WIA2002: Software Modelling
Semester 1, Session 2016/17

Lecture 10: Specifying Control –

UML State Machine Diagrams

Learning Objectives

- Understand the purposes of state machine diagrams.
- Know how to model object life cycles using state machines.
- Know how to develop state machine diagrams using two approaches (behavioural and life cycle).
- Know how to model concurrent behaviour in an object.
- Know how to model protocol state machine diagrams.
- Know how to ensure consistency with other UML models.

State Machine Diagrams

- Some of the classes in the class diagrams are quite dynamic in that they **pass through a variety of states** over the course of their existence.
  - E.g. A vehicle can change over time from being “new” to “pre-owned,” on the basis of its status with the dealership
- State Machine Diagram is a dynamic model that shows the different states that a single class passes through during its life in response to events.
- Not used for all classes, but just to further define complex classes

Purposes of State Machine Diagrams

- To identify requirements for control in an application.
- To show the different states of the class and what events cause the class to change from one state to another.

- In comparison to the sequence diagrams:
  - **Behavioural State Machine Diagrams** should be used if you are interested in understanding the <u>dynamic aspects of a single class</u> and how its instances evolve over time
  - **Sequence diagrams** should be used if you are interested on <u>how a particular use case scenario is executed over a set of classes</u>.

State

- The current state of an object is determined by the current value of the object's attributes and the links that it has with other objects.

- For example the class `StaffMember` has an attribute `startDate` which determines whether a `StaffMember` object is in the probationary state.

State

- A state describes a particular condition that a modelled element (e.g. object) may occupy for a period of time while it awaits some event or *trigger*.
- The possible states that an object can occupy are limited by its class.
- Objects of some classes have only one possible state.
- Conceptually, an object remains in a state for an interval of time.

State Machine

- The current state of a `GradeRate` object can be determined by the two attributes `rateStartDate` and `rateFinishDate`.

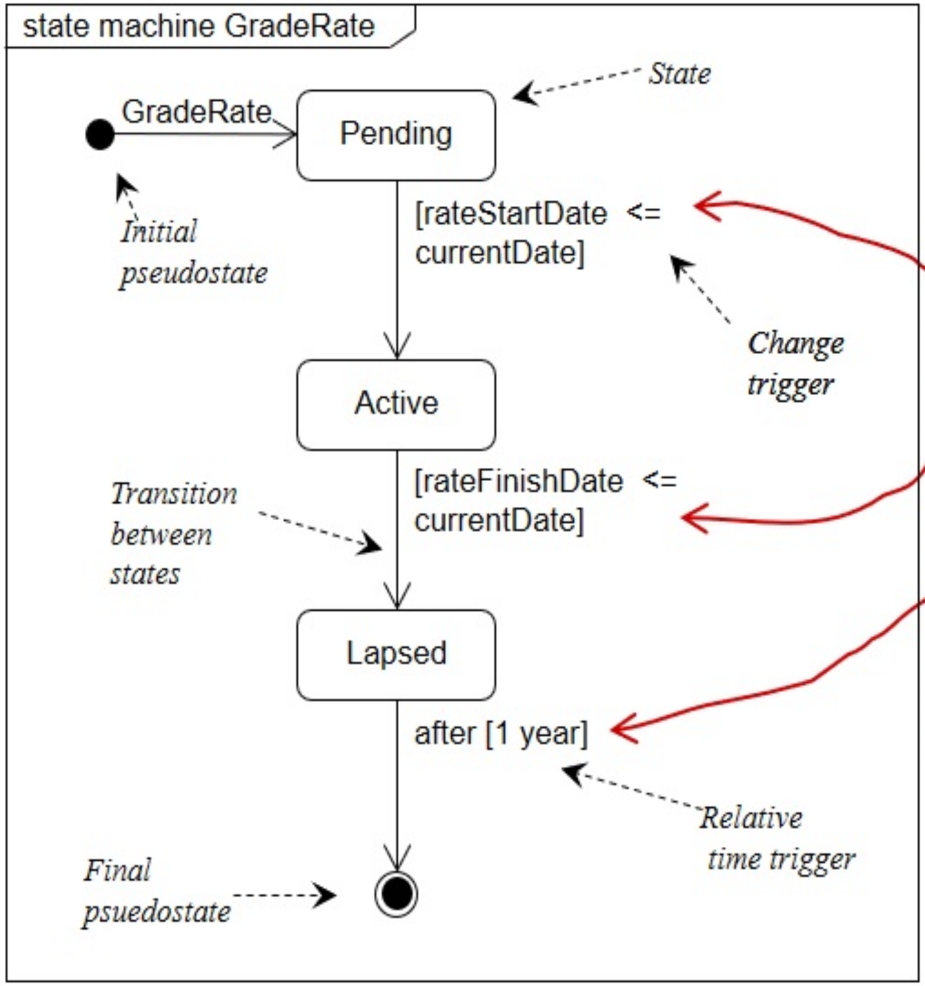
- An enumerated state variable may be used to hold the object state, possible values would be *Pending*, *Active* or *Lapsed*.

- Movement from one state to another state is called a transition, and is initiated by a trigger.

- A trigger is an event that can cause a state change.

State Machine Diagram (Notation)

state machine for the class `GradeRate`.



Movement from one state to another is dependent upon events that occur with the passage of time.

State Machine Diagram (Notation)

Term and Definition	Symbol
A state ▪ Is shown as a rectangle with rounded corners. ▪ Has a name that represents the state of an object.	
An initial state ▪ Is shown as a small filled-in circle. ▪ Represents the point at which an object begins to exist.	
A final state ▪ Is shown as a circle surrounding a small, solid filled-in circle (bull's-eye). ▪ Represents the completion of activity.	

State Machine Diagram (Notation)

An event ■ Is a noteworthy occurrence that triggers a change in state. ■ Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period. ■ Is used to label a transition.	Event name
A transition ■ Indicates that an object in the first state will enter the second state. ■ Is triggered by the occurrence of the event labeling the transition. ■ Is shown as a solid arrow from one state to another, labeled by the event name.	

Types of Trigger Event

- A *change trigger* occurs when a condition becomes true.
- A *call trigger* occurs when an object receives a call for one of its operations either from another object or from itself.
- A *signal trigger* occurs when an object receives a signal (an asynchronous communication).
- An *relative-time trigger* is caused by the passage of a designated period of time after a specified event (frequently the entry to the current state).

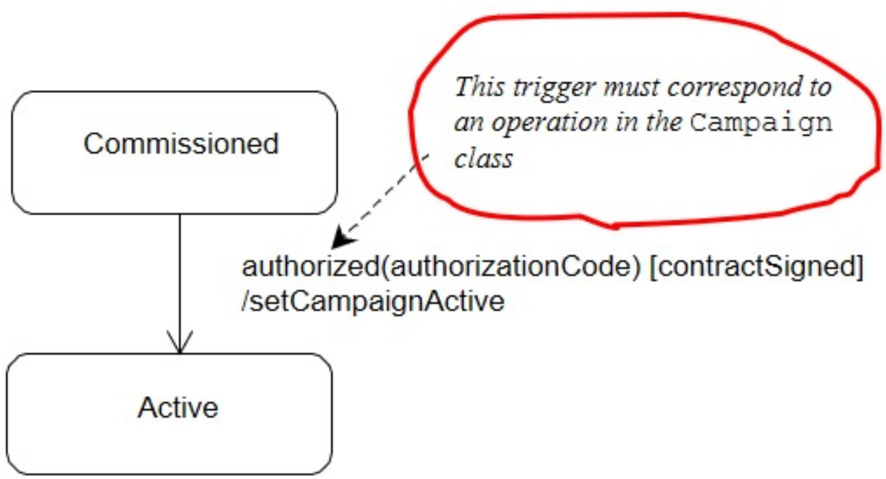
Events

For call and signal events, the format of the transition string is as follows:

`trigger-signature '[' constraint ']' '/' activity-expression`

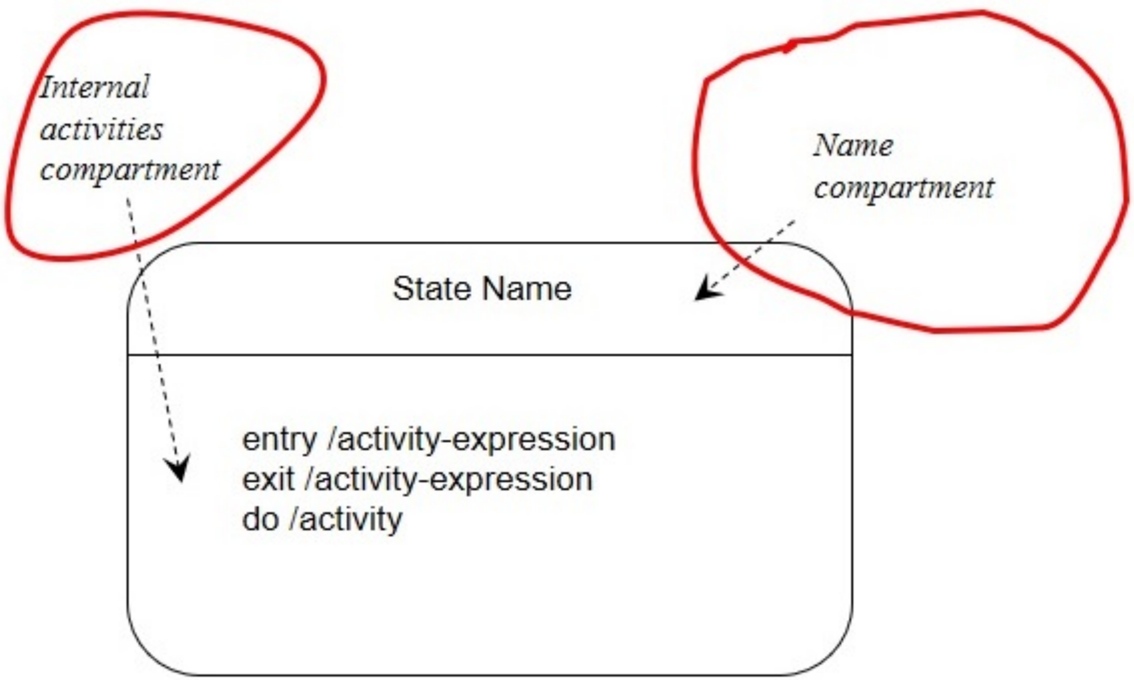
The trigger signature takes the following format:

`event-name '(' parameter-list ')'`

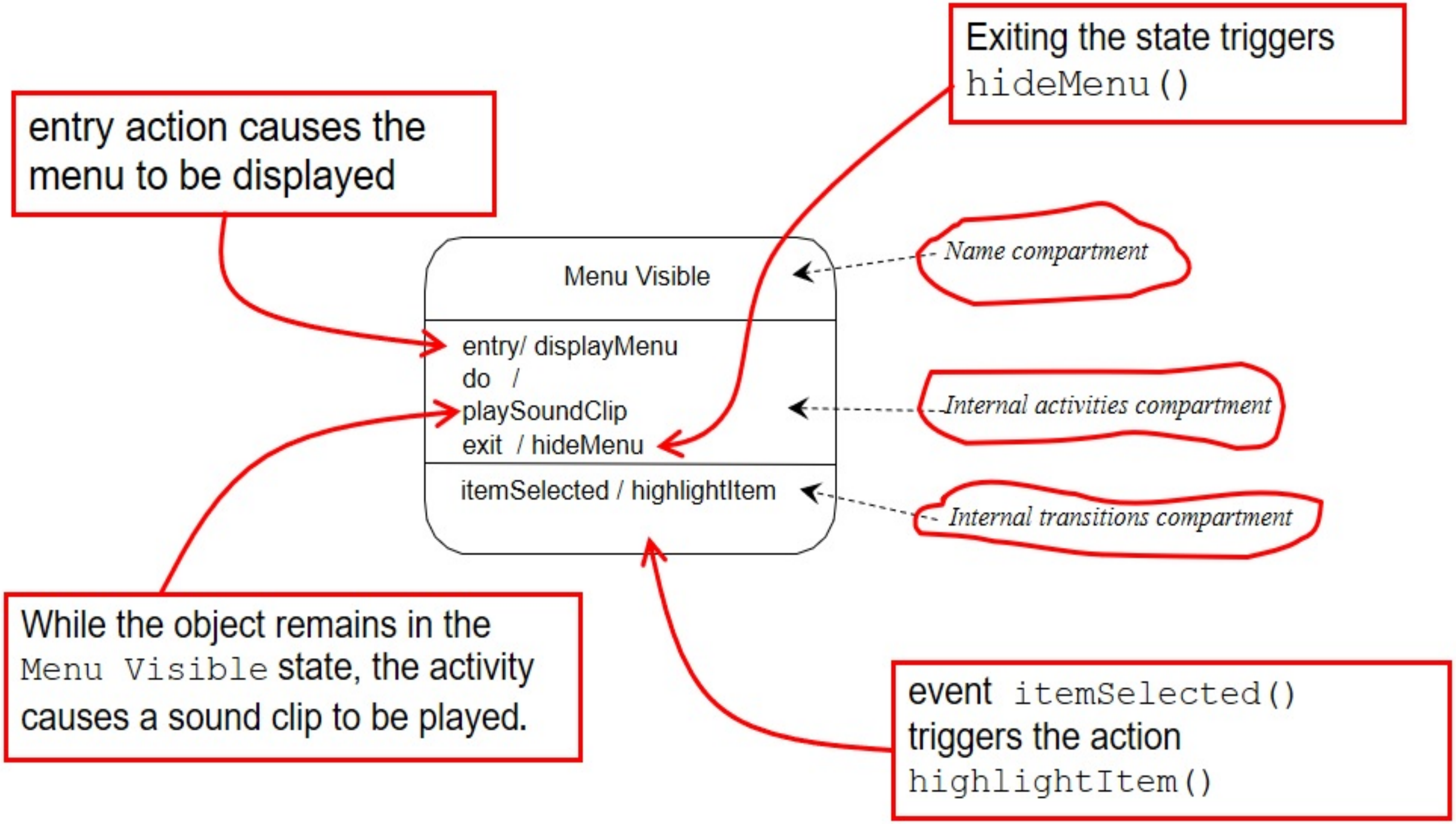


Internal Activities

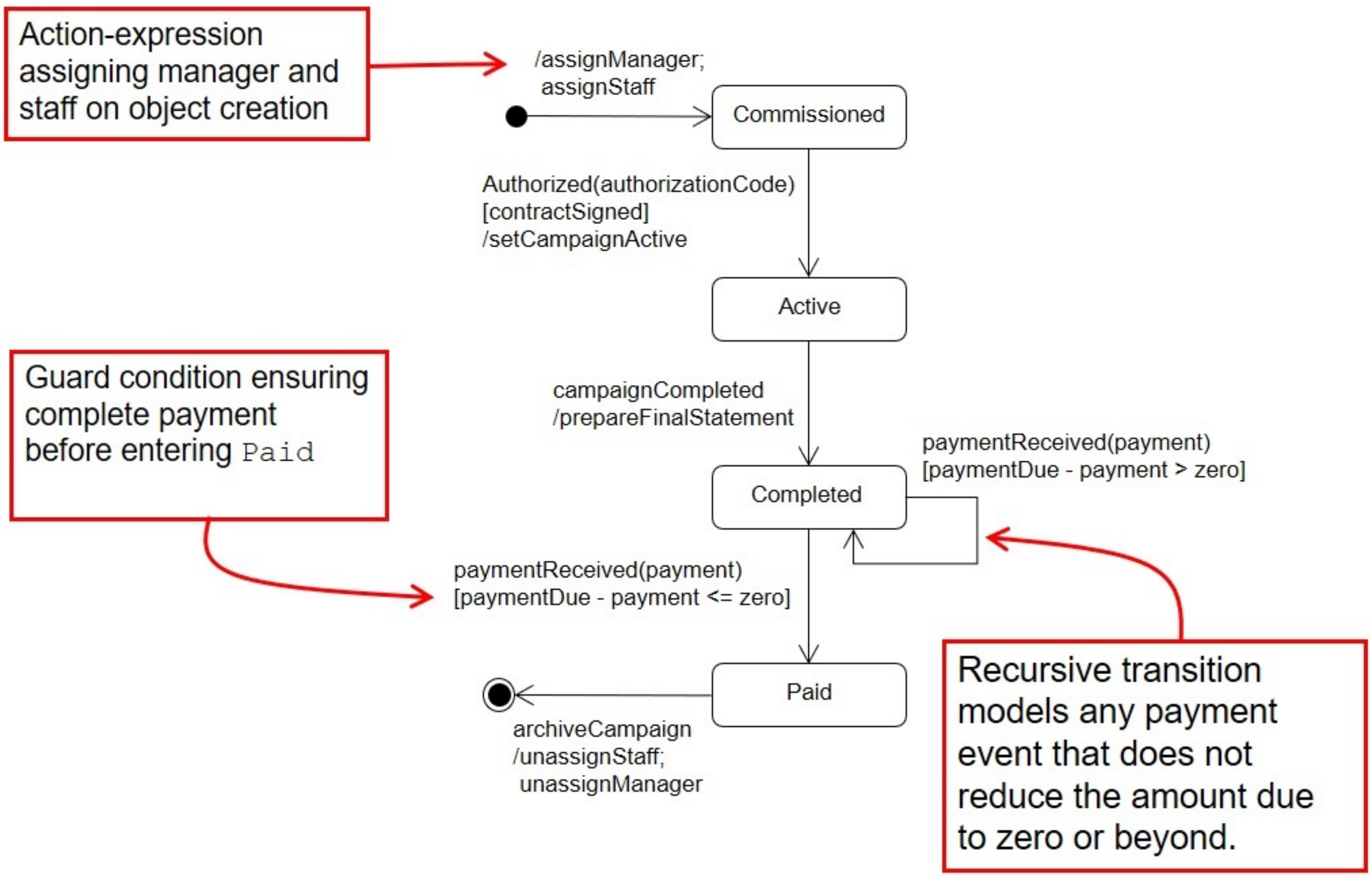
'entry' '/' activity name '(' parameter-list ')'
'exit' '/' activity name '(' parameter-list ')'
'do' '/' activity name '(' parameter-list ')'



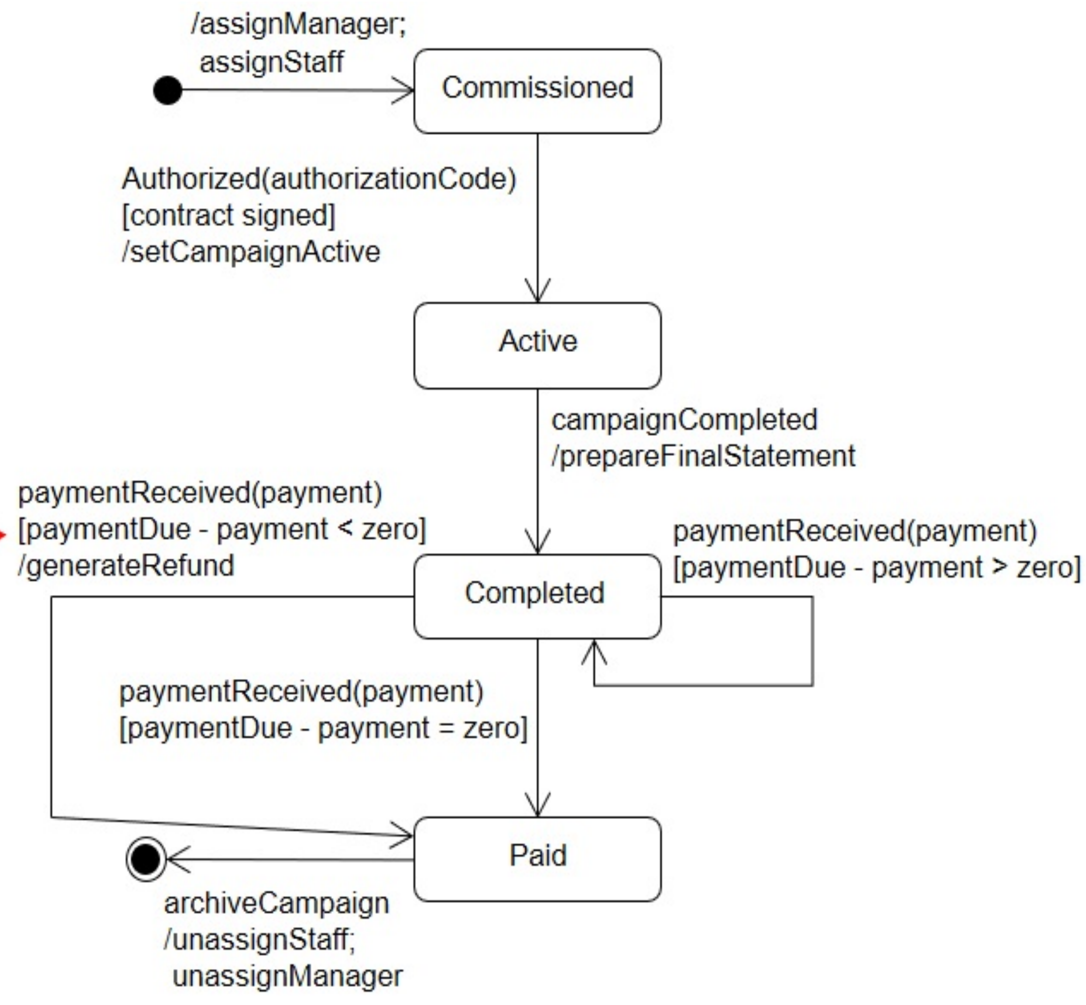
Example: ‘Menu Visible’ State of a DropDownMenu object



Example: State Machine Diagram for class `Campaign`



Example: A revised state machine for the class `Campaign`

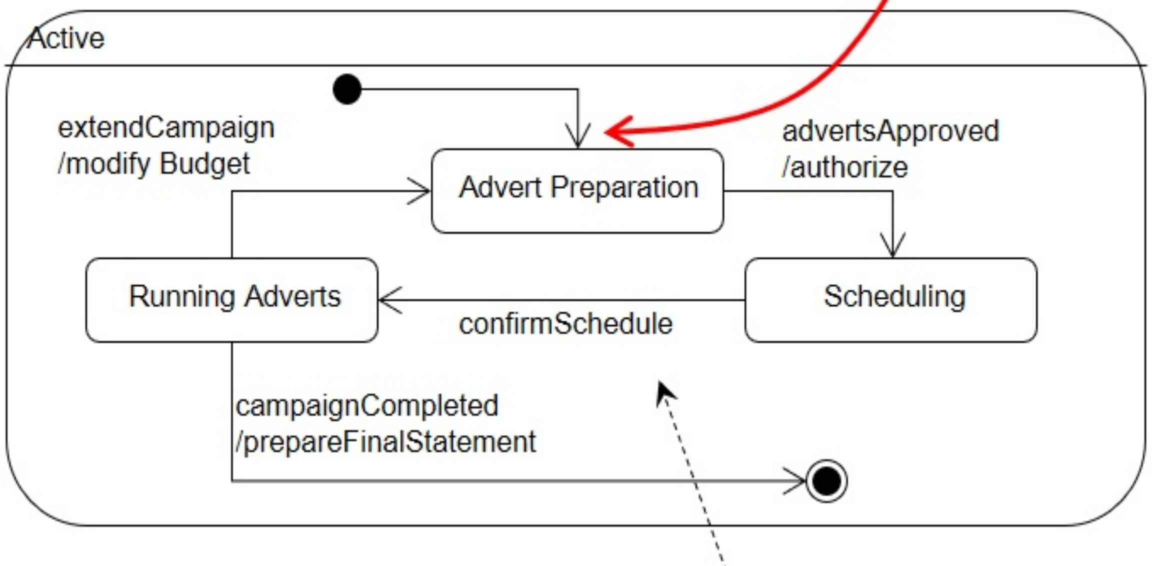


If the user requirements were to change, so that an overpayment is now to result in the automatic generation of a refund, a new transition is added.

Nested Substates

The `Active` *state of* `Campaign` *showing nested substates.*

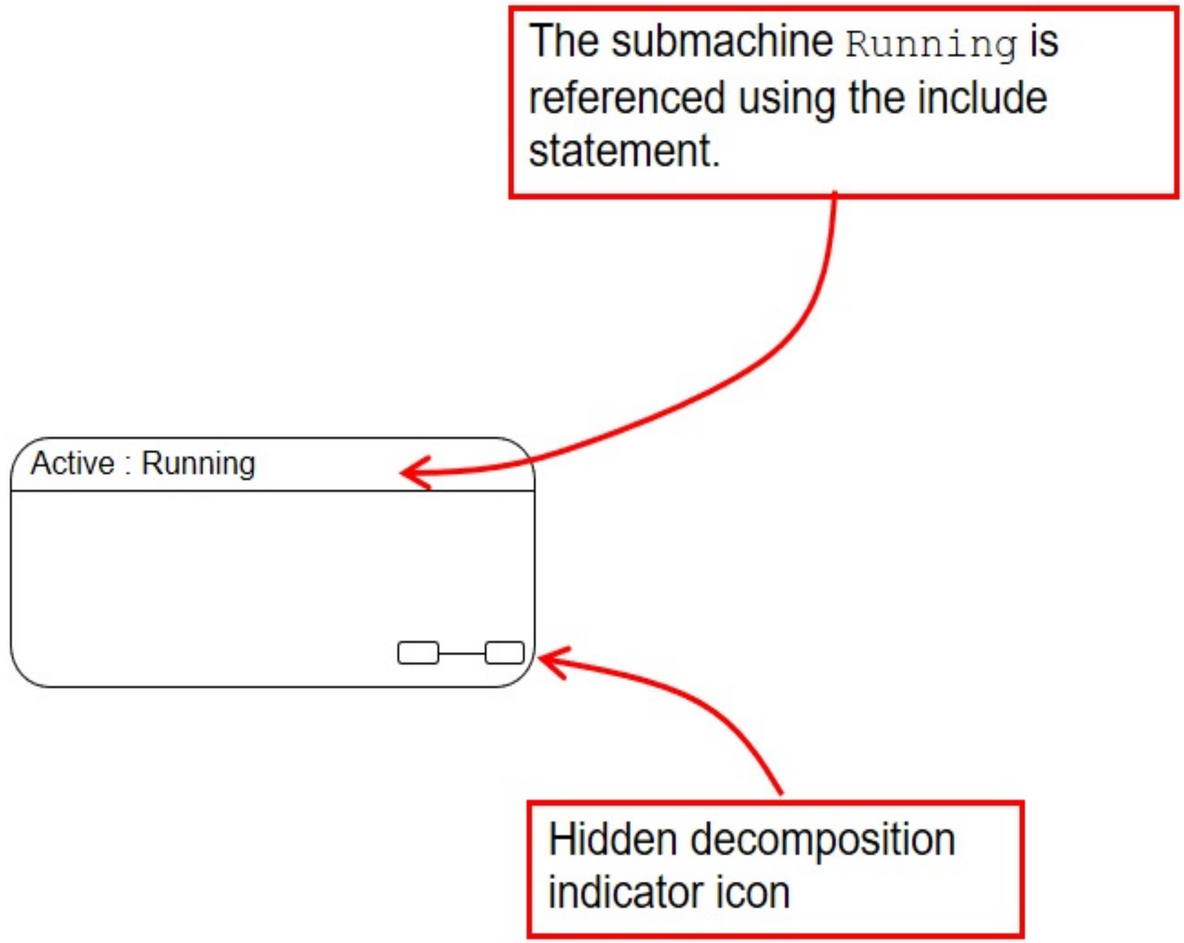
The transition from the initial pseudostate symbol should not be labelled with an event but may be labelled with an action, though it is not required in this example



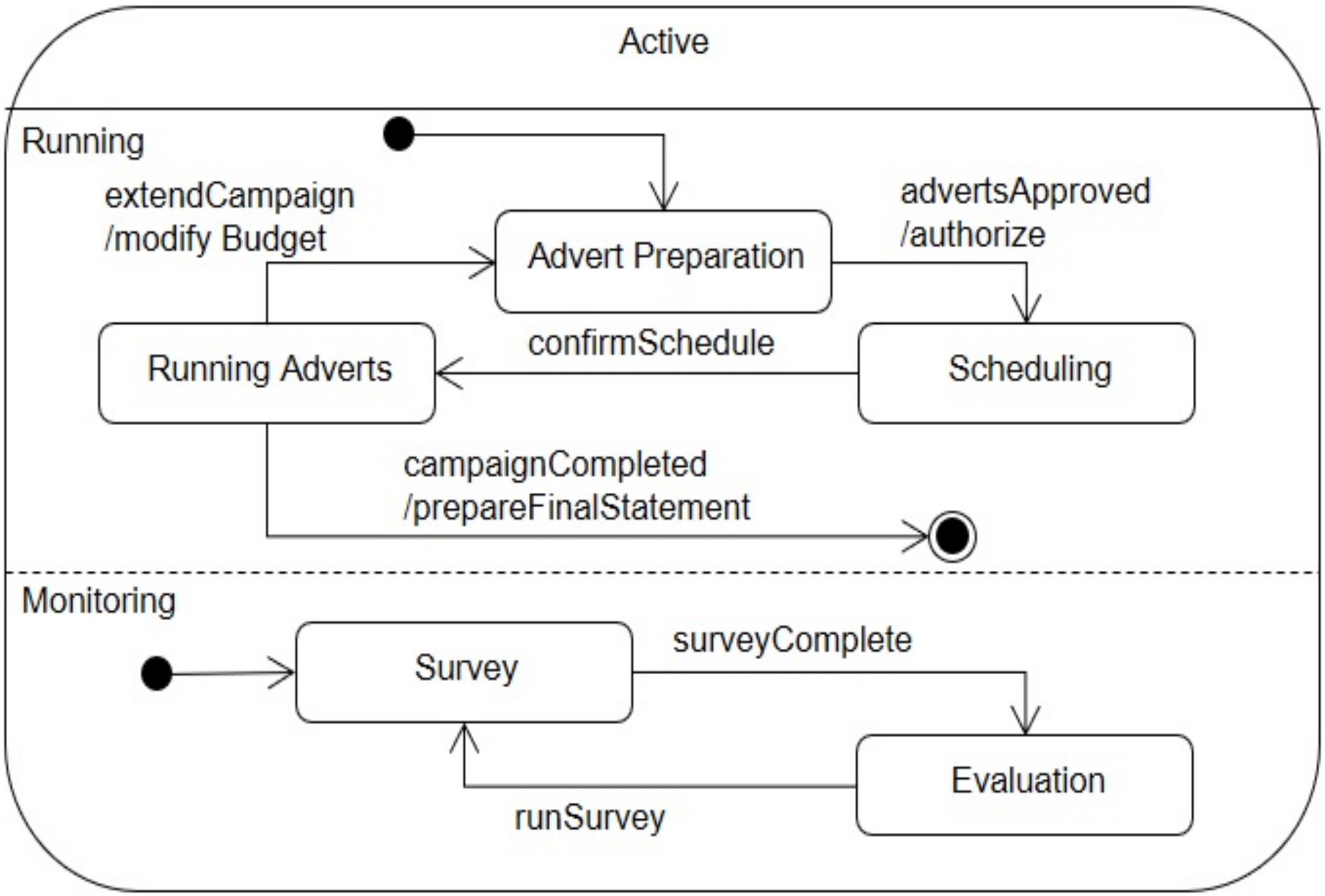
Decomposition compartment

Nested States

The `Active` *state of* `Campaign` *with the detail hidden.*



The Active state with concurrent substates



Concurrent States

- A transition to a complex state is equivalent to a **simultaneous transition** to the initial states of each concurrent state machine.

- An **initial state** must be specified in both nested state machines in order to avoid ambiguity about which substate should first be entered in each concurrent region.

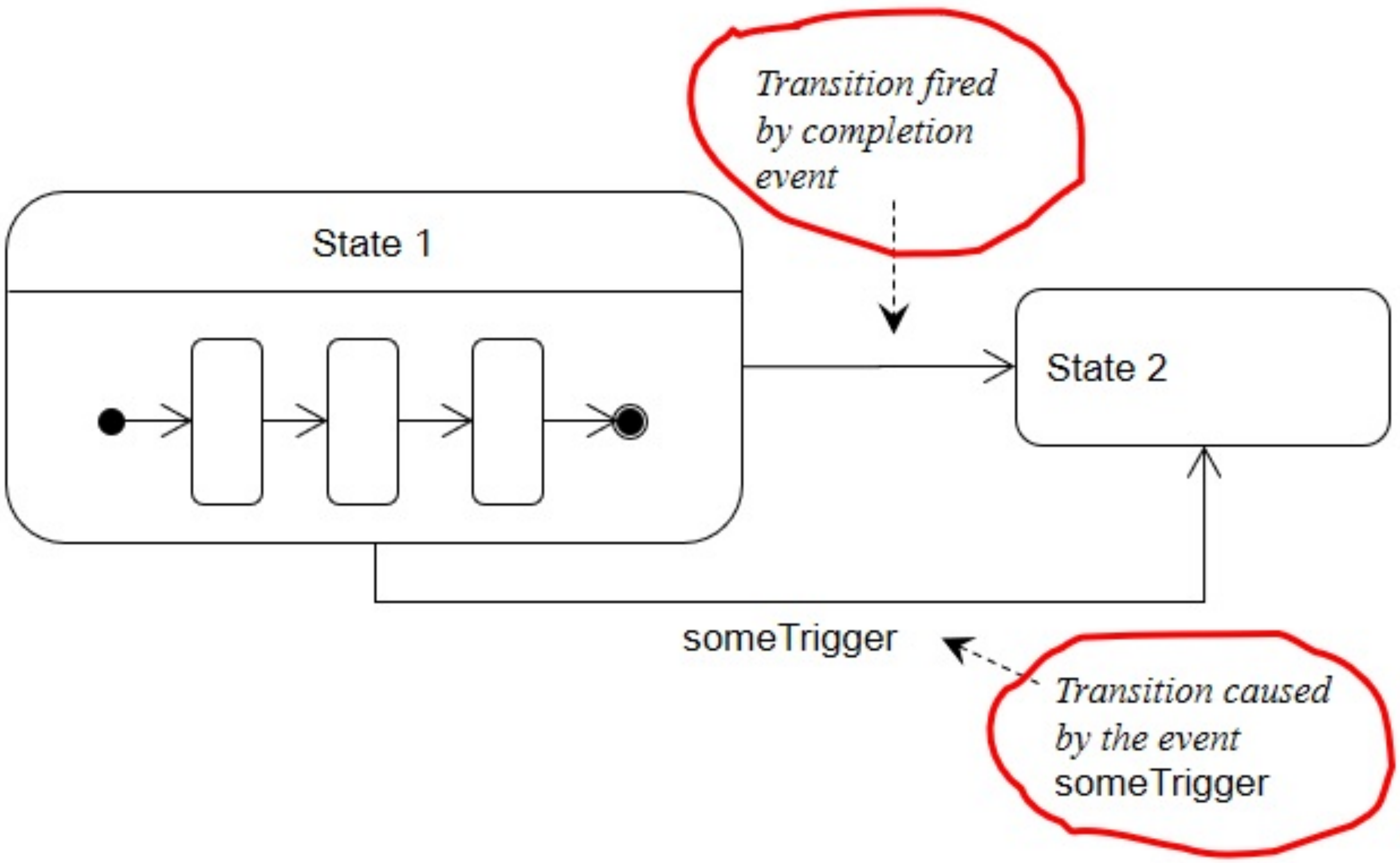
- A transition to the `Active` state means that the `Campaign` object <u>simultaneously enters</u> the `Advert Preparation` and `Survey` states.

Concurrent States

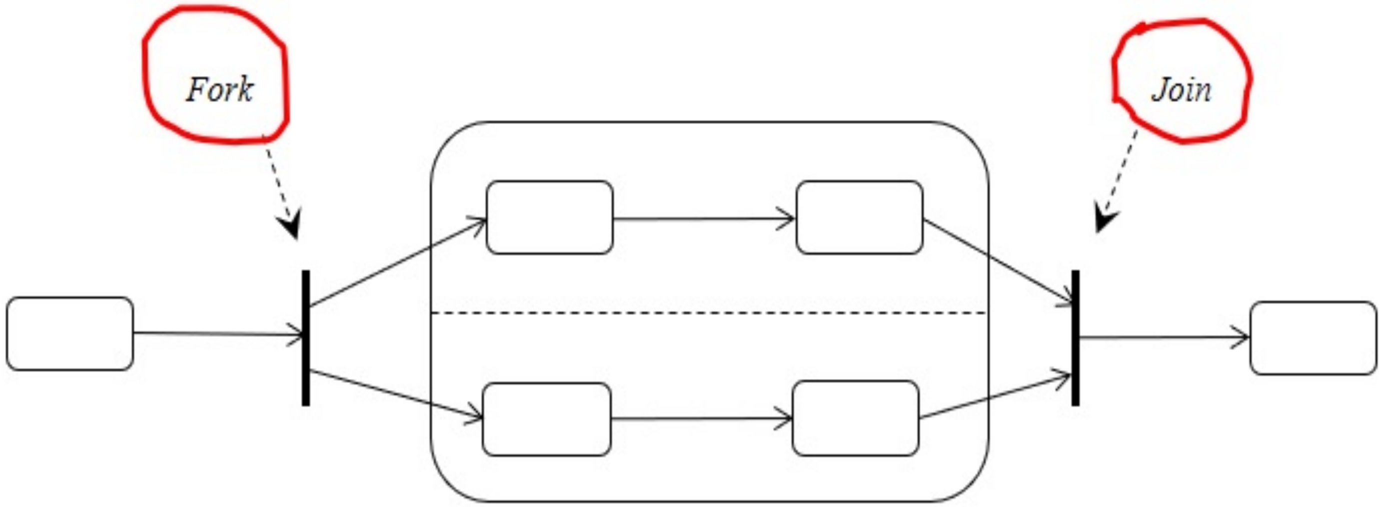
- Once the composite state is entered, a transition may occur within either concurrent region **without having any effect** on the state in the other concurrent region.

- A transition **out** of the `Active` state applies to **all** its substates (no matter how deeply nested).

Completion Event



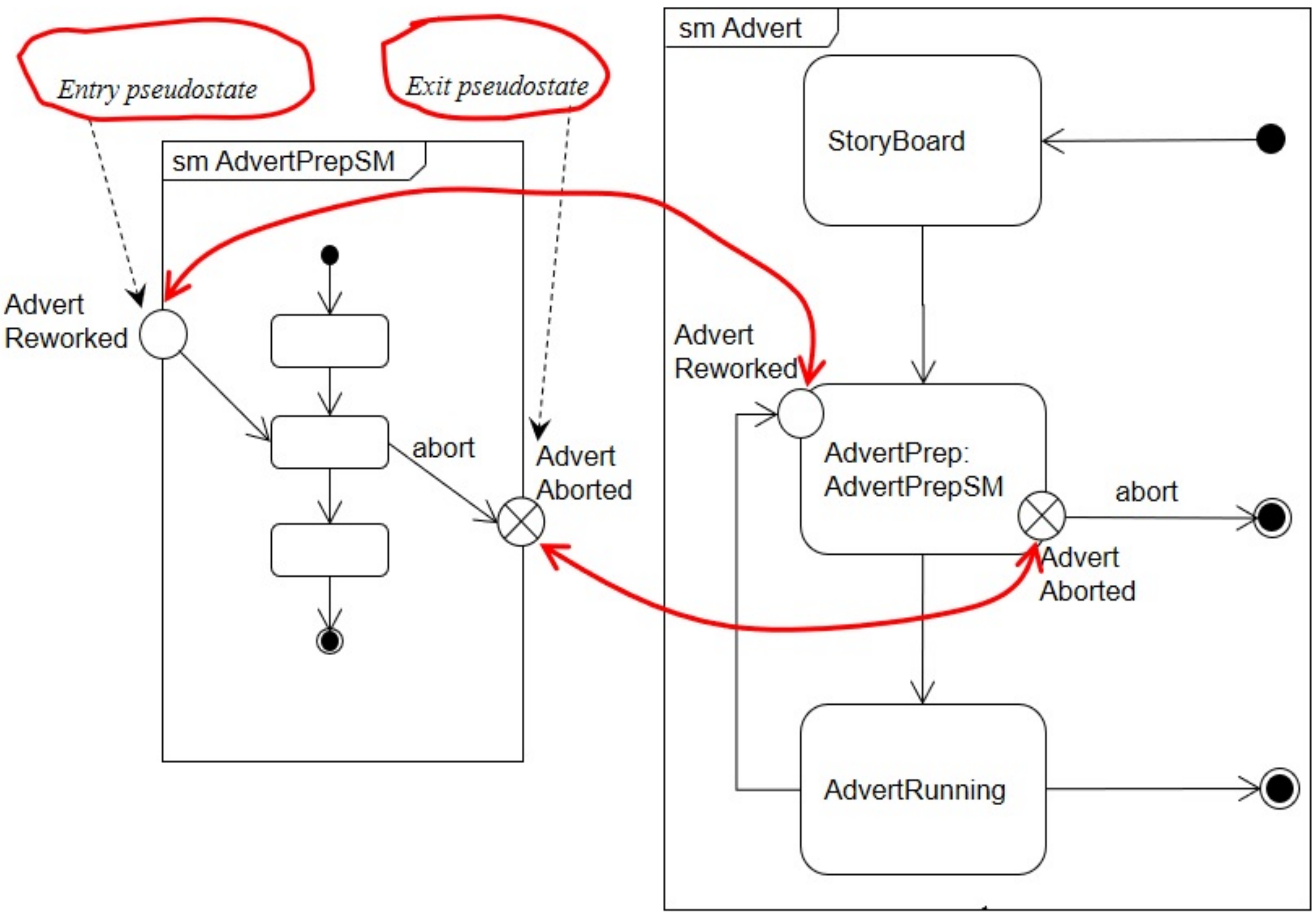
Synchronized Concurrent Threads



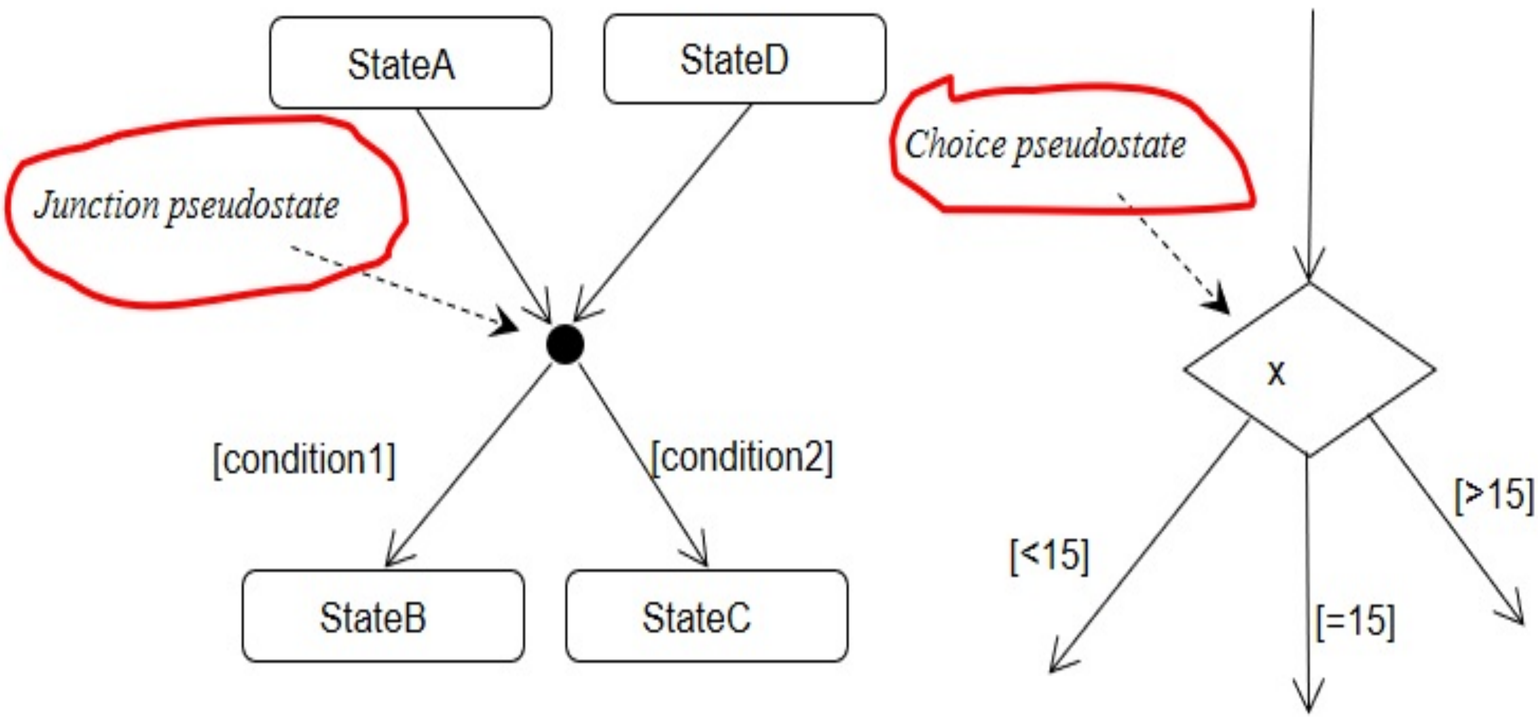
•Explicitly showing how an event triggering a transition to a state with nested concurrent states causes specific concurrent substates to be entered.

•Shows that the composite state is not exited until both concurrent nested state machines are exited.

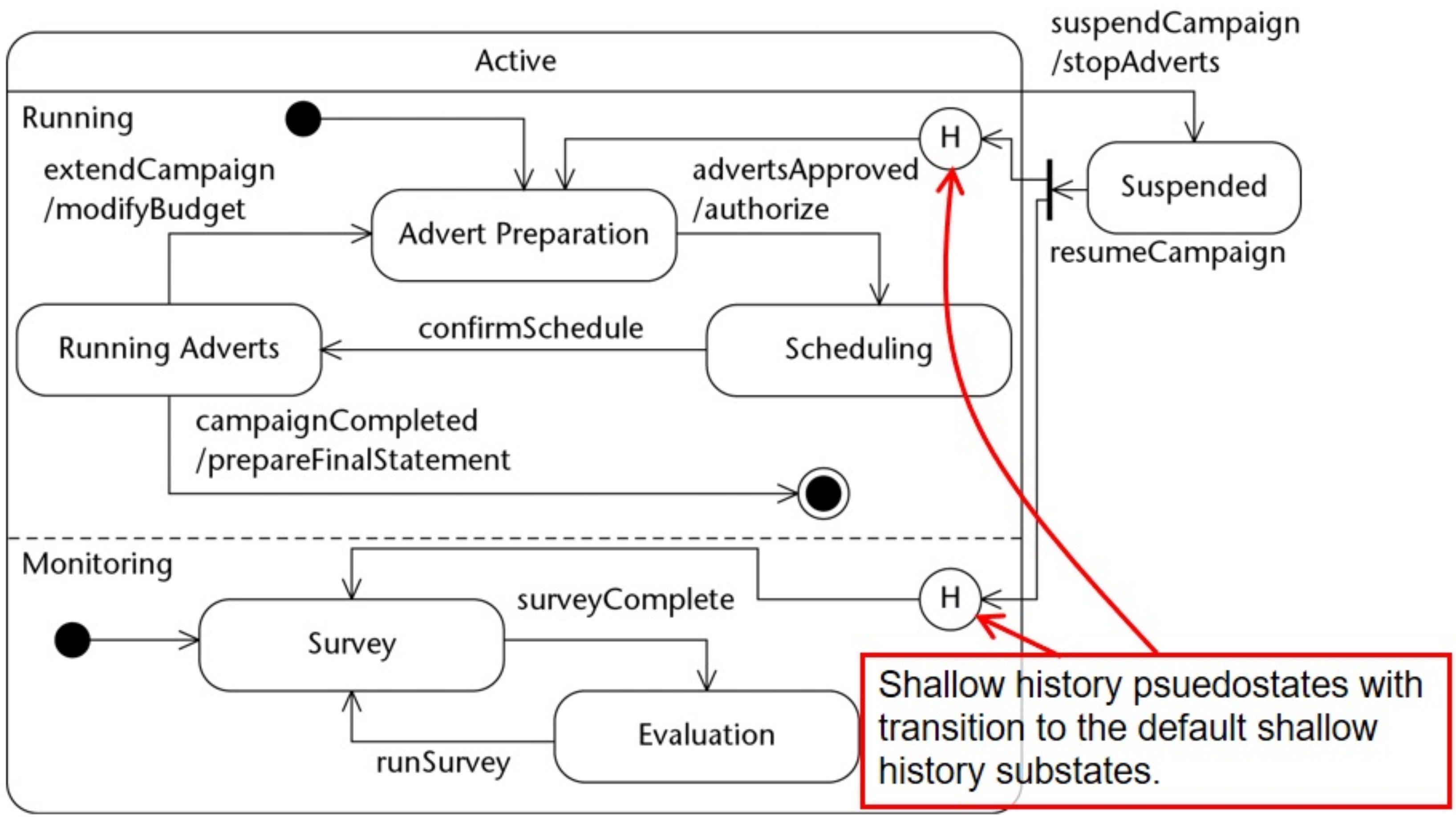
Entry & Exit Pseudostates



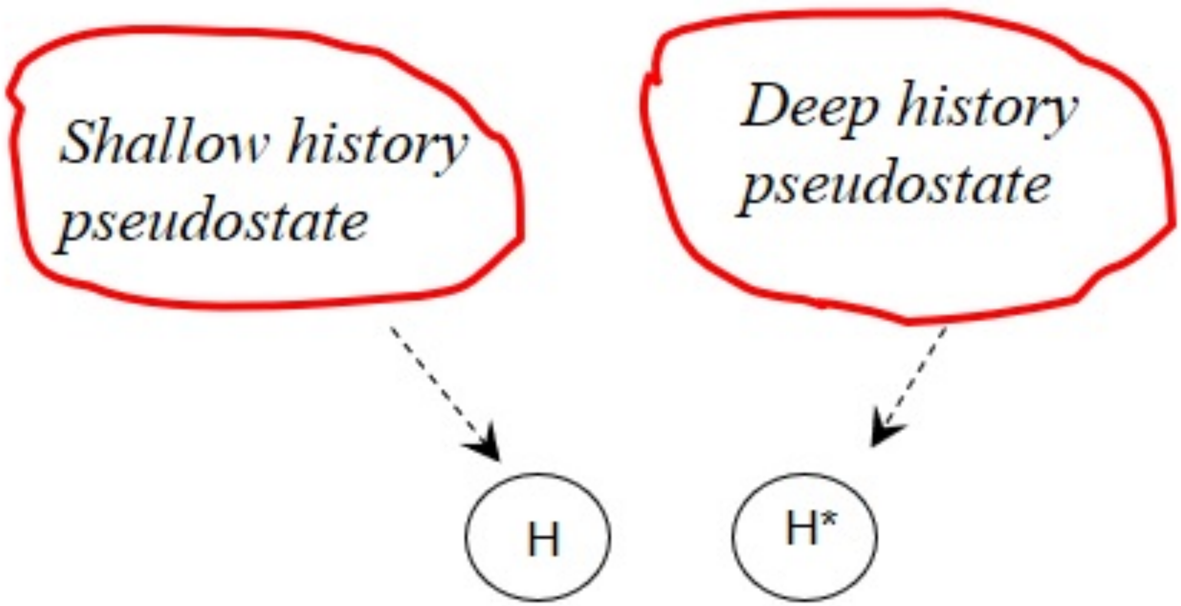
Junction & Choice Pseudostates



History Pseudostates



History Pseudostates



Preparing state machines

- Two approaches may be used:
  a. Behavioural approach
  b. Life cycle approach

a. Behavioural Approach (Steps)

1. Examine all interaction diagrams that involve each class that has heavy messaging.

2. Identify the incoming messages on each interaction diagram that may correspond to events. Also identify the possible resulting states.

3. Document these events and states on a state machine.

4. Elaborate the state machine as necessary to cater for additional interactions as these become evident, and add any exceptions.

a. Behavioural Approach (Steps)

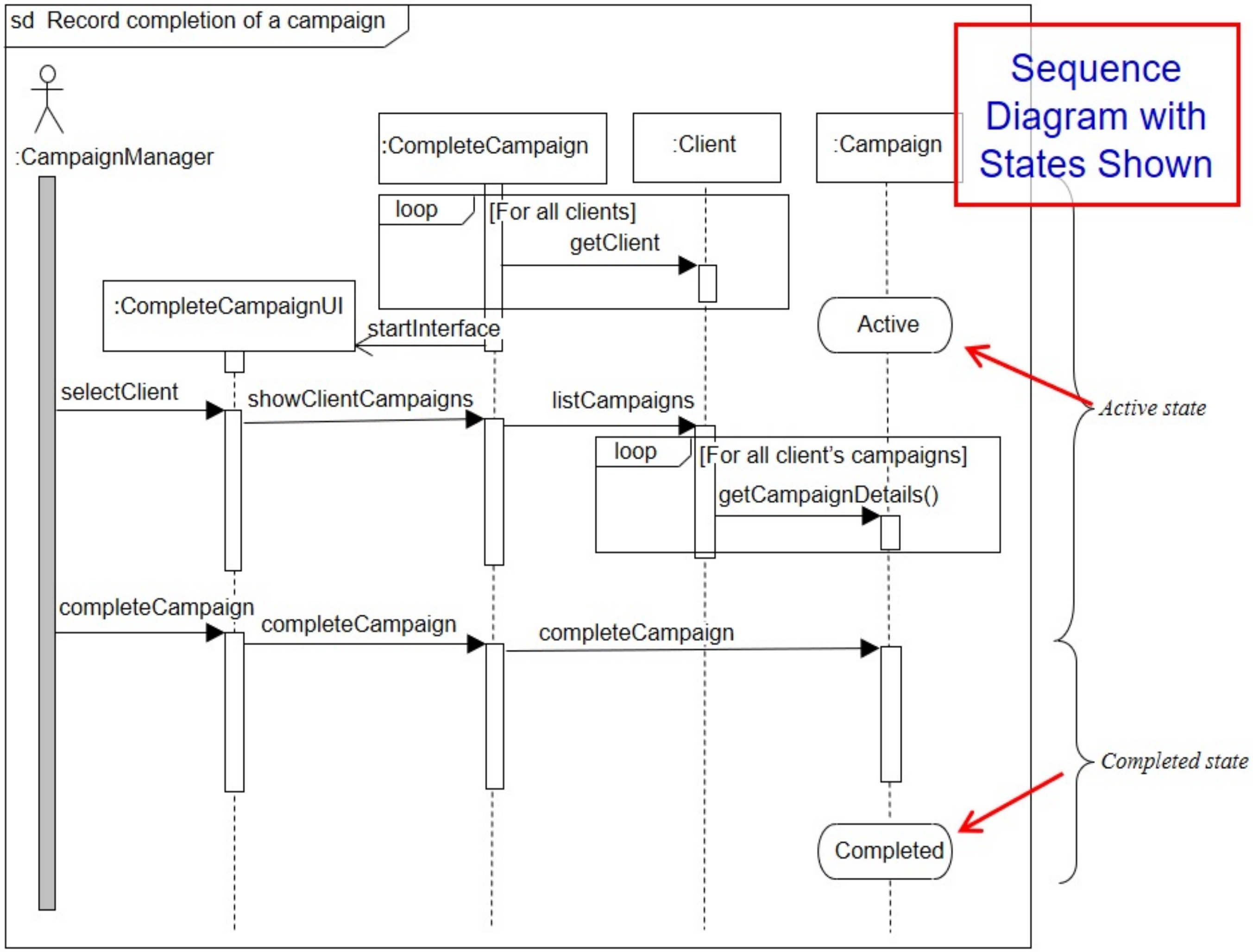
5. Develop any nested state machines (unless this has already been done in an earlier step).

6. Review the state machine to ensure consistency with use cases. In particular, check that any constraints that are implied by the state machine are appropriate.

7. Iterate steps 4, 5 and 6 until the state machine captures the necessary level of detail.

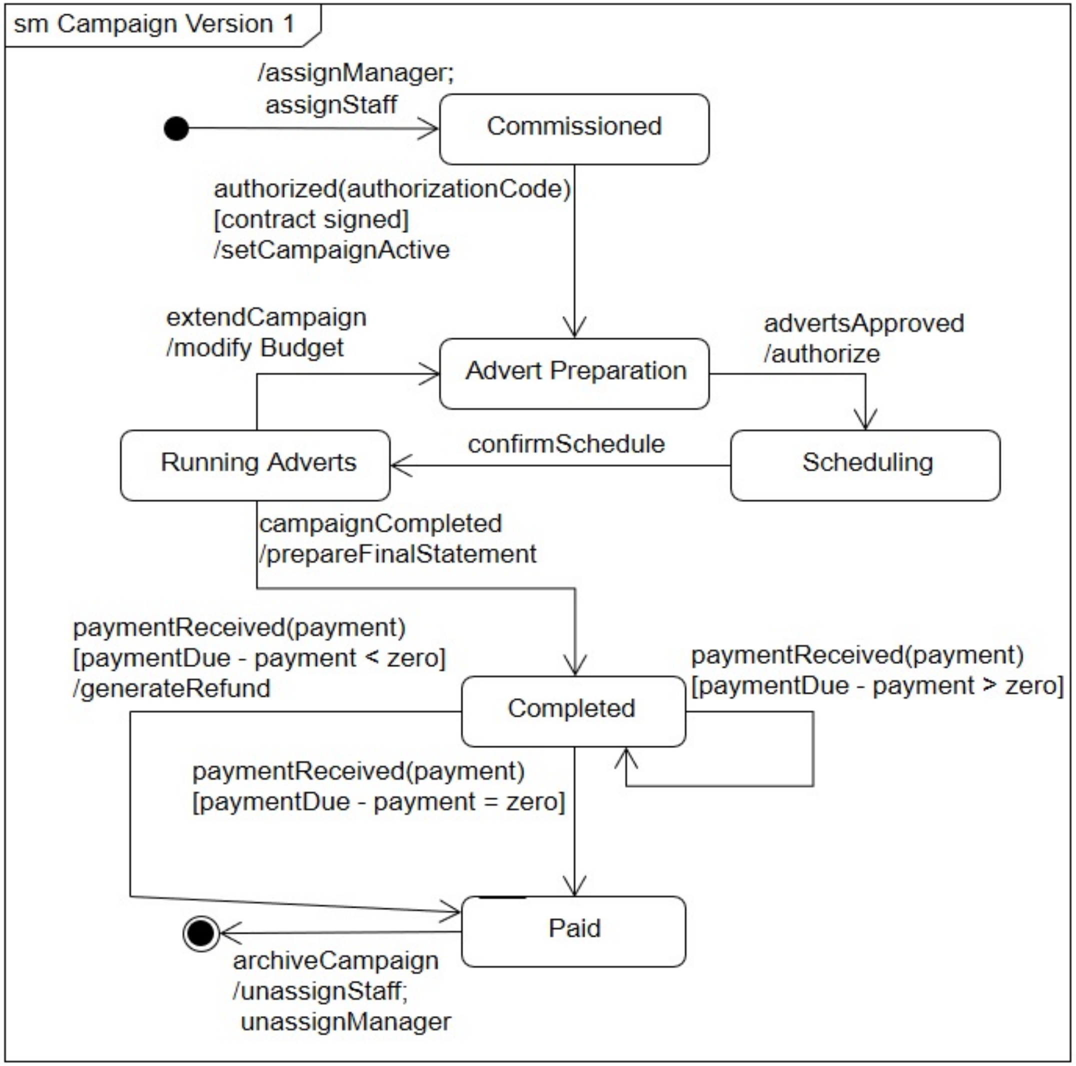
8. Check the consistency of the state machine with the class diagram, with interaction diagrams and with any other state machines and models.

a. Behavioural Approach (Example)

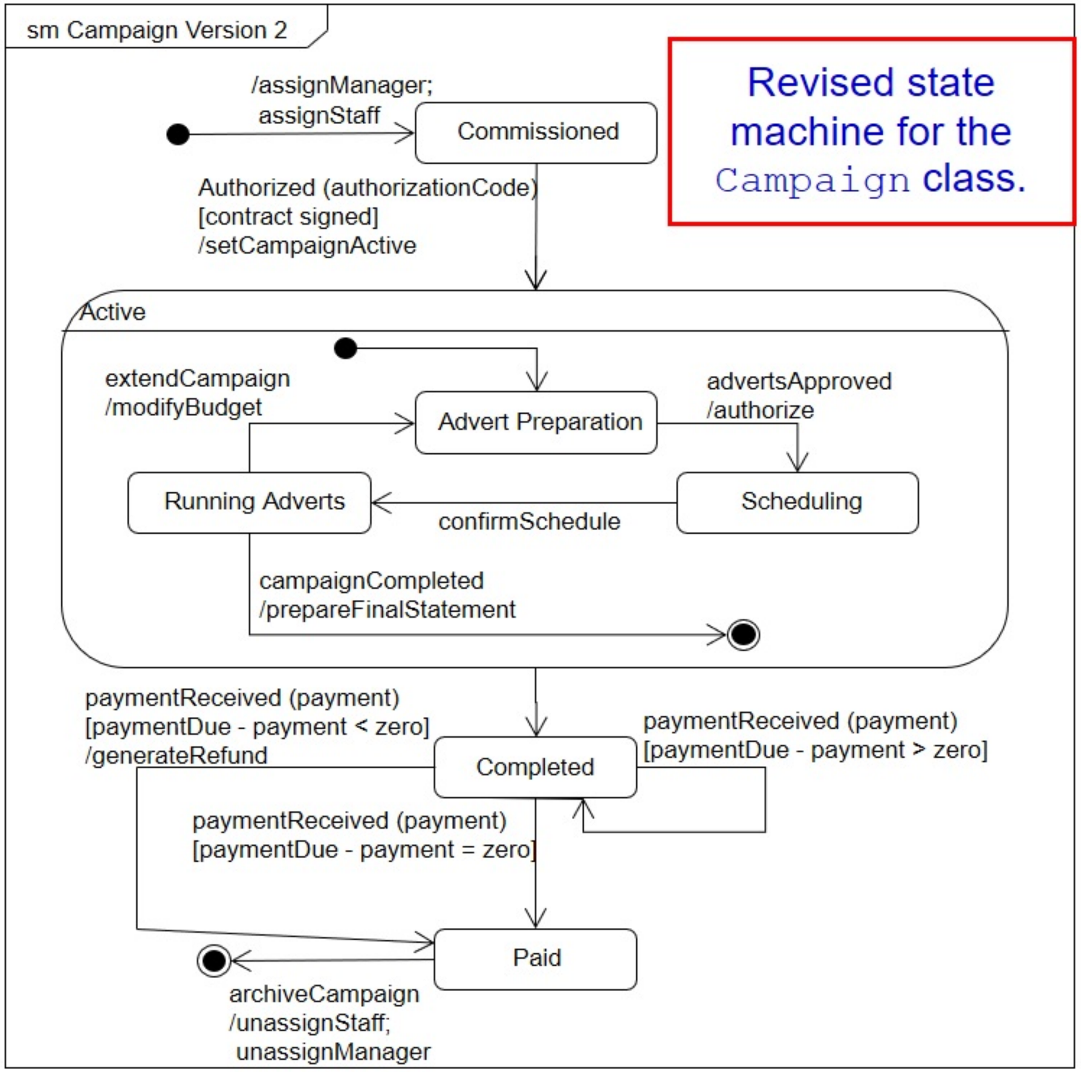


a. Behavioural Approach (Example)

Initial state machine for the Campaign class—using a behavioral approach.

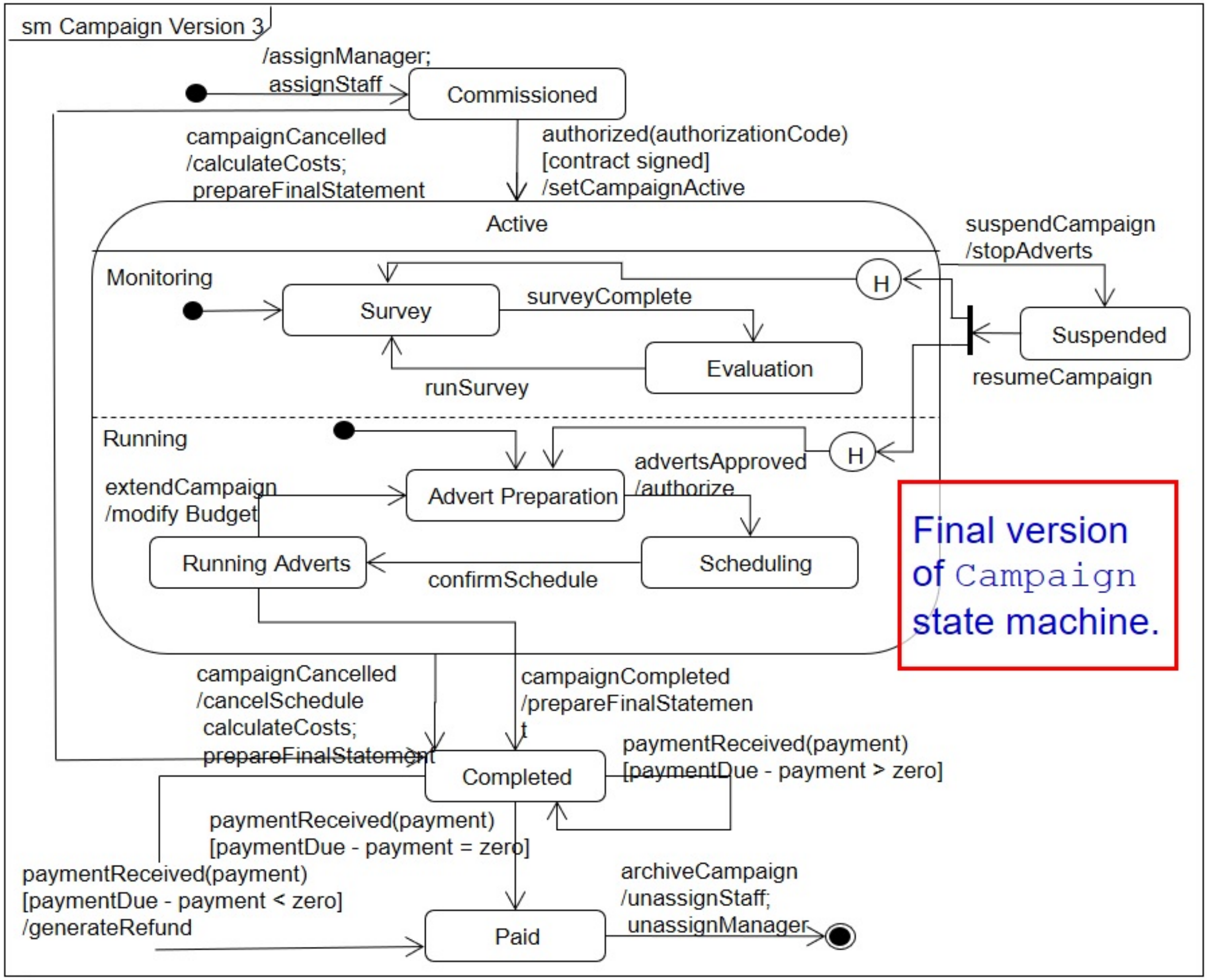


a. Behavioural Approach (Example)



Revised state machine for the `Campaign` class.

a. Behavioural Approach (Example)



Final version of `Campaign` state machine.

b. Life Cycle Approach

- Consider the life cycles for objects of each class.
- Events and states are identified directly from use cases and from any other requirements documentation that happens to be available.
- First, the main system events are listed.
- Each event is then examined in order to determine which objects are likely to have a state dependent response to it.

b. Life Cycle Approach (Steps)

1. Identify major system events.

2. Identify each class that is likely to have a state dependent response to these events.

3. For each of these classes produce a first-cut state machine by considering the typical life cycle of an instance of the class.

4. Examine the state machine and elaborate to encompass more detailed event behaviour.

b. Life Cycle Approach (Steps)

5. Enhance the state machine to include alternative scenarios.

6. Review the state machine to ensure that is consistent with the use cases. In particular, check that the constraints that the state machine implies are appropriate.

7. Iterate through steps 4, 5 and 6 until the state machine captures the necessary level of detail.

8. Ensure consistency with class diagram and interaction diagrams and other state machines.

b. Life Cycle Approach

- Less formal than the behavioural approach in its initial identification of events and relevant classes.

- Often helpful to use a combination of the two, since each provides checks on the other.

PROTOCOL STATE MACHINES

Protocol State Machines

- UML 2.0 introduced a distinction between protocol and behavioural state machines.
- All the state machines so far have been behavioural.
- Protocol state machines differ in that they only show all the legal transitions with their <u>pre- and post-conditions</u>.

Protocol State Machines

- The states of a protocol state machine cannot have
  - entry, exit or do activity sections.
  - deep or shallow history states.
- All transitions must be protocol transitions.

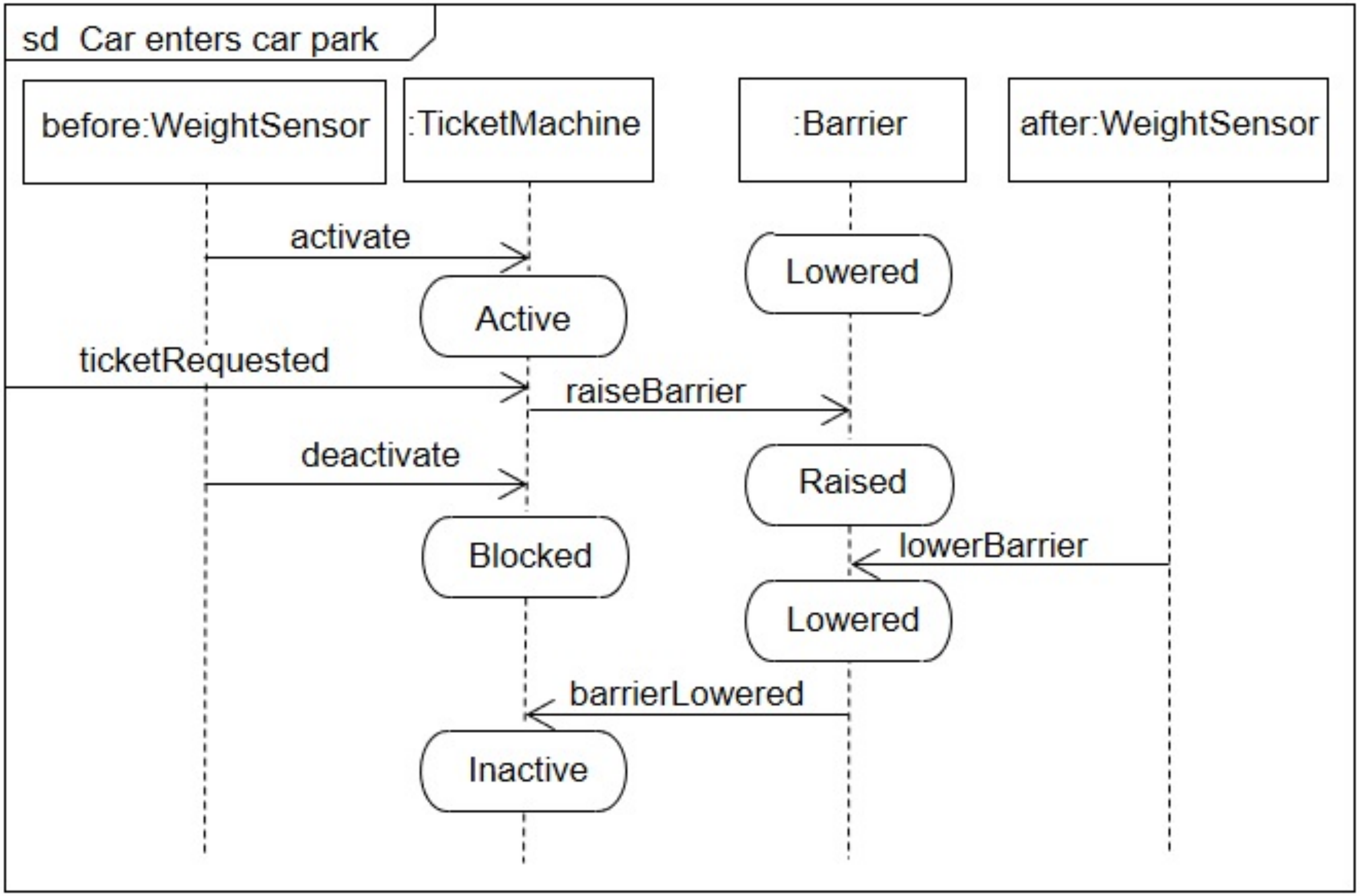
Protocol State Machines (Syntax)

- The syntax for a protocol transition label is as follows:

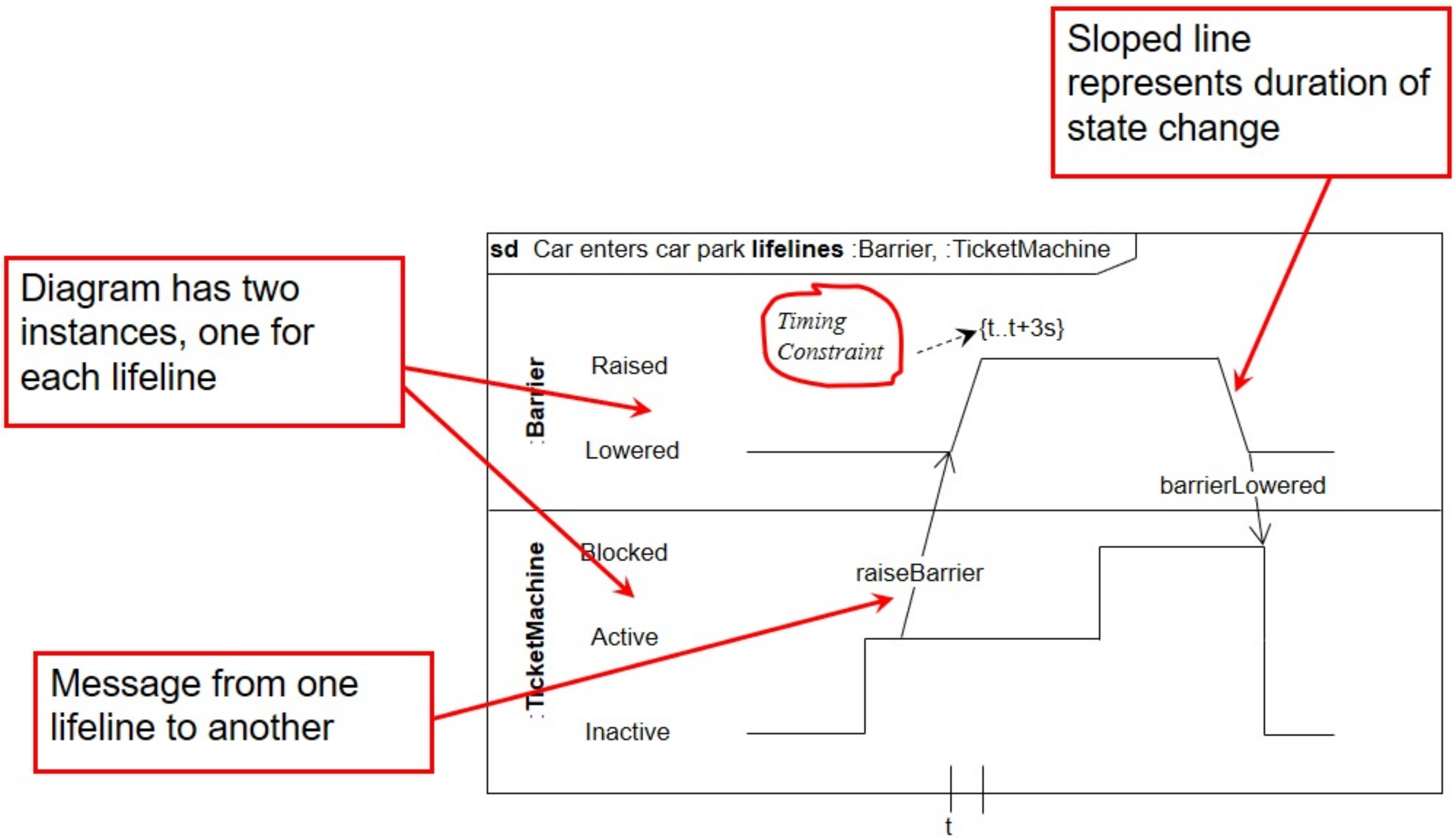
  ```
  '[' pre-condition ']' trigger '/'
  '[' post-condition ']’
  ```

- Unlike behavioural transitions protocol transitions do not have activity expressions.

Sequence Diagram for Protocol State Machine (Example)



Timing Diagram for Protocol State Machine (Example)

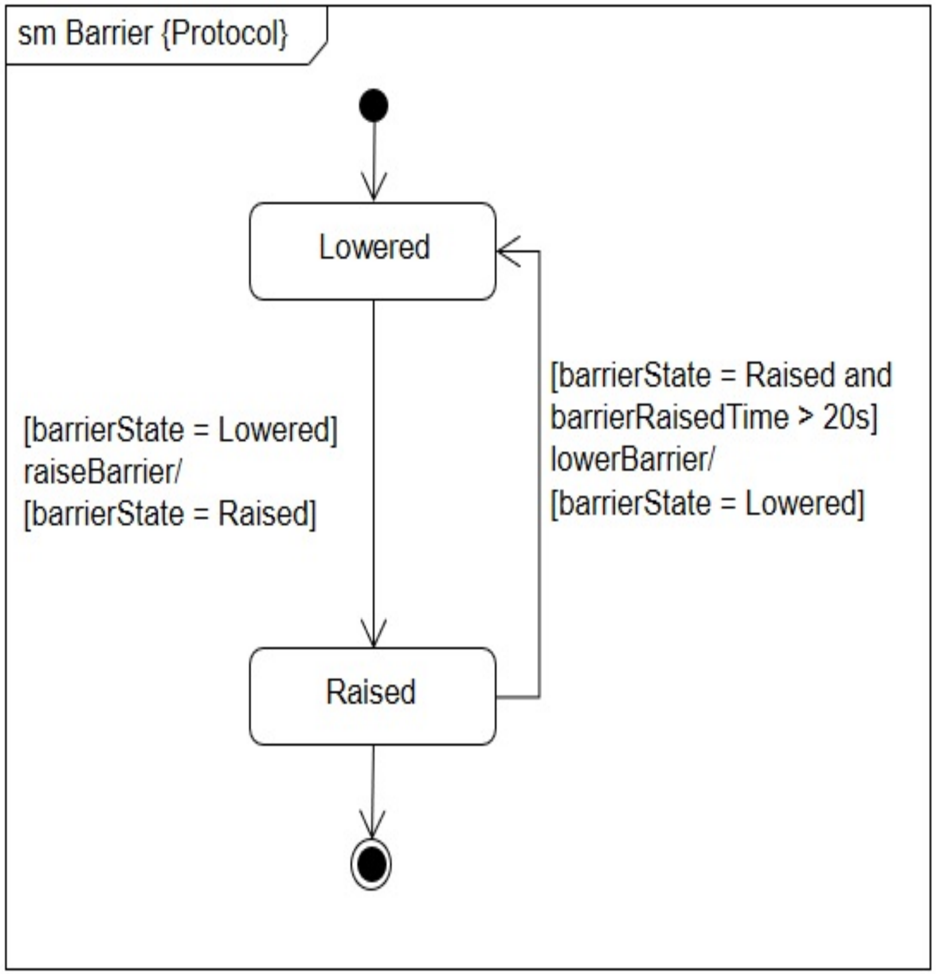


Protocol State Machine (Example)

Precondition:
barrierState = Lowered

Trigger: raiseBarrier

Post-condition:
barrierState = Raised



Precondition:
barrierState = Raised
and barrierRaisedTime >
20s

Trigger: lowerBarrier

Post-condition:
barrierState = Lowered

Consistency Checking

- Every event should appear as an incoming message for the appropriate object on an interaction diagram(s).
- Every action should correspond to the execution of an operation on the appropriate class, and perhaps also to the dispatch of a message to another object.
- Every event should correspond to an operation on the appropriate class (but note that not all operations correspond to events).
- Every outgoing message sent from a state machine must correspond to an operation on another class.

Consistency Checking

- Consistency checks are an important task in the preparation of a complete set of models.
- Highlights omissions and errors, and encourages the clarification of any ambiguity or incompleteness in the requirements.

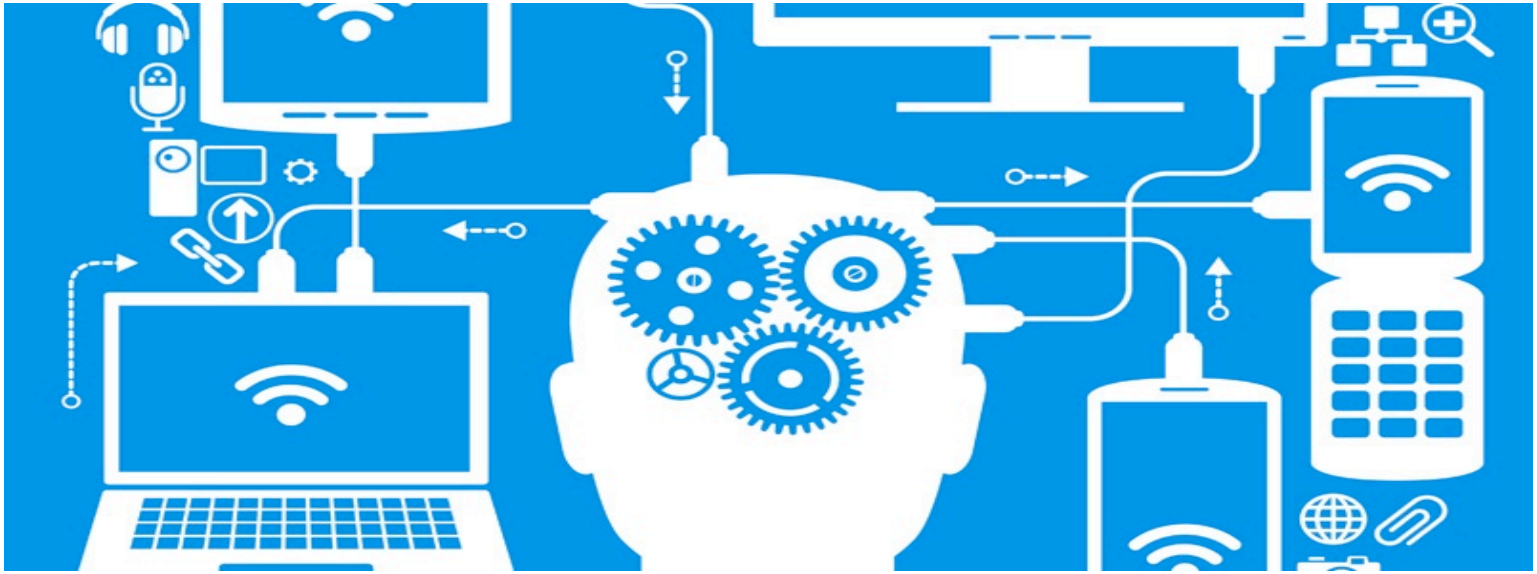
Key points

- The purposes of having state machine diagrams
  - To identify requirements for control in an application.
  - To show the different states of the class and what events cause the class to change from one state to another.
- Object life cycles can be modelled using state machines.
- State machine diagrams can be developed using two approaches (behavioural and life cycle).
- Concurrent behaviour in an object can be modelled using state meachines.
- Protocol state machines differ in that they only show all the legal transitions with their pre- and post-conditions.
- Consistency checks are an important task in the preparation of a complete set of models.
- .

References

- Alan Dennis, Barbara Haley Wixom & David Tegarden. 2015. Systems Analysis and Design with UML, 5th edition, Wiley.
- Simon Bennett, Steve McRobb & Ray Farmer. 2010. Object Oriented Systems Analysis and Design using UML 4th Edition, McGraw-Hill.
- UML Reference Manual (OMG, 2009)
- Bennett, Skelton and Lunn (2005)

In the next lecture..



Lecture 11: Design Modelling