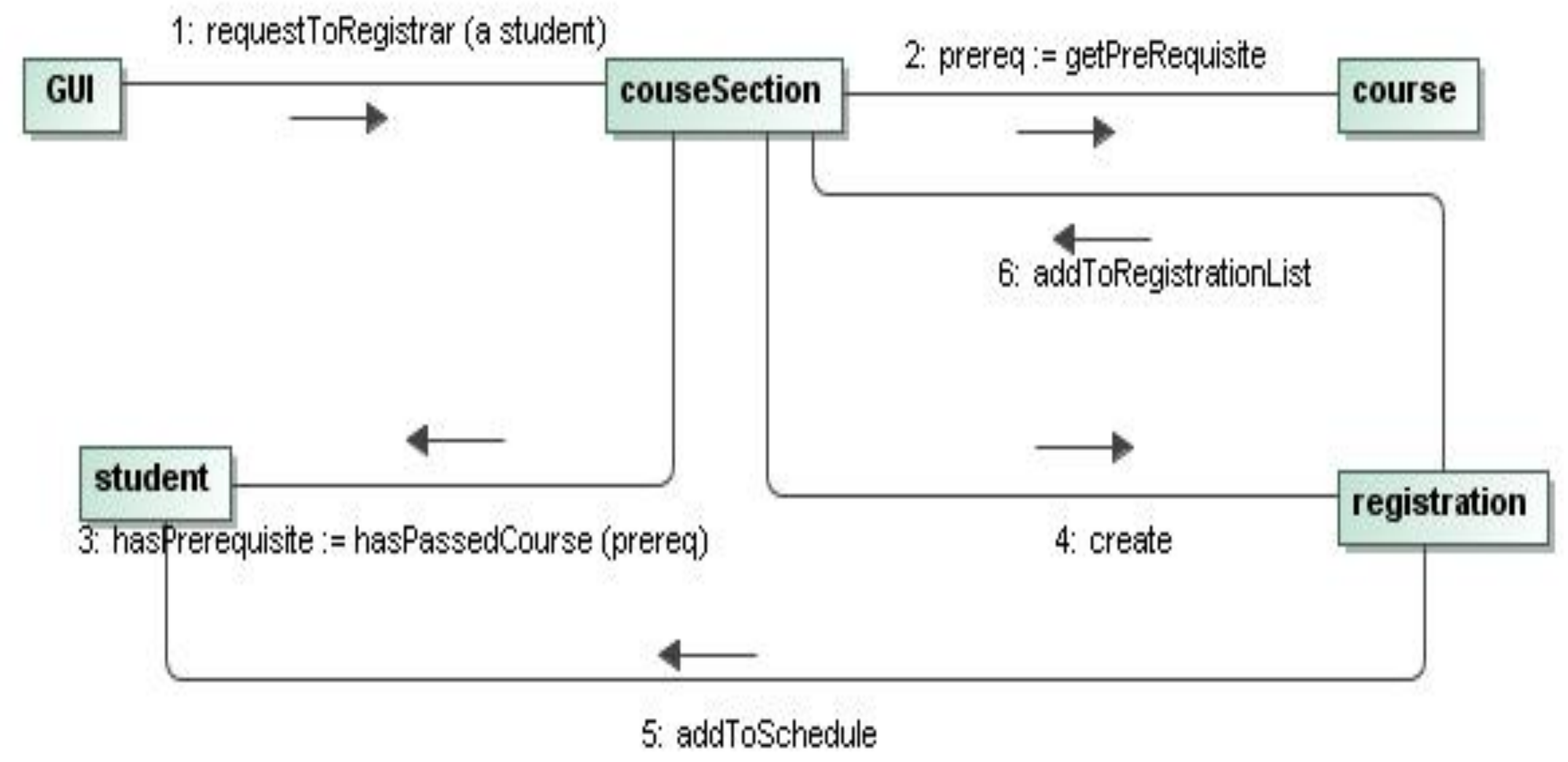
Collaboration or Communication Diagram

- Communication diagrams emphasise how the objects collaborate in order to realize an interaction
- Shows both **structural** and **behavioral** aspects
- Objects are **collaborator**, shown as boxes
- Messages between objects shown as a **solid line**
- A message is shown as a **labeled arrow** placed near the link
- Messages are prefixed with **sequence numbers** to show relative sequencing

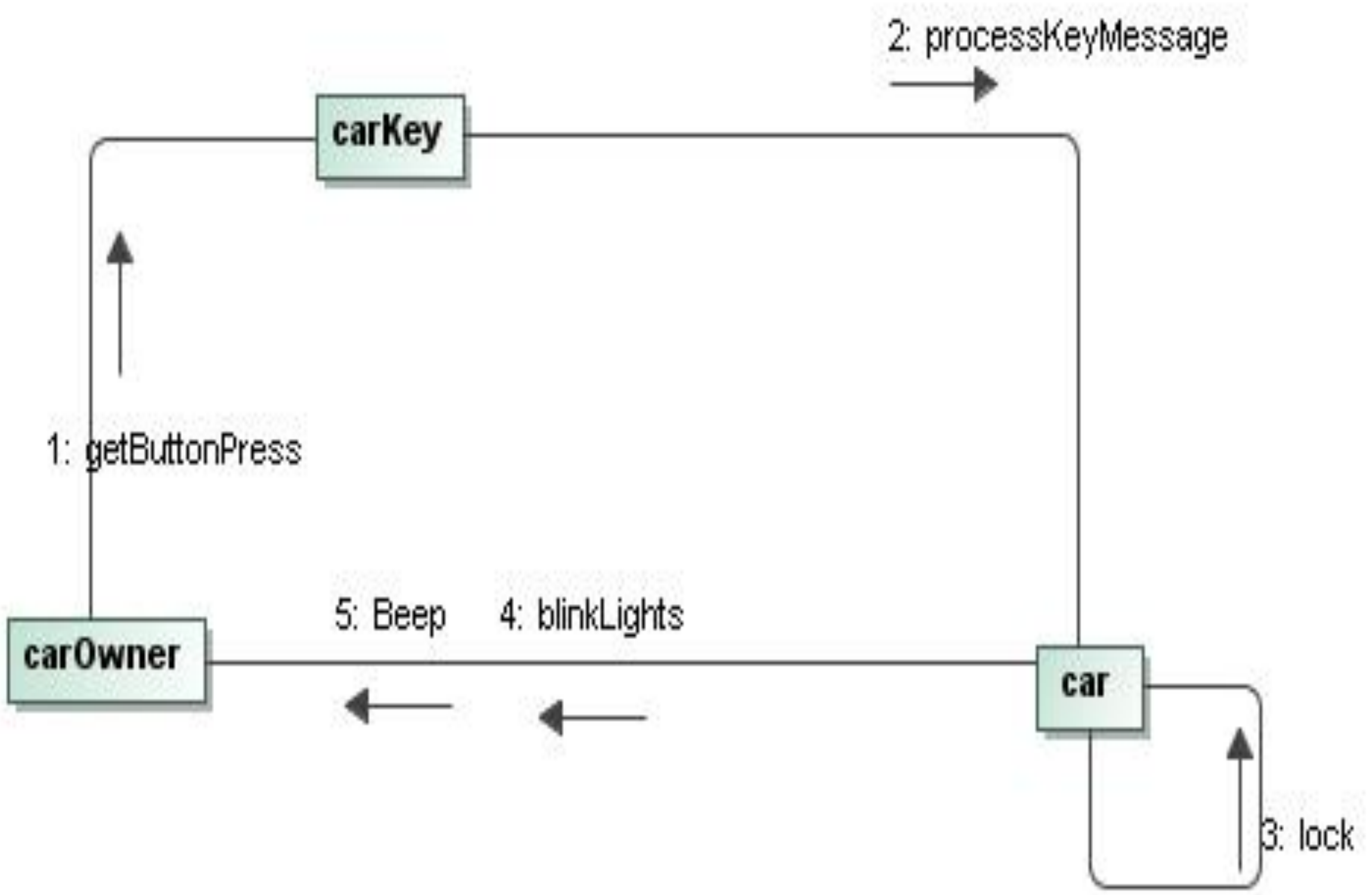
An Example of Collaboration Diagram



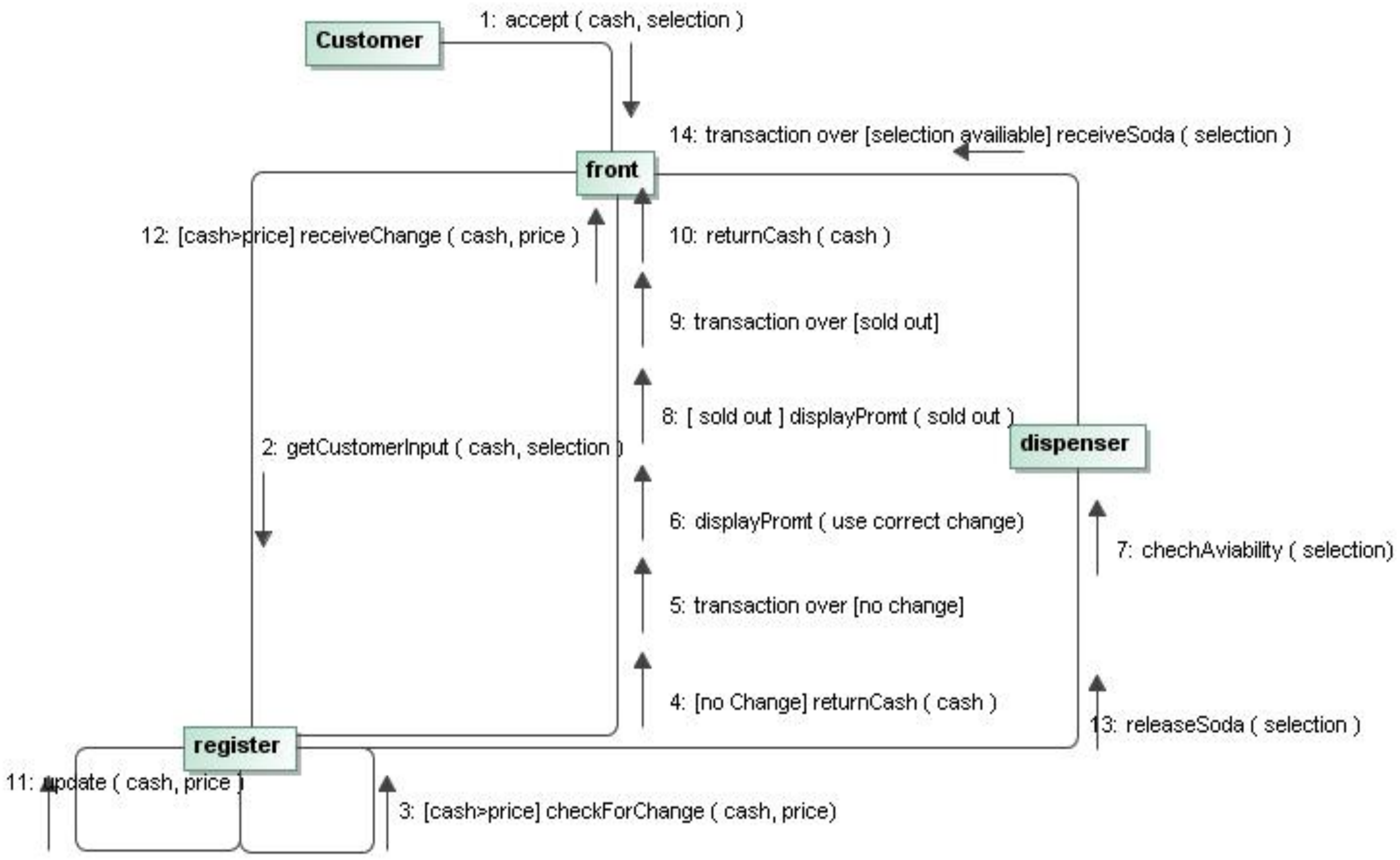
Another Example of Collaboration Diagram


1: requestToRegistrar (a student)
GUI
couseSection
2: prereq := getPreRequisite
course
6: addToRegistrationList
student
registration
3: hasPrerequisite := hasPassedCourse (prereq)
4: create
5: addToSchedule

Communication Diagram for Car



Communication Diagram of Soda Machine


Customer
1: accept (cash, selection)
14: transaction over [selection availiable] receiveSoda (selection)
front
12: [cash>price] receiveChange (cash, price)
10: returnCash (cash)
9: transaction over [sold out]
8: [sold out] displayPromt (sold out)
dispenser
2: getCustomerInput (cash, selection)
6: displayPromt (use correct change)
7: chechAviability (selection)
5: transaction over [no change]
4: [no Change] returnCash (cash)
13: releaseSoda (selection)
register
11: update (cash, price)
3: [cash>price] checkForChange (cash, price)

Communication links

- A communication link can exist between two objects whenever it is possible for one object to send a message to the other one.

- Several situations can make this message exchange possible:

1. The classes of the two objects have an *association* between them.
    - This is the most common case.
    - If all messages are sent in the same direction, then probably the association can be made unidirectional.

Other communication links

2. The receiving object is stored in a *local* variable of the sending method.
   - This often happens when the object is created in the sending method or when some computation returns an object .
   - The stereotype to be used is «local» or [L].

3. A reference to the receiving object has been received as a *parameter* of the sending method.
   - The stereotype is «parameter» or [P].

Other communication links

4. The receiving object is global.

- This is the case when a reference to an object can be obtained using a static method.
- The stereotype «global», or a [G] symbol is used in this case.

5. The objects communicate over a network.

- We suggest to write «network».

How to choose between using a sequence or communication diagram

- Sequence diagrams
  - Make explicit the time ordering of the interaction.
    - Use cases make time ordering explicit too
    - So sequence diagrams are a natural choice when you build an interaction model from a use case.
  - Make it easy to add details to messages.
    - Communication diagrams have less space for this

How to choose between using a Sequence or communication diagram

- Communication diagrams
  - Can be seen as a projection of the class diagram
    - Might be preferred when you are *deriving* an interaction diagram from a class diagram.
    - Are also useful for *validating* class diagrams.

This image cannot currently be displayed.

Activity Diagram

Activity Diagram

- Not present in earlier modelling techniques, possibly based on event diagram of **Odell** [1992].
- Represents processing activity, may not correspond to methods.
- Activity is a state with an internal action and one/many outgoing transitions.
- Somewhat related to flowcharts.

Activity Diagram vs Flow Chart

- Can represent parallel activity and synchronization aspects
- Swim lanes can be used to group activities based on who is performing them
- Example: academic department vs. hostel

Activity Diagrams

- An activity diagram
  - Can be used to understand the flow of work that an object or component performs.
  - Can also be used to visualize the interrelation and interaction between different use cases.
  - Is most often associated with several classes.

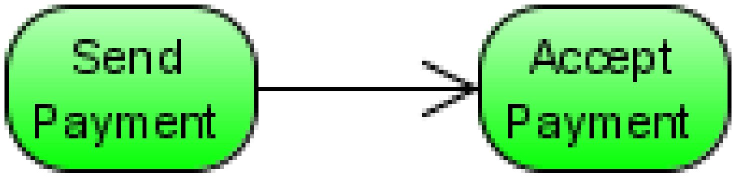
- One of the strengths of activity diagrams is the representation of *concurrent* activities.

Activity Diagram

- Normally employed in business process modeling.
- Carried out during requirements analysis and specification stage.
- Can be used to develop interaction diagrams.

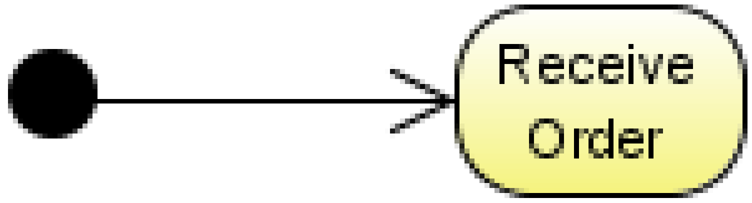
Action

- An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action.



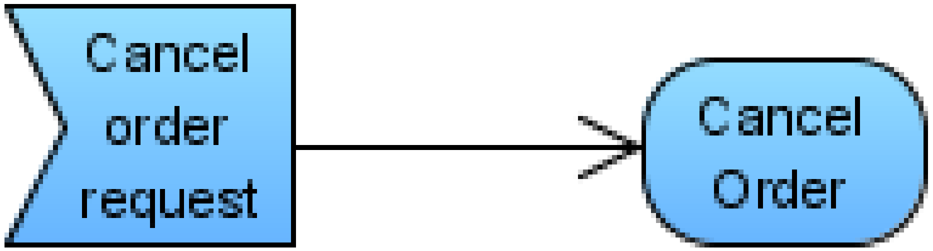
InitialNode

- An activity may have more than one initial node.



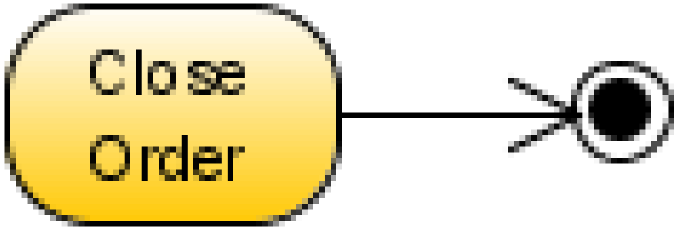
AcceptEventAction

- AcceptEventAction is an action that waits for the occurrence of an event meeting specified conditions.



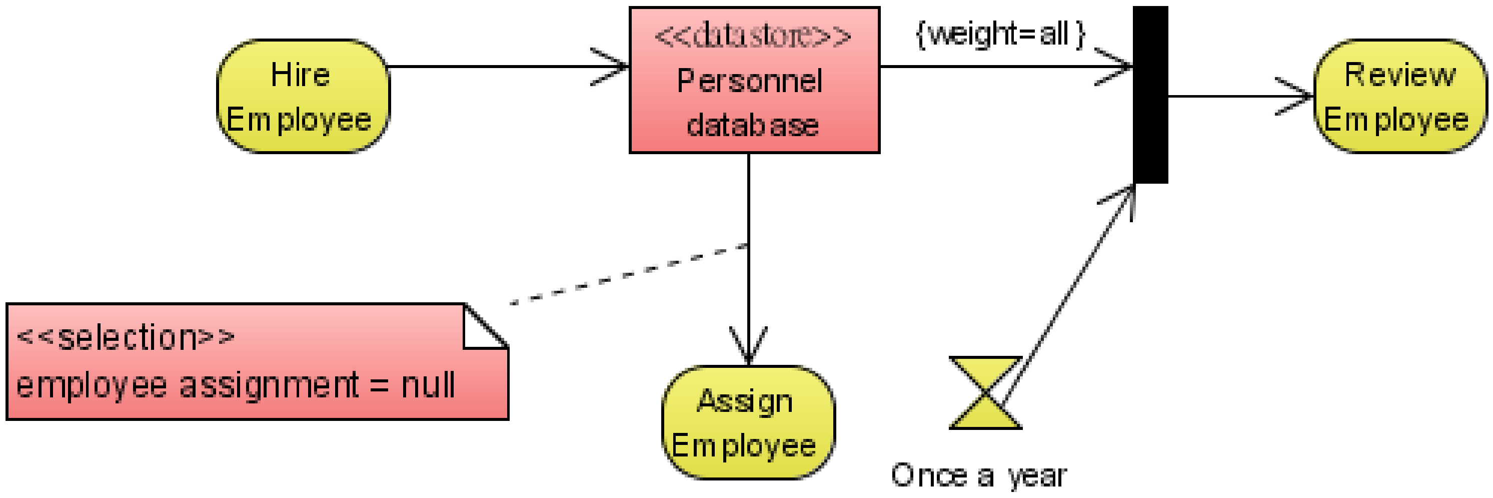
ActivityFinal

- An activity may have more than one activity as final node. The first one reached stops all flows in the activity.



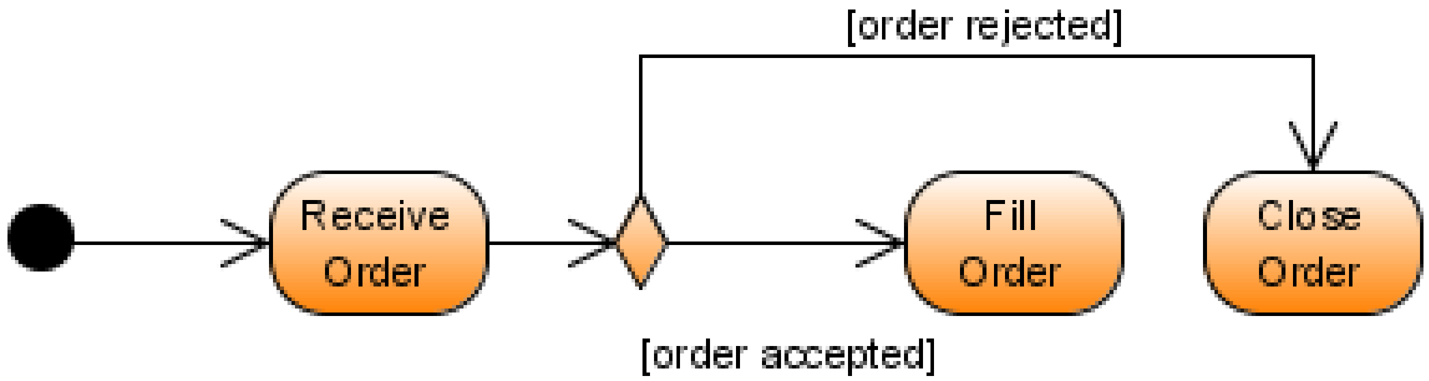
DataStore

- A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object.



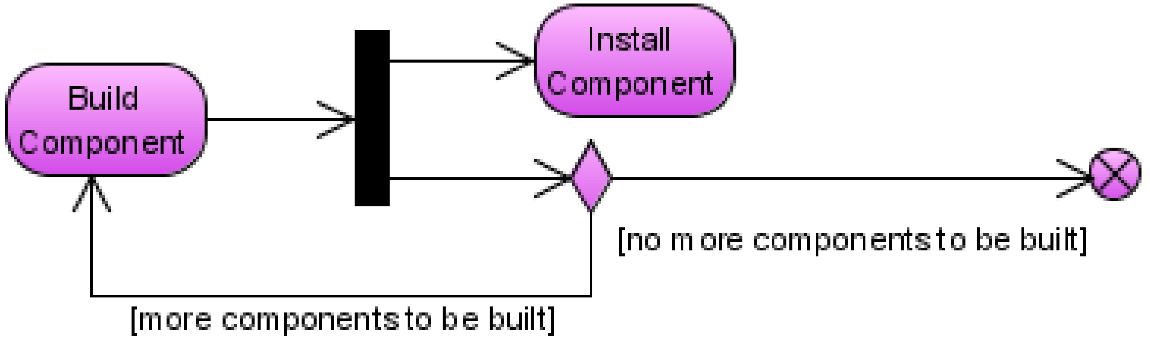
DecisionNode

- A decision node is a control node that chooses between outgoing flows.



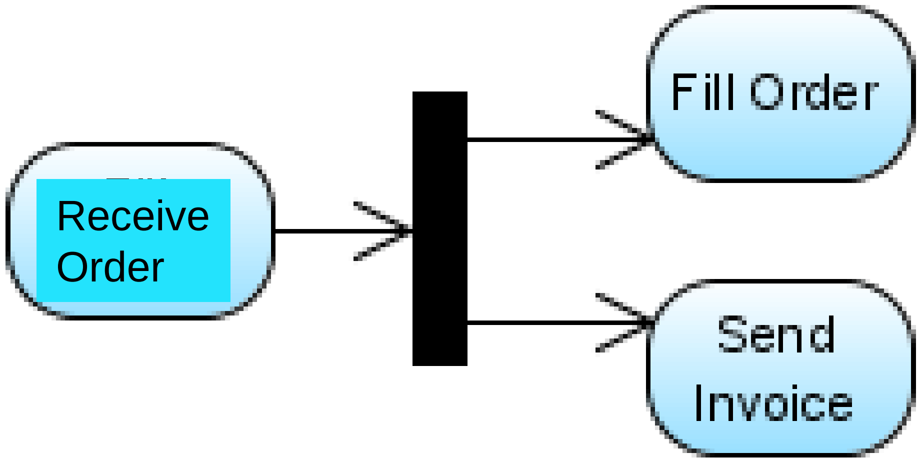
FlowFinal

- A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity.


Build Component
Install Component
[no more components to be built]
[more components to be built]

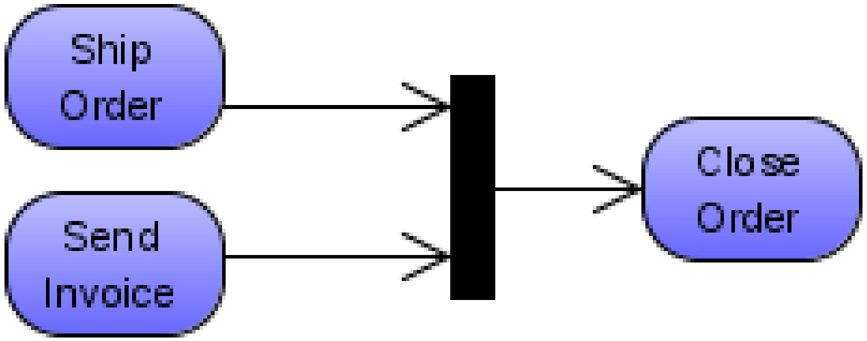
ForkNode

- A fork node has one incoming edge and multiple outgoing edges.



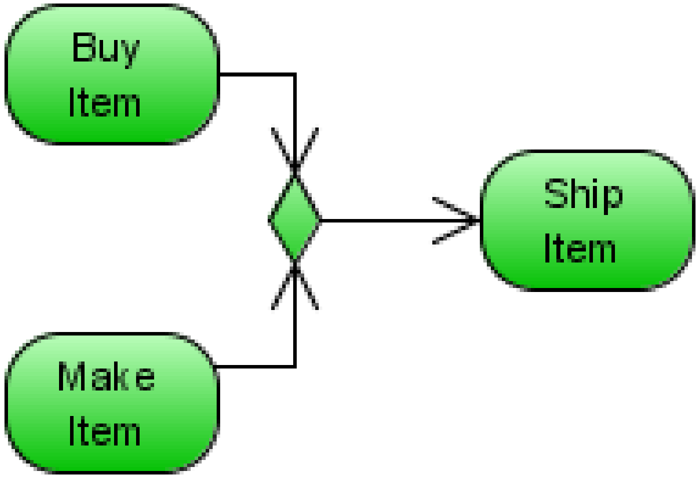
JoinNode

- A join node has multiple incoming edges and one outgoing edge



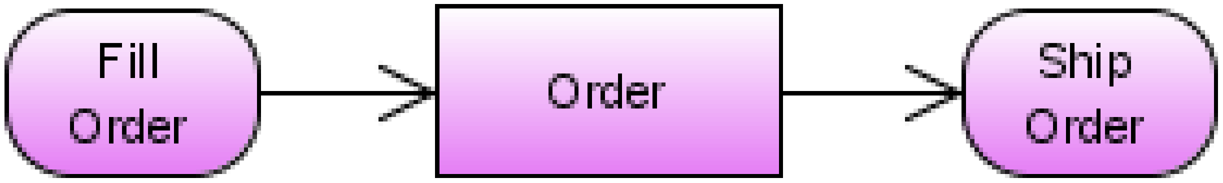
MergeNode

- A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.



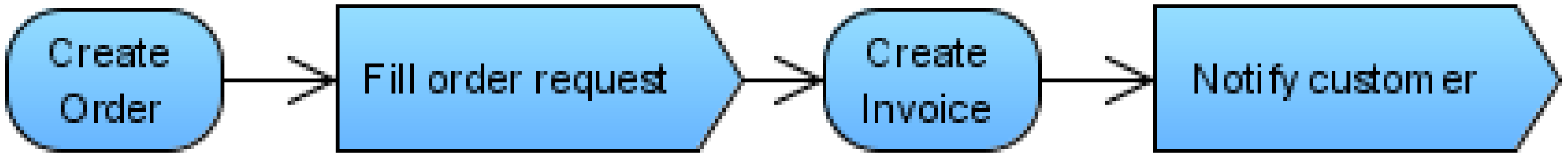
ObjectNode

- An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to, as described in the semantics section.



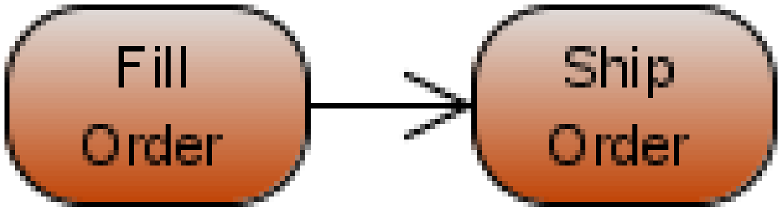
SendSignalAction

- SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity



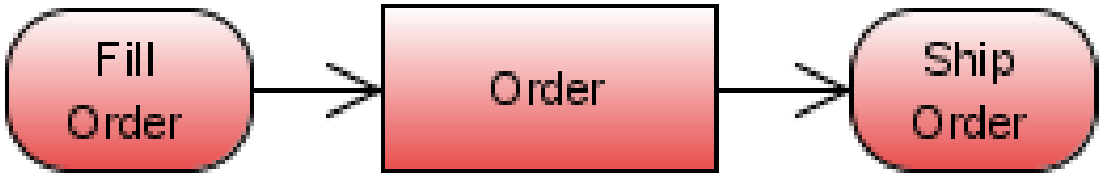
ControlFlow

- Objects and data cannot pass along a control flow edge.



ObjectFlow

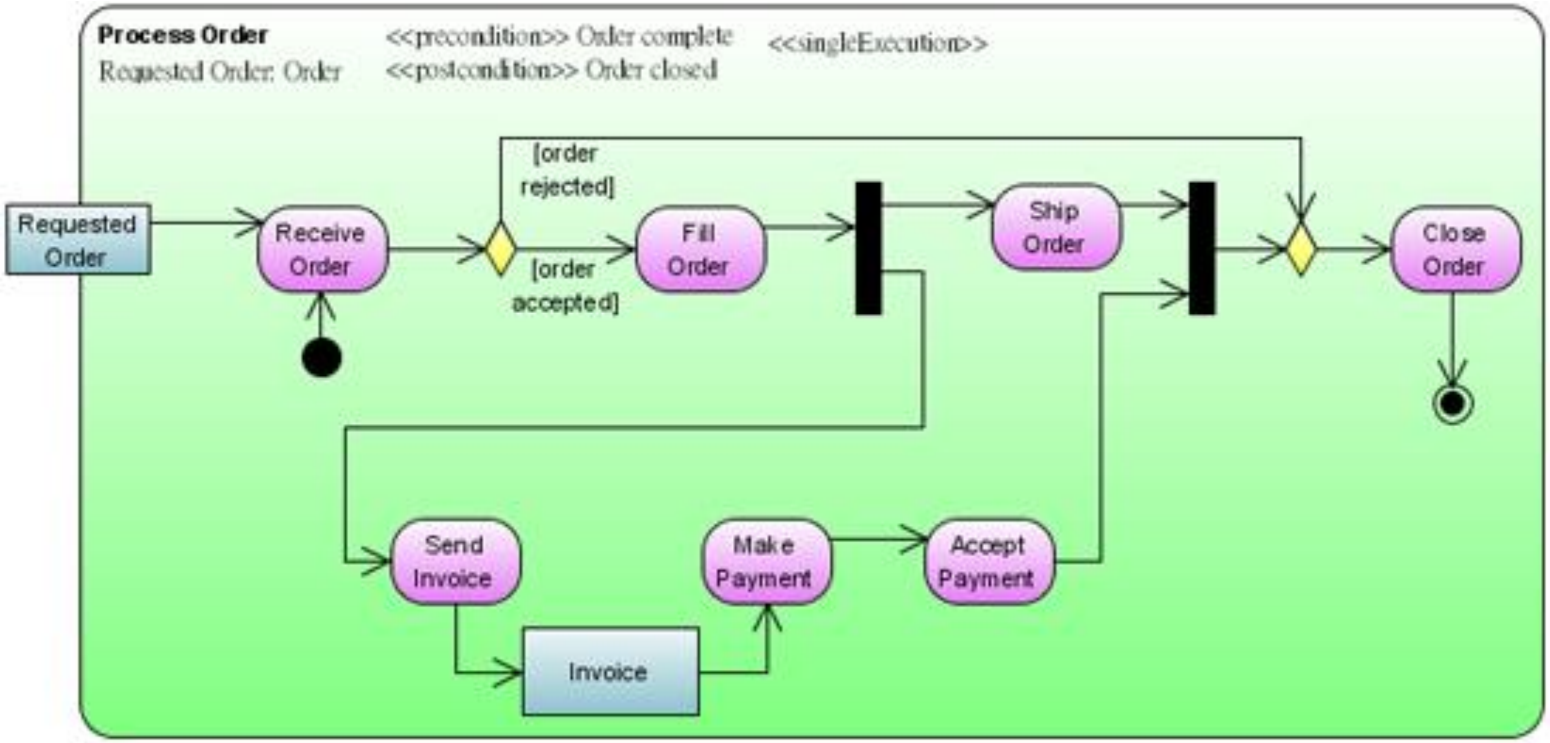
- An object flow models the flow of values to or from object nodes



Activity

- An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. The subordinate behaviors coordinated by these models may be initiated because other behaviors in the model finish executing, because objects and data become available, or because events occur external to the flow.

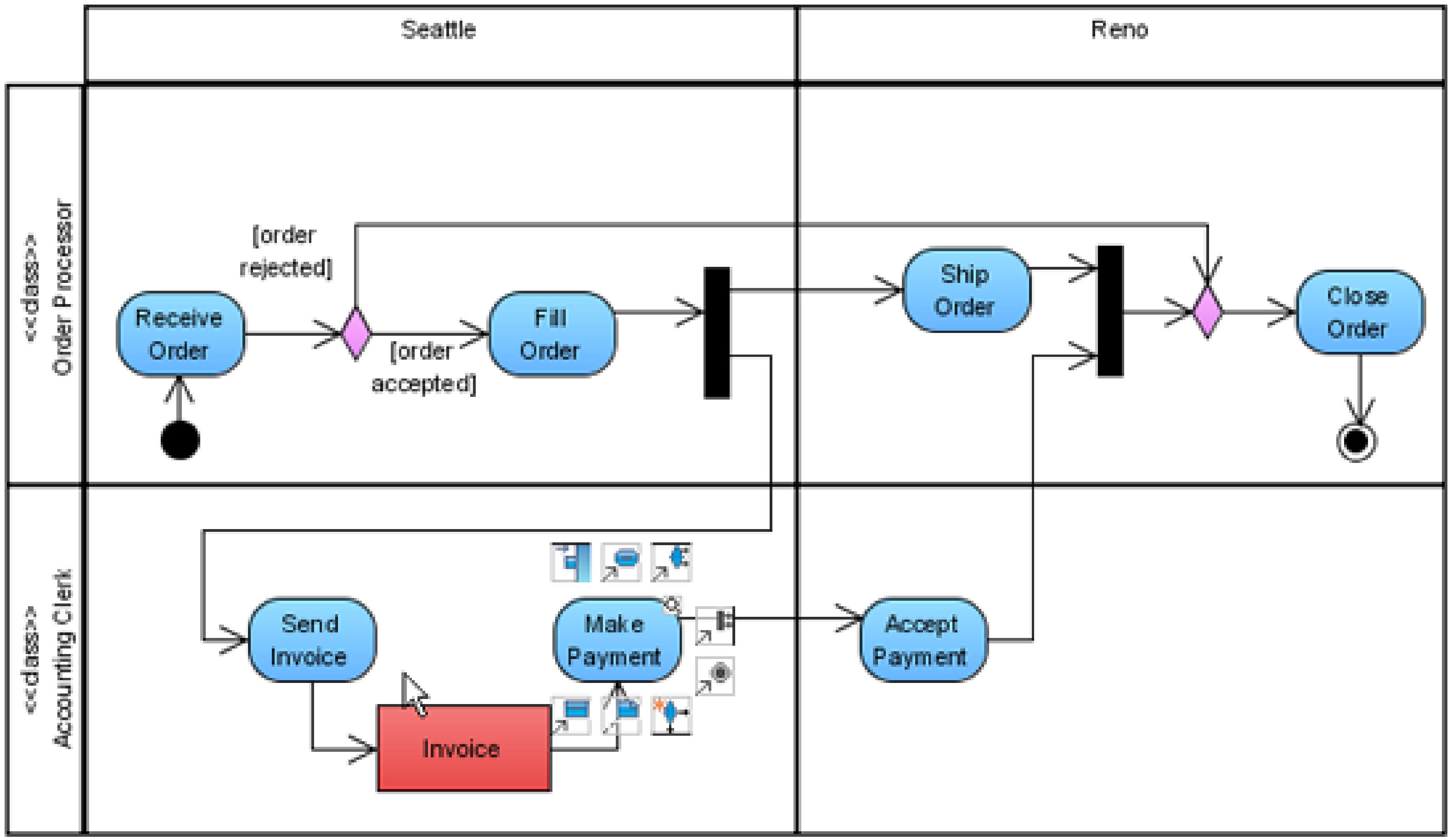
Activity


Process Order
Requested Order: Order
<<precondition>> Order complete
<<postcondition>> Order closed
<<singleExecution>>
Requested Order
Receive Order
[order rejected]
[order accepted]
Fill Order
Ship Order
Close Order
Send Invoice
Invoice
Make Payment
Accept Payment

ActivityPartition

- Partitions divide the nodes and edges to constrain and show a view of the contained nodes. Partitions can share contents. They often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the nodes of an activity.

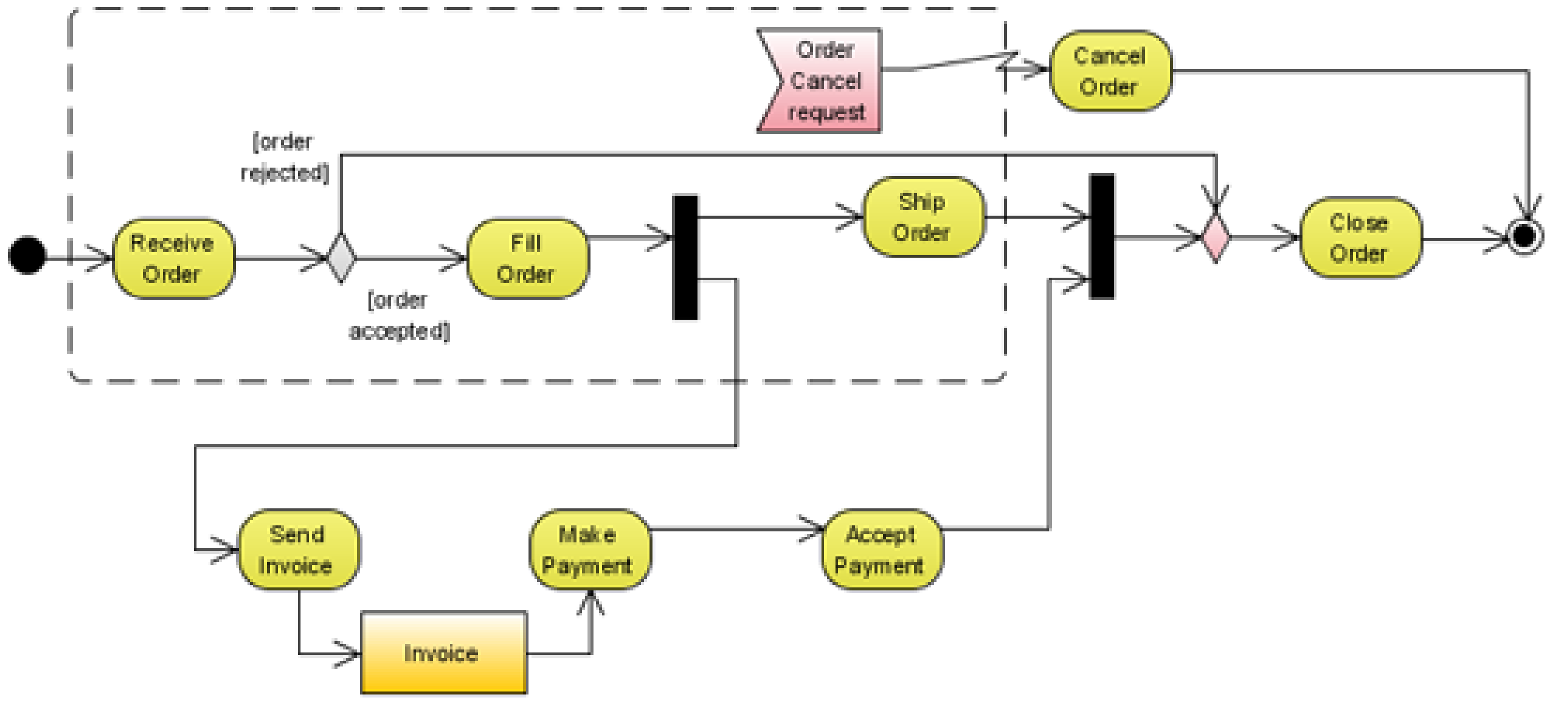
ActivityPartition



InterruptibleActivityRegion

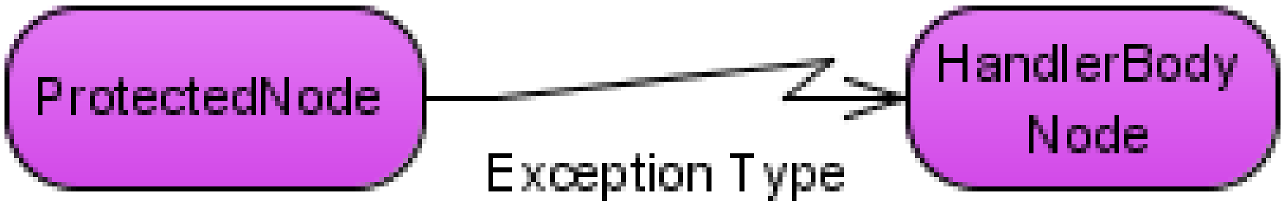
- An interruptible region contains activity nodes. When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated.

InterruptibleActivityRegion



ExceptionHandler

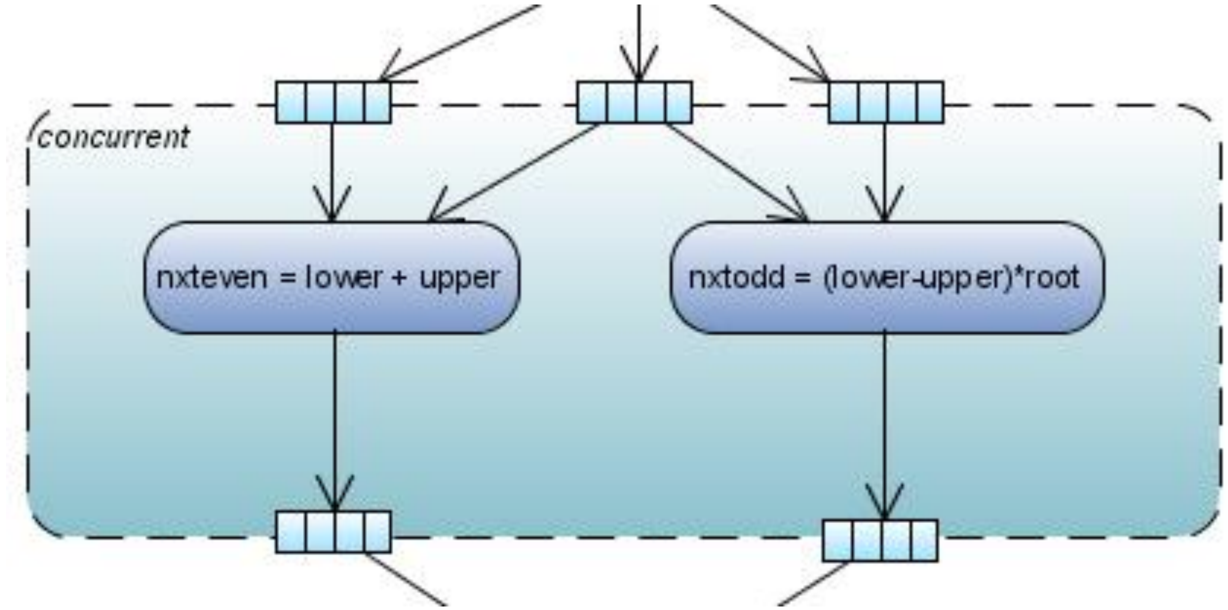
- An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node



ExpansionRegion

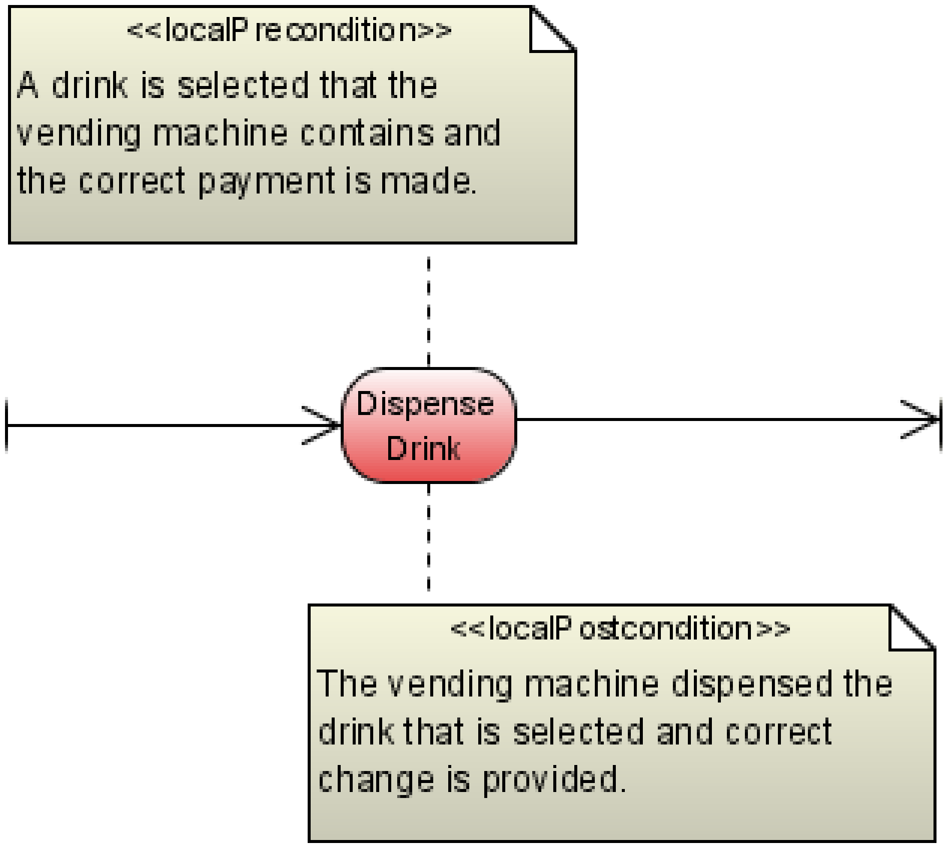
- An expansion region is a strictly nested region of an activity with explicit input and outputs (modeled as ExpansionNodes). Each input is a collection of values. If there are multiple input pins, each of them must hold the same kind of collection, although the types of the elements in the different collections may vary. The expansion region is executed once for each element (or position) in the input collection.

ExpansionRegion



Local pre- and post conditions

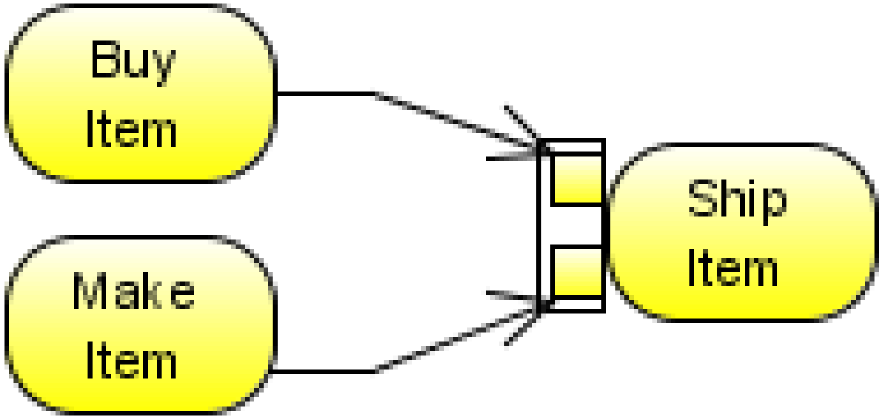
- In CompleteActivities, action is extended to have pre- and postconditions



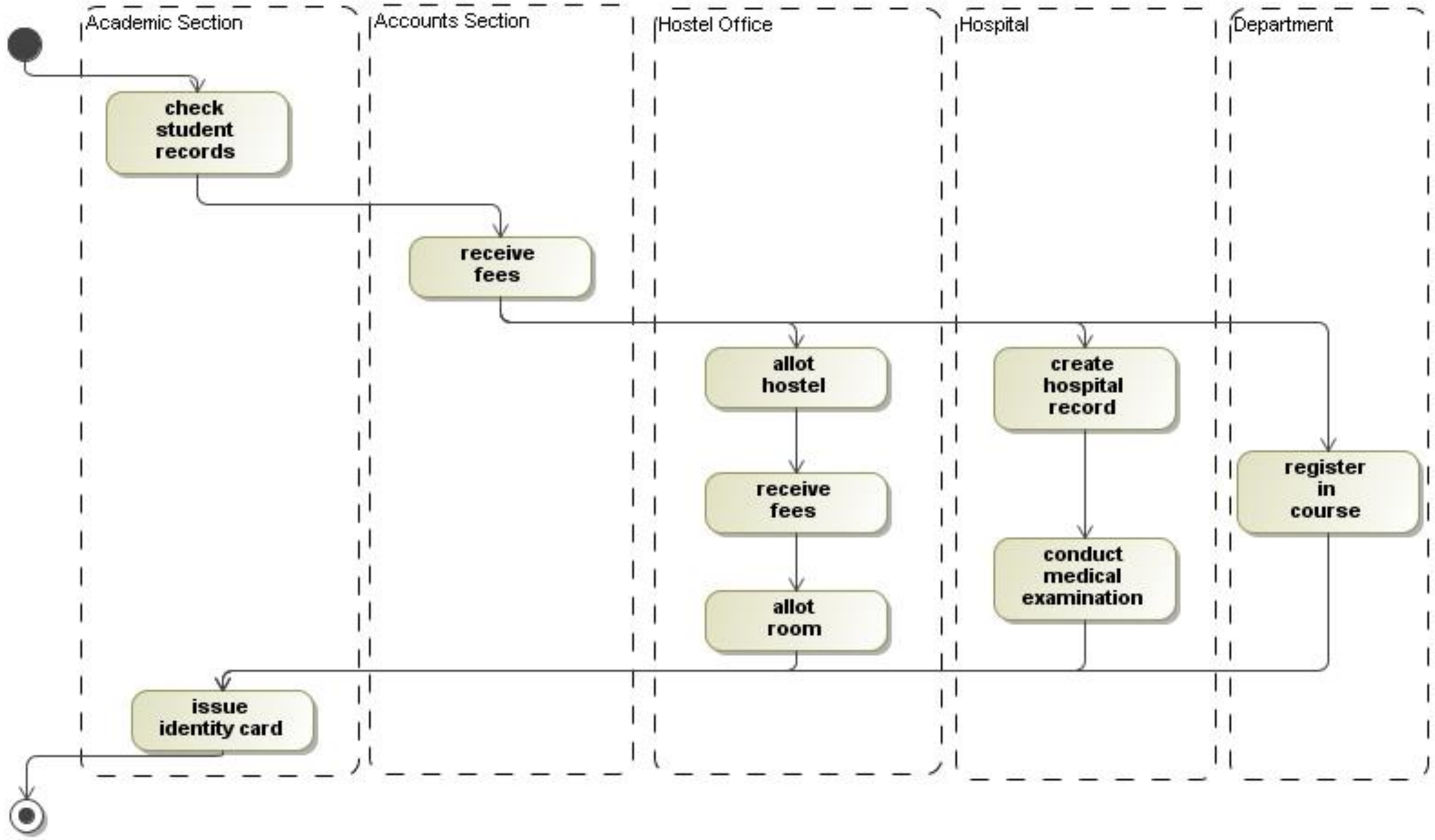
<<localPrecondition>>
A drink is selected that the vending machine contains and the correct payment is made.
Dispense Drink
<<localPostcondition>>
The vending machine dispensed the drink that is selected and correct change is provided.

ParameterSet

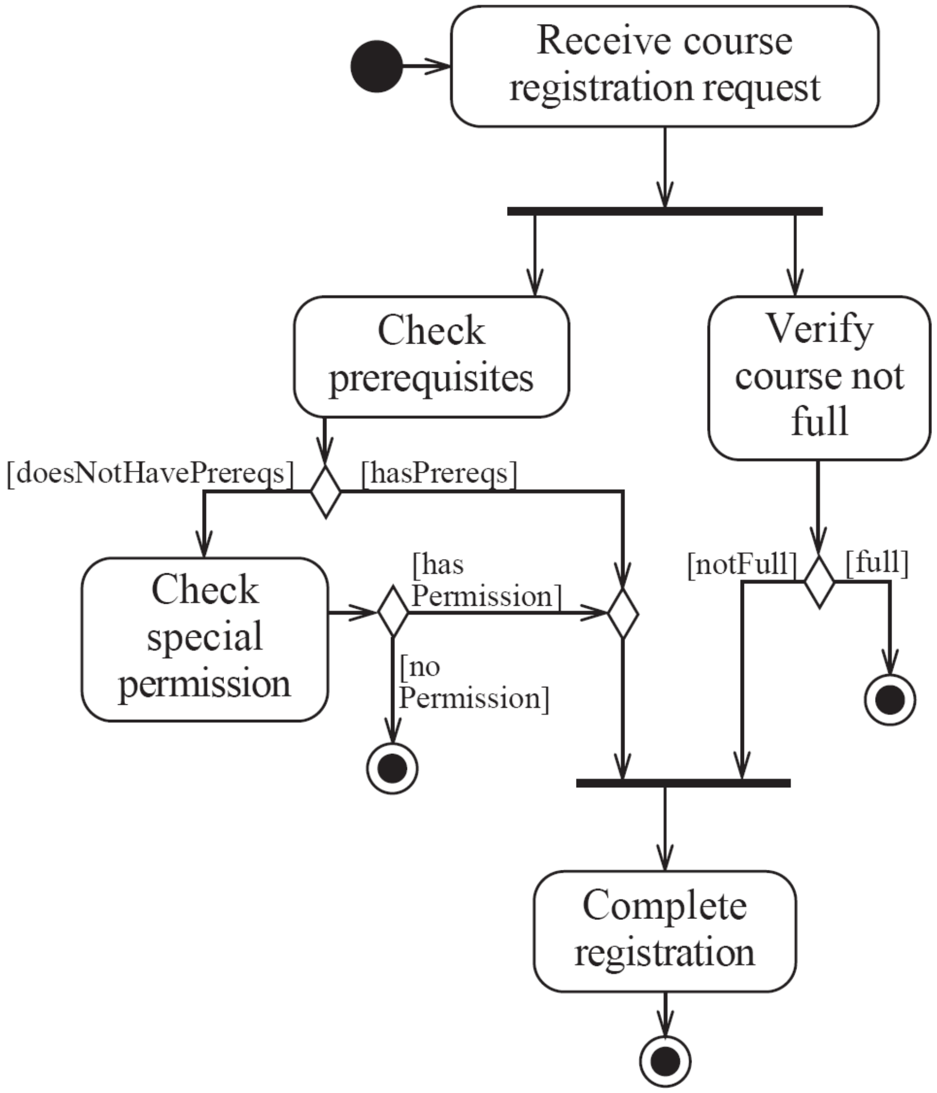
- An parameter set acts as a complete set of inputs and outputs to a behavior, exclusive of other parameter sets on the behavior.



An Example of
An Activity Diagram



Activity diagrams – an example



Representing concurrency

- Concurrency is shown using forks, joins and rendezvous.
  - A *fork* has one incoming transition and multiple outgoing transitions.
    - The execution splits into two concurrent threads.
  - A *rendezvous* has multiple incoming and multiple outgoing transitions.
    - Once all the incoming transitions occur all the outgoing transitions may occur.

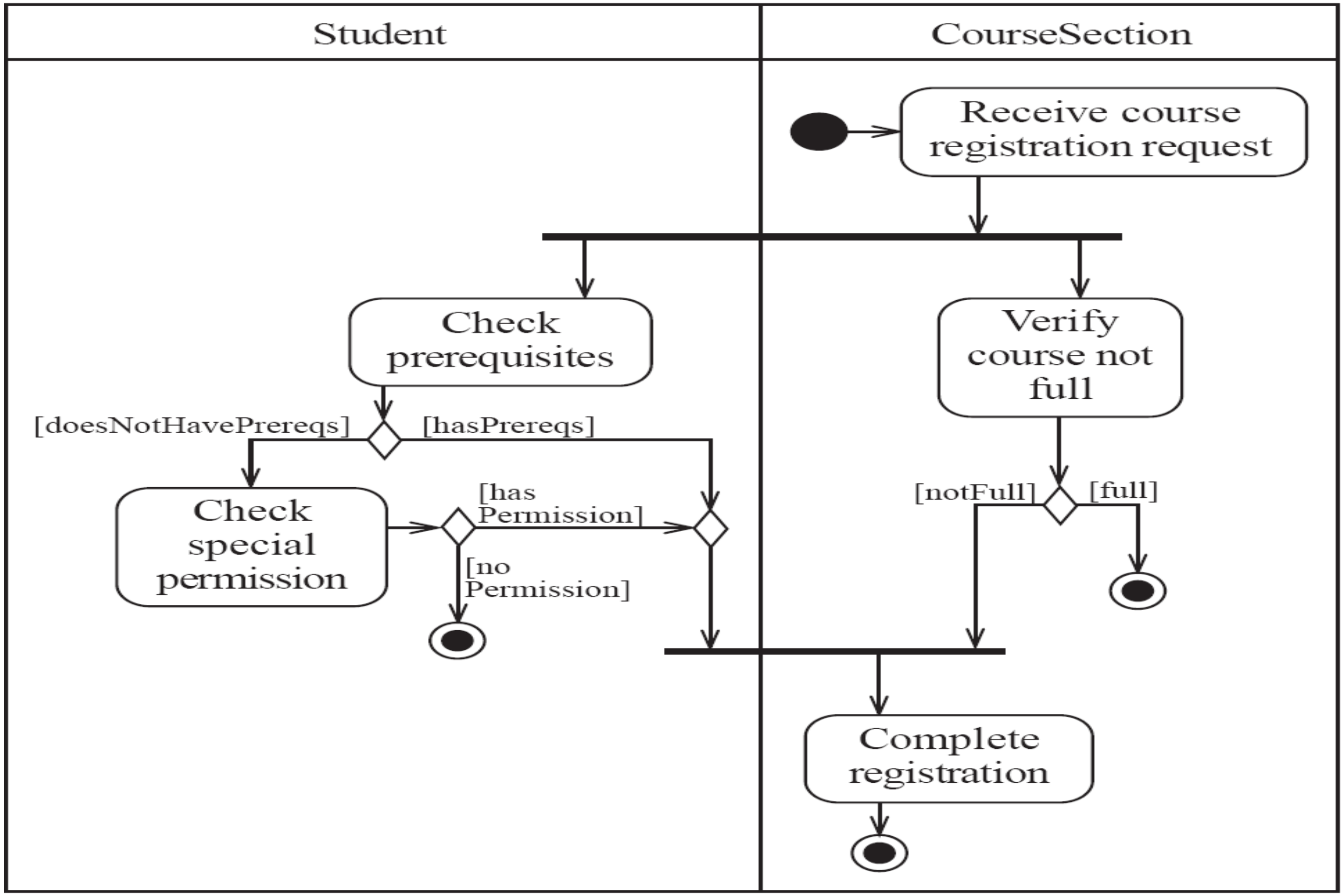
Representing concurrency

- A *join* has **multiple** incoming transitions and **one** outgoing transition.
  - The outgoing transition will be taken when all incoming transitions have occurred.
  - The incoming transitions must be triggered in separate threads.
  - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

Swimlanes

- Activity diagrams are most often associated with several classes.
  - The partition of activities among the existing classes can be explicitly shown using *swimlanes*.

Activity diagrams – an example with swimlanes



This image cannot currently be displayed.

Thank you

Questions?