Specifying Control Using State Machines

- CE202 Software Engineering
- M. Gardner

In this lecture you will learn:

- how to identify requirements for control in an application;
- how to model object life cycles using state machines;
- Basic state machine features
- More advanced features
- how to develop state machine diagrams from interaction diagrams;
- how to ensure consistency with other UML models.

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

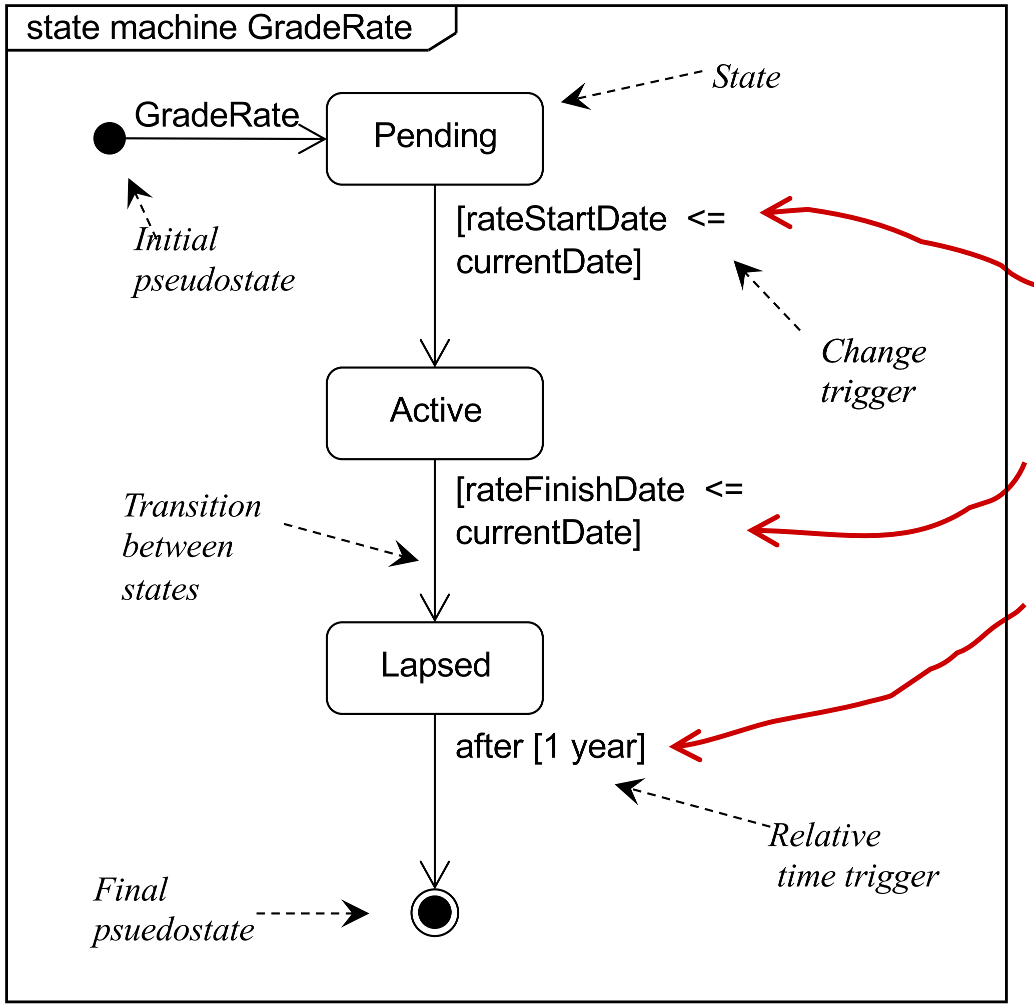
- A state describes a particular condition that a modelled element (e.g. object) may occupy for a period of time while it awaits some event or *trigger*.
- The possible states that an object can occupy are limited by its class.
- Objects of some classes have only one possible state.
- Conceptually, an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

State machine example

- 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**.

State machine example (continued)

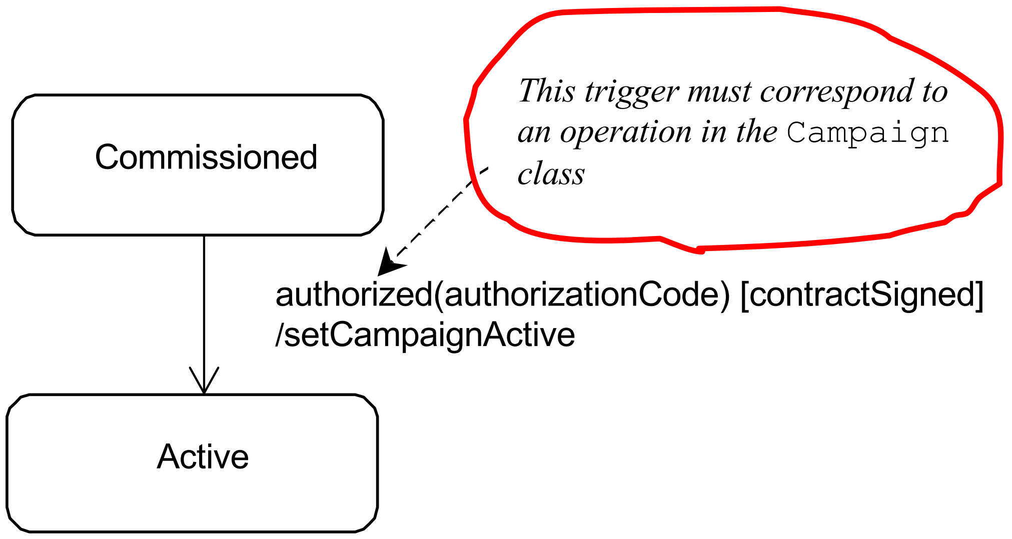
state machine for the class `GradeRate`.



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

Example Event (with more detailed trigger information)

State machine for a **Campaign** object


Commissioned
Active
authorized(authorizationCode) [contractSigned] /setCampaignActive
This trigger must correspond to an operation in the Campaign class

Call and Signal Events

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

- Where *trigger-signature* takes following form
  - *event-name '(' parameter-list ')'*
  - Where *event-name* may be the call or signal name
  - Where *parameter-list* contains parameters of the form, separated by commas:
    - *parameter-name ':' type-expression*
    - Can use single-quotes for literals
- A constraint is a guard condition. The transition only occurs if the guard condition is true (Boolean)

- *activity-expression* is executed when a trigger is fired

Activity Expressions

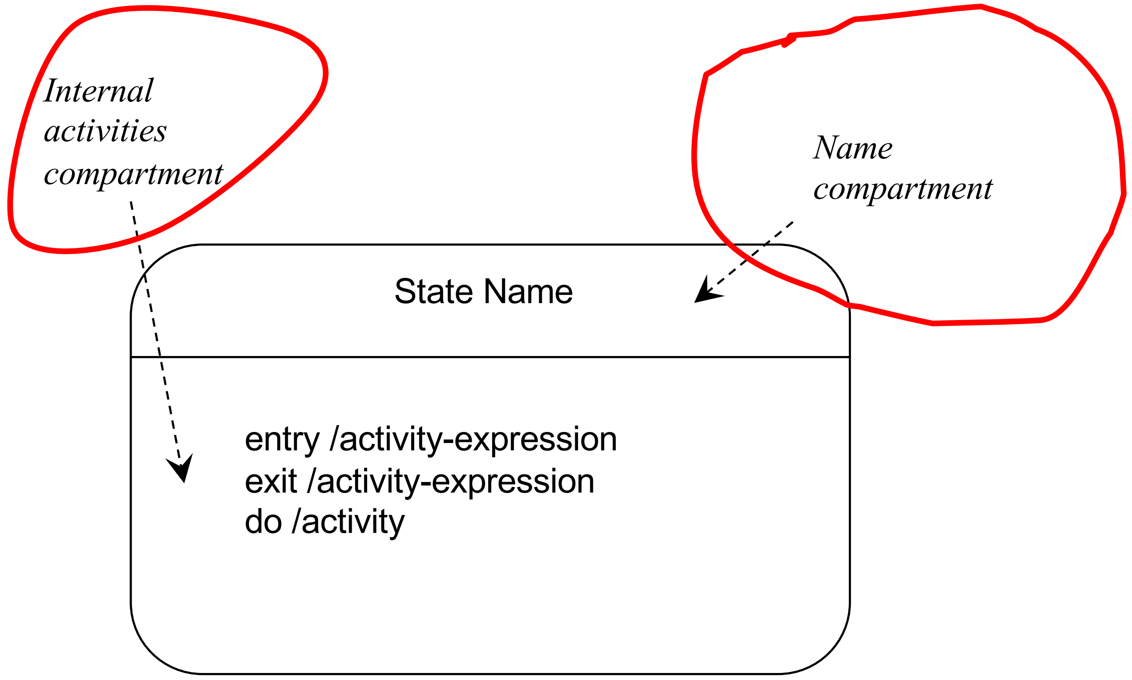
- May have multiple events
- Separated by semi-colons

Example:

left-mouse-down(location) [validItemSelected] / menuChoice = pickMenuItem(location); menuChoice.highlight

The SEQUENCE of actions is significant as it determines the order in which they are executed

Internal Activities



Internal activities compartment
Name compartment
State Name
entry /activity-expression
exit /activity-expression
do /activity

Used to model internal activities associated with a state
Three types of internal activity: entry, exit and state activities (do)
Transition into the state causes the entry activity to fire
Transition out of the state causes the exit activity to fire

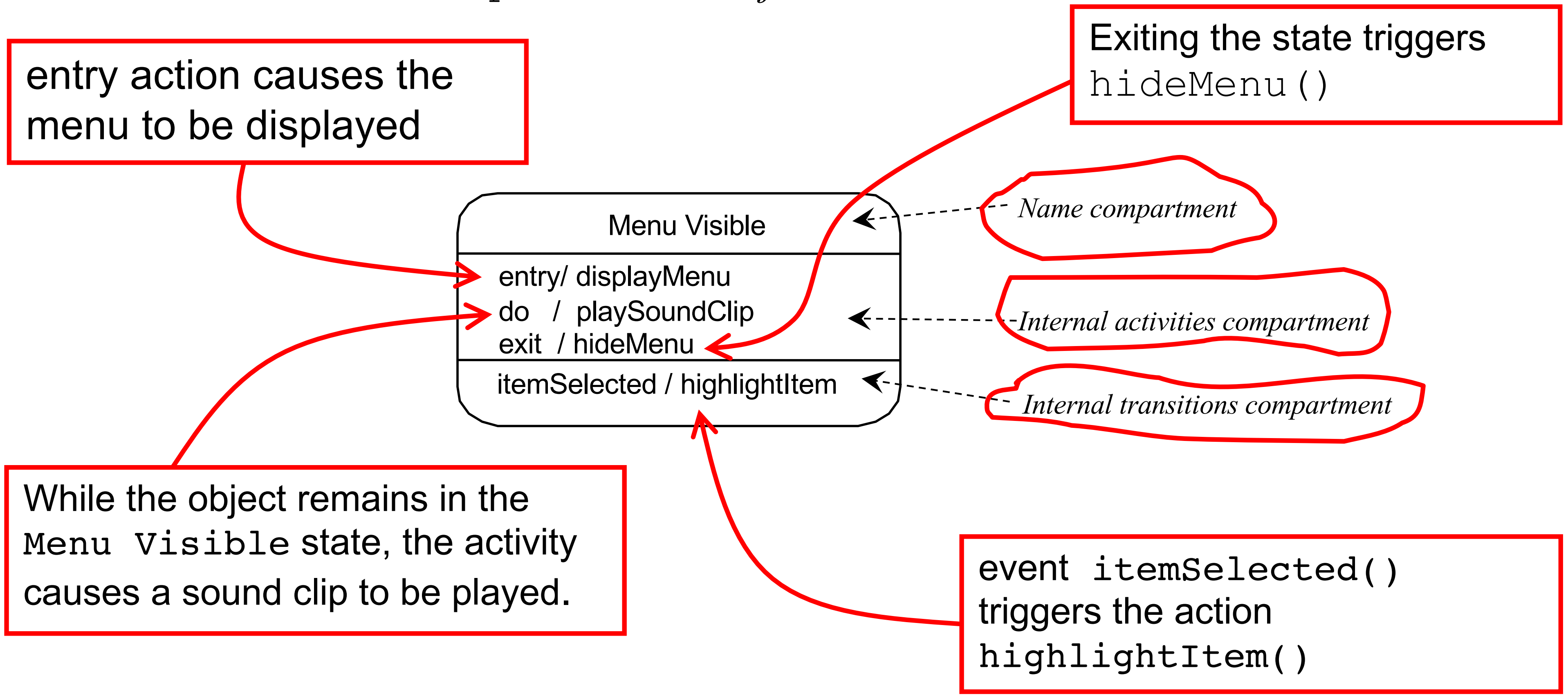
Internal Activities/Transitions

- Internal activities
  - *‘entry’ ‘/’ activity-name ‘(‘ parameter-list ‘)’*
  - ‘exit’ ‘/’ activity-name ‘(‘ parameter-list ‘)’
  - ‘do’ ‘/’ activity-name ‘(‘ parameter-list ‘)’

- Internal transitions
  - Same syntax as triggers

An example: ‘Menu Visible’ State

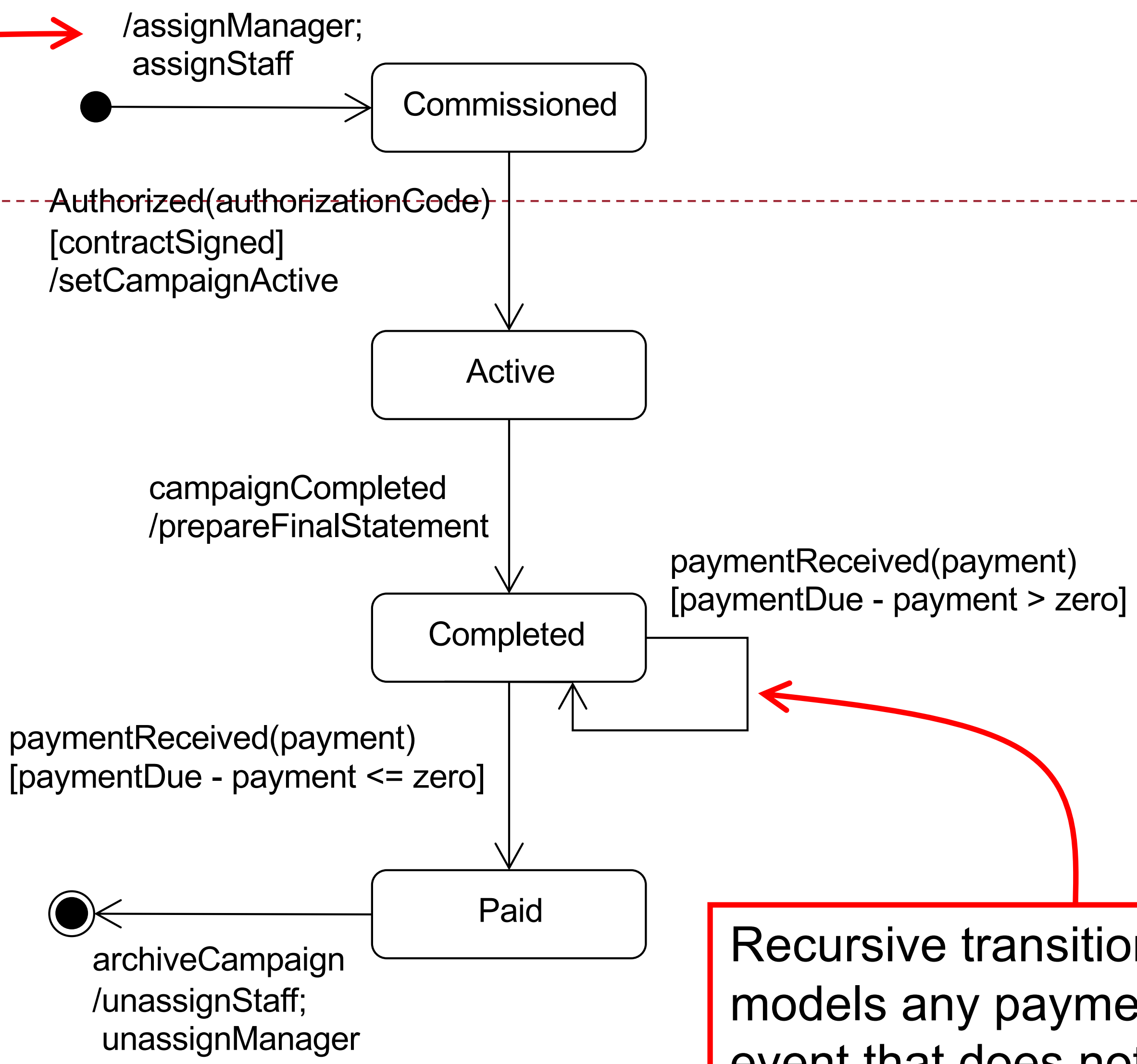
`Menu Visible` *state for a* `DropDownMenu` *object.*


entry action causes the menu to be displayed
Exiting the state triggers hideMenu()
Menu Visible
entry/ displayMenu
do / playSoundClip
exit / hideMenu
itemSelected / highlightItem
Name compartment
Internal activities compartment
Internal transitions compartment
While the object remains in the Menu Visible state, the activity causes a sound clip to be played.
event itemSelected() triggers the action highlightItem()

In this example, the entry action causes the menu to be displayed. While the object remains in the Menu Visible state, the activity causes a sound clip to be played and, if the event itemSelected() occurs, the action highlightItem() is invoked. It is important to note that when the event itemSelected() occurs the Menu Visible state is not exited and entered and as a result the exit and entry actions are not invoked. When the state is actually exited the menu is hidden.

Action-expression assigning manager and staff on object creation

state machine for the class **`Campaign`**.

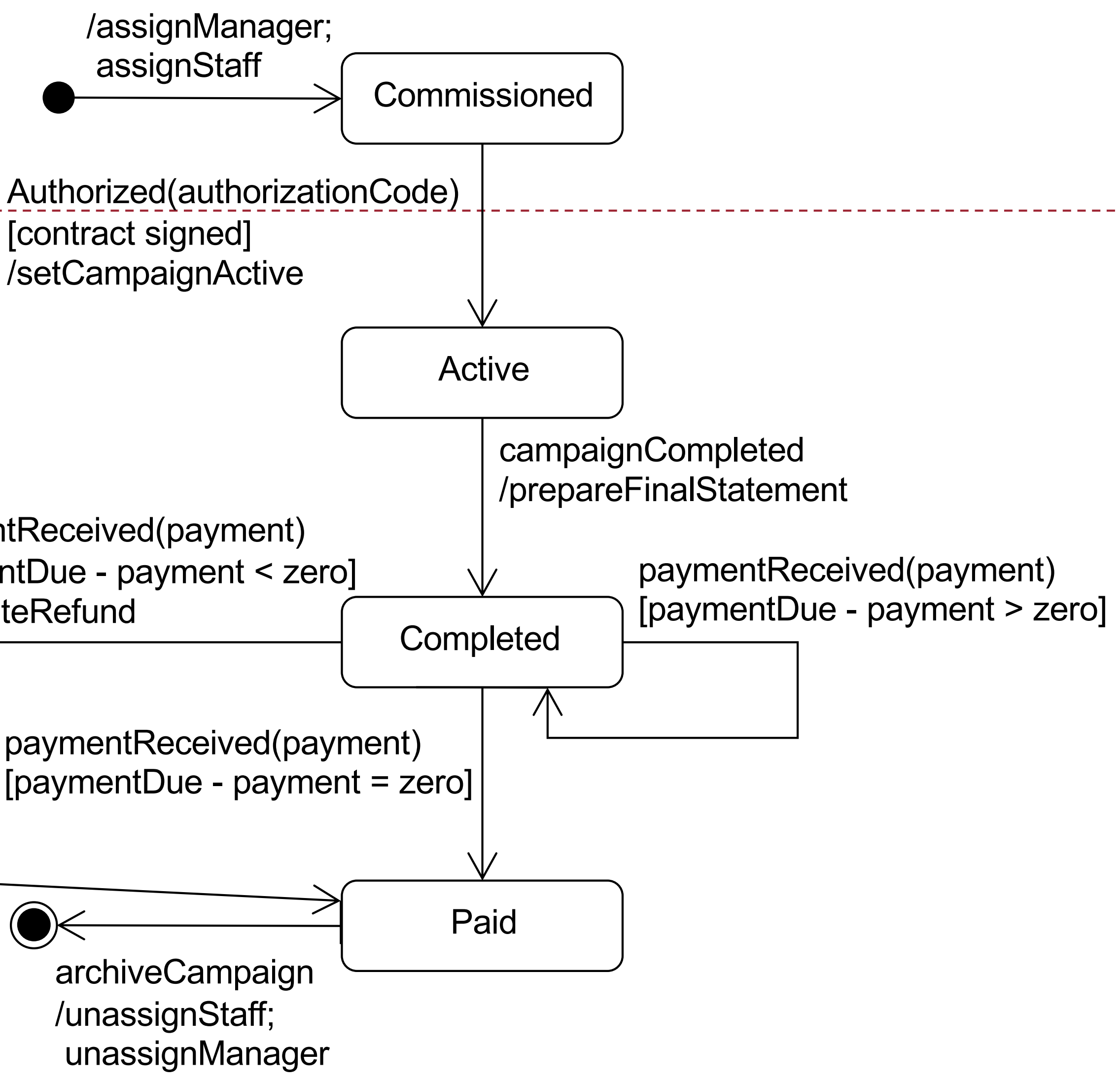

/assignManager; assignStaff
Commissioned
Authorized(authorizationCode) [contractSigned] /setCampaignActive
Active
campaignCompleted /prepareFinalStatement
Completed
paymentReceived(payment) [paymentDue - payment > zero]
paymentReceived(payment) [paymentDue - payment <= zero]
Paid
archiveCampaign /unassignStaff; unassignManager

Guard condition ensuring complete payment before entering `Paid`

Recursive transition models any payment event that does not reduce the amount due to zero or beyond.

The recursive transition from the Completed state models any payment event that does not reduce the amount due to zero or beyond. Only one of the two transitions from the Completed state (one of which is recursive) can be triggered by the paymentReceived event since the guard conditions are mutually exclusive. It would be bad practice to construct a state machine where one event can trigger two different transitions from the same state. A life-cycle is only unambiguous when all the transitions from each state are mutually exclusive.

A revised state machine for the class `Campaign`


/assignManager; assignStaff
Commissioned
Authorized(authorizationCode) [contract signed] /setCampaignActive
Active
campaignCompleted /prepareFinalStatement
paymentReceived(payment) [paymentDue - payment < zero] /generateRefund
Completed
paymentReceived(payment) [paymentDue - payment > zero]
paymentReceived(payment) [paymentDue - payment = zero]
Paid
archiveCampaign /unassignStaff; unassignManager

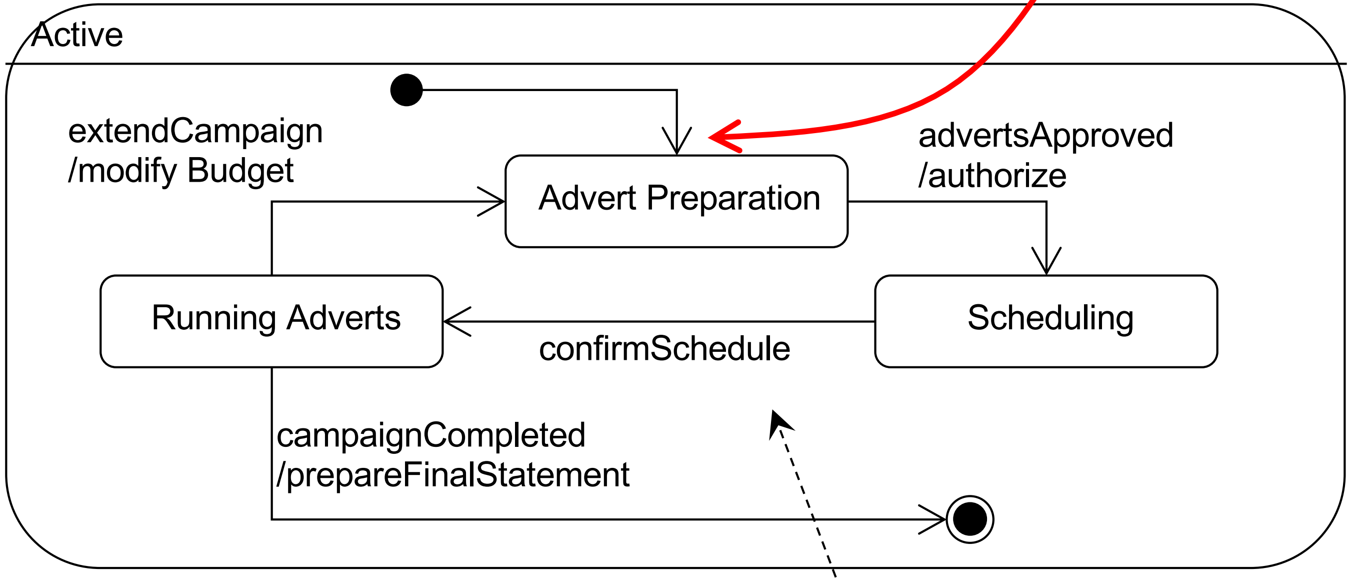
If the user requirements were to change, so that an overpayment is now to result in the automatic generation of a refund, a new transition is added.

If the user requirements were to change, so that an overpayment is now to result in the automatic generation of a refund, the state machine can be changed as follows. Since the action that results from an overpayment is different from the action that results from a payment that reduces paymentDue to zero, a new transition is needed from the Completed state to the Paid state. The guard conditions from the Completed state must also be modified.

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


Active
extendCampaign /modify Budget
Advert Preparation
advertsApproved /authorize
Running Adverts
Scheduling
confirmSchedule
campaignCompleted /prepareFinalStatement

Decomposition compartment

In the nested state machine within the Active state, there is an initial state symbol with a transition to the first substate that a Campaign object enters when it becomes active. The transition from the initial pseudostate symbol to the first substate (Advert Preparation) should not be labelled with an event but it may be labelled with an action, though it is not required in this example. It is implicitly fired by any transition to the Active state. A final pseudostate symbol may also be shown on a nested state diagram. A transition to the final pseudostate symbol represents the completion of the activity in the enclosing state (i.e. Active) and a transition out of this state triggered by the completion event. This transition may be unlabelled (as long as this does not cause any ambiguity) since the event that triggers it is implied by the completion event.

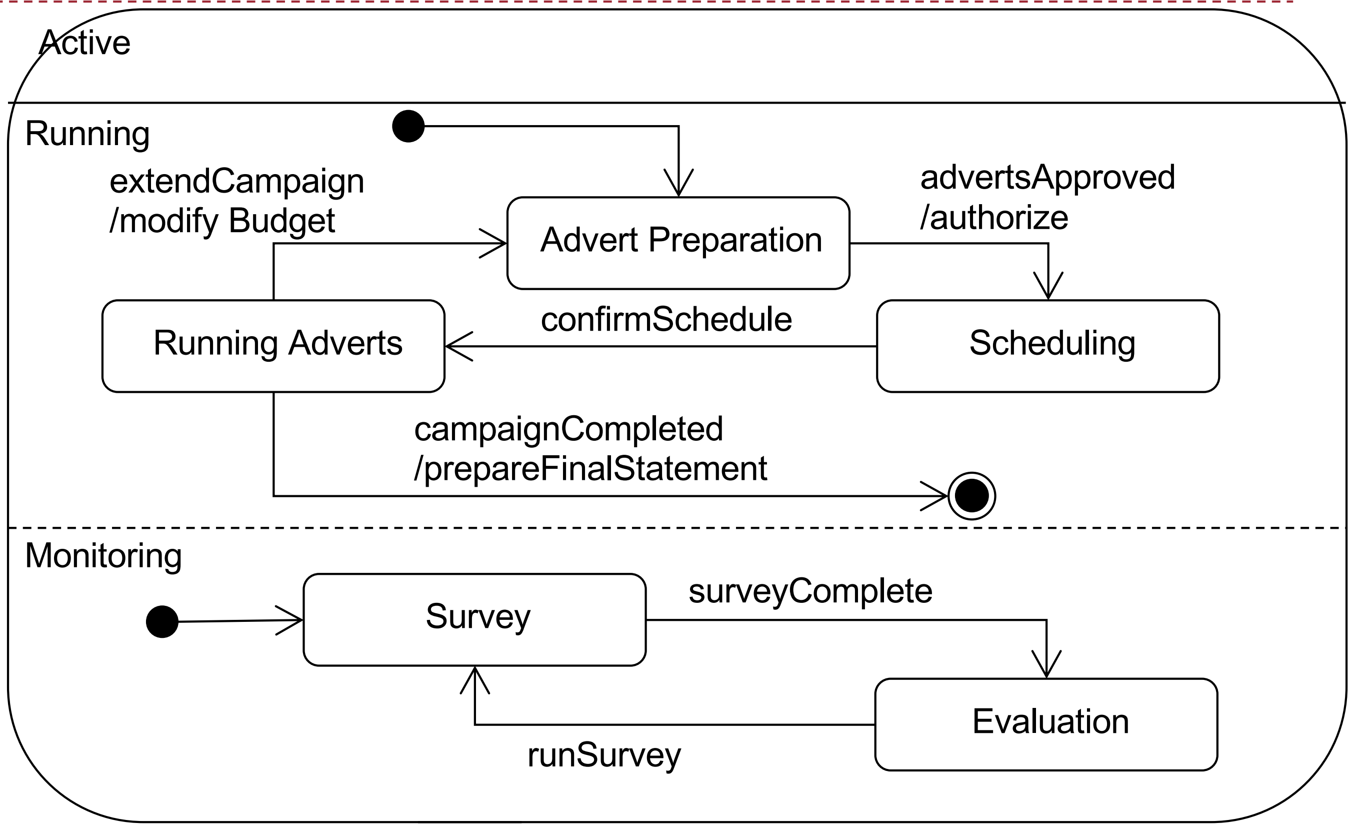
Summary of basic UML state machines

- What we have covered so far, forms the basis for most of the state machines we will consider in CE202
- Basic state machines will need:
  - **Start point** and an **initial starting state** (no trigger required)
  - Transitions to other states (triggers and **optional** activity expressions)
  - Each state will require a **sensible state name**
  - Decide on the level of detail required – **start simple**
    - Sometimes you can specify internal activities (but not always necessary – depends on the level of detail required)
    - Sometimes have nested state machines if needed
  - Sometimes an **exit point** (but not always)

Additional UML state machine features

The next slides describe additional features of UML state machines (some of which will be covered in the class exercises)

The Active state with concurrent substates.



Suppose that further investigation reveals that a campaign is surveyed and evaluated while it is also active. A campaign may occupy either the Survey substate or the Evaluation substate when it is in the Active state. Transitions between these two states are not affected by the campaign's current state in relation to the preparing and running of adverts. We model this by splitting the Active state into two concurrent nested state machines, Running and Monitoring, each in a separate sub-region of the Active state machine. This is shown by dividing the state icon with a dashed line.

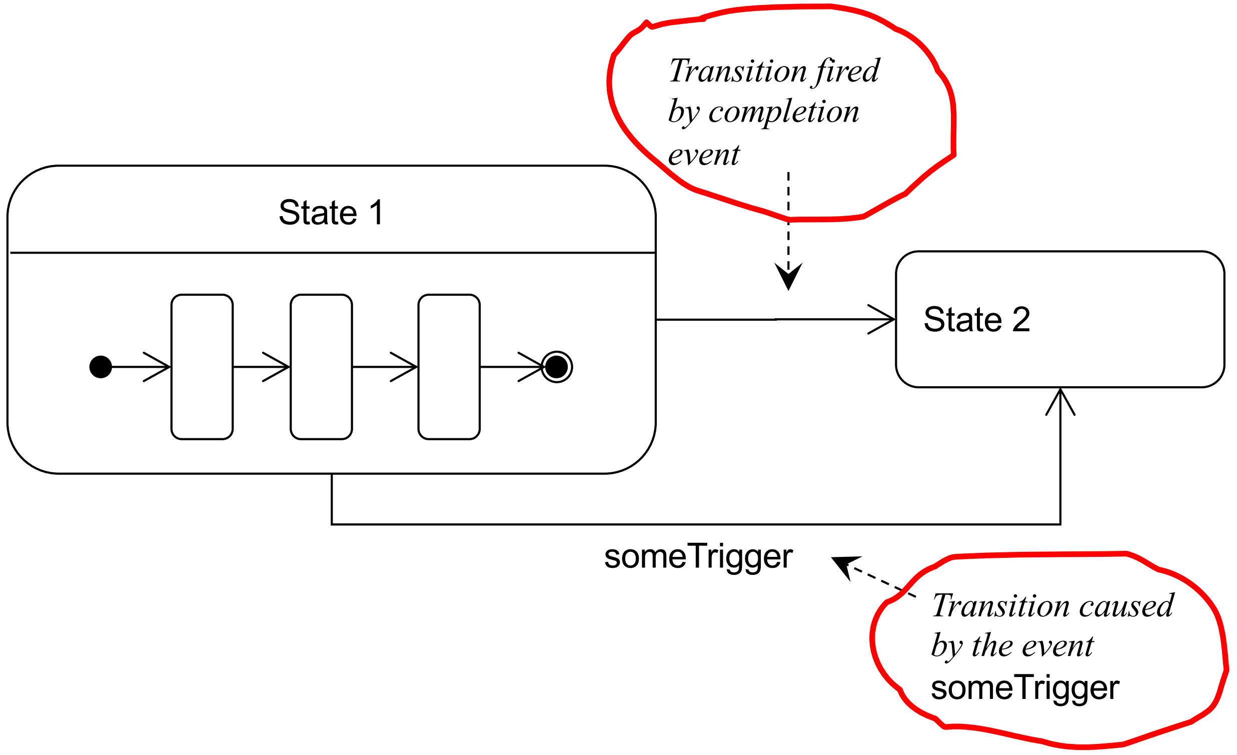
Concurrent States

- 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 simultaneously enters the `Advert Preparation` and `Survey` states.

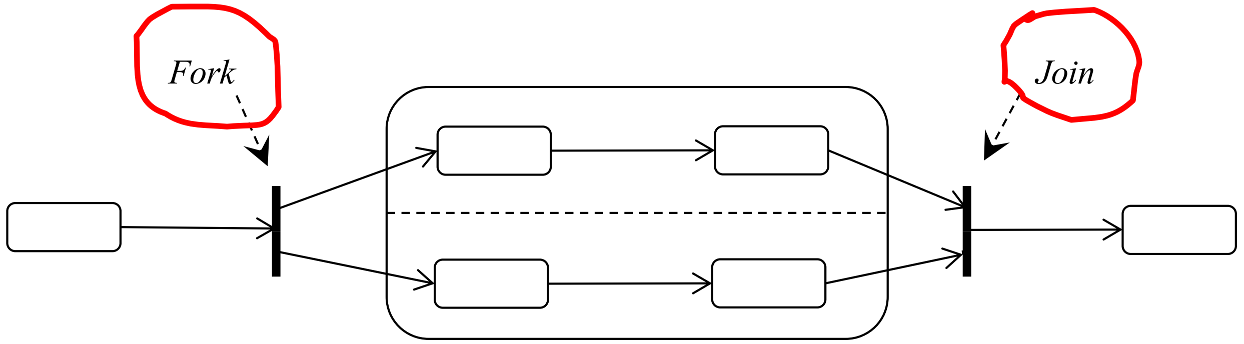
Concurrent States

- 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


Transition fired by completion event
State 1
State 2
someTrigger
Transition caused by the event someTrigger

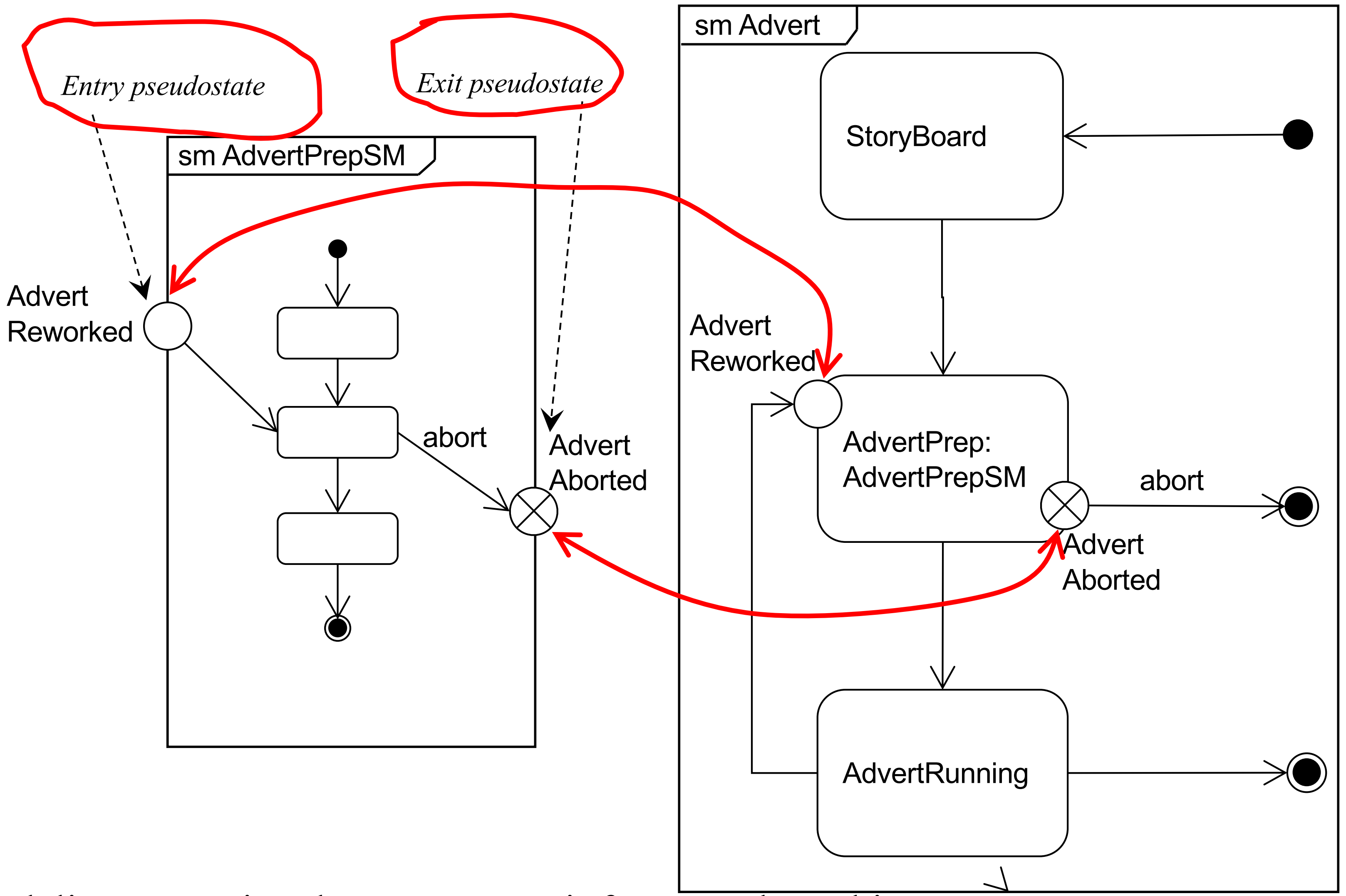
Synchronized Concurrent Threads.



•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.

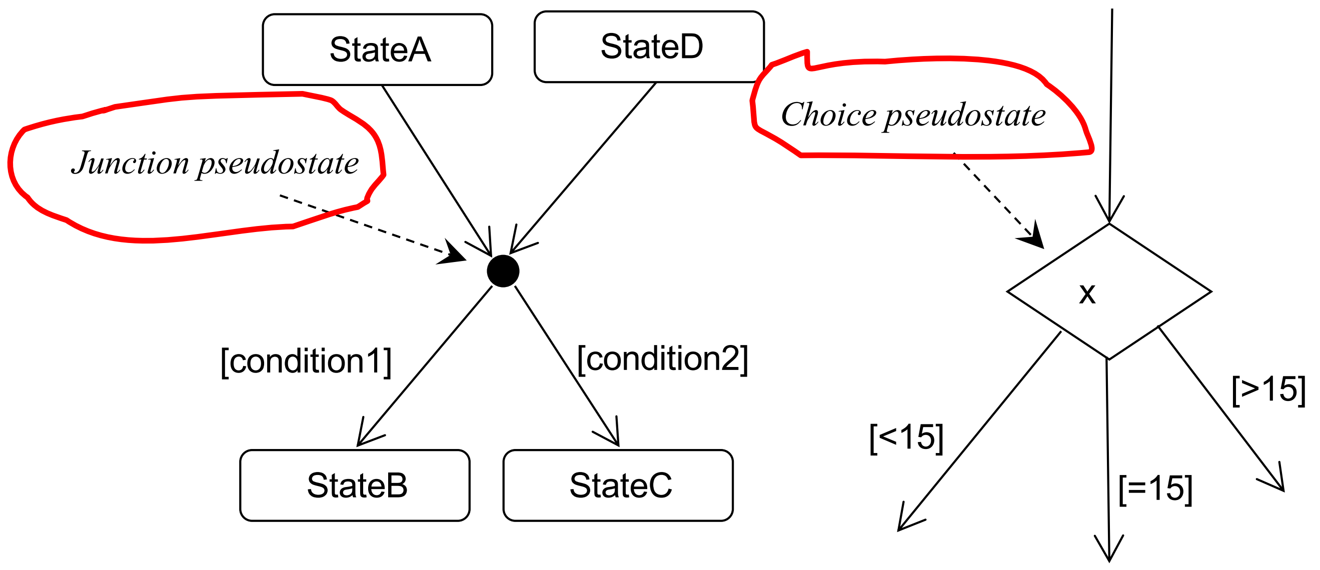
Entry & Exit Pseudostates



Used for modeling exceptional entry to or exit from a submachine state

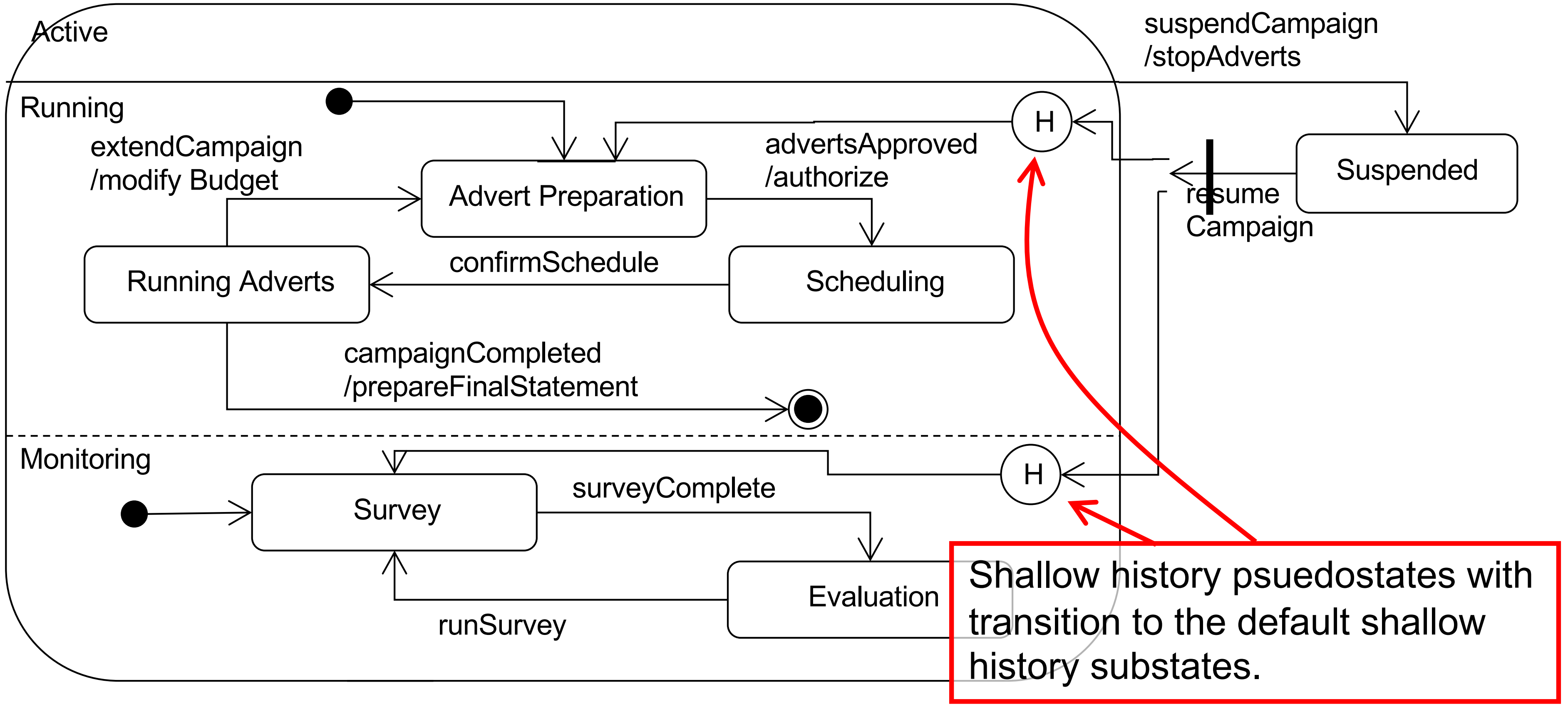
Can be shown in the frame-boundary OR inside the frame

Junction & Choice Pseudostates



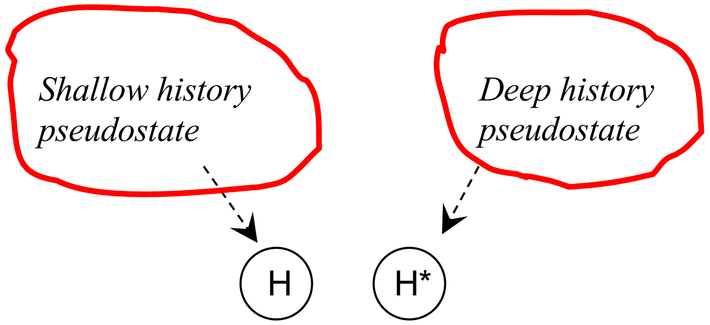
When there are many entry transitions and one exit this is known as a *Merge*
When there are several exit transitions and only one entry transition this is known as a *Static Conditional Branch*

History Pseudostates



Shallow history psuedostates with transition to the default shallow history substates.

History Pseudostates



Use deep history pseudostate if more than 1 substate

Preparing state machines

Behavioural Approach

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.

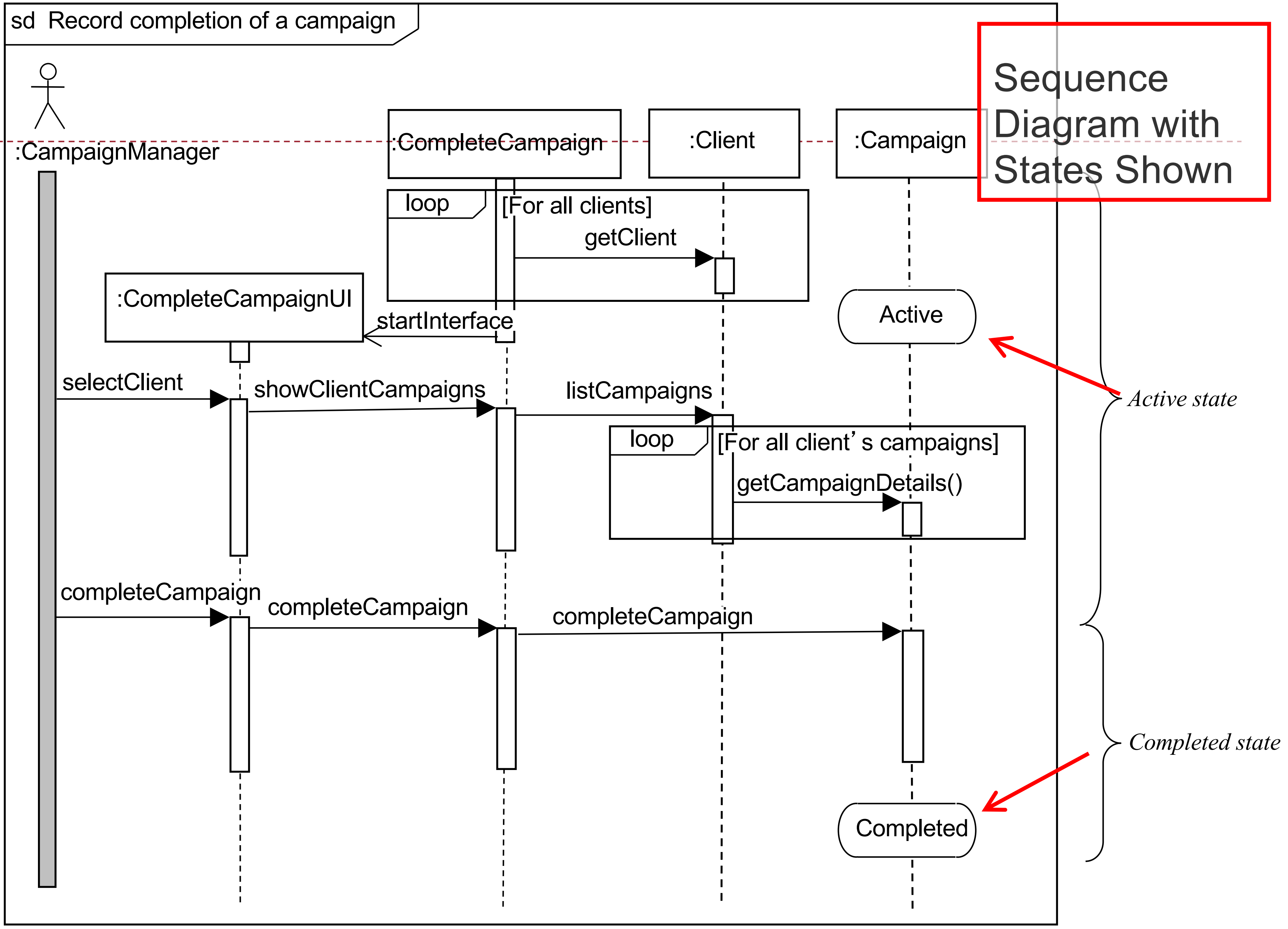
Behavioural Approach

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.

Behavioural Approach

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.

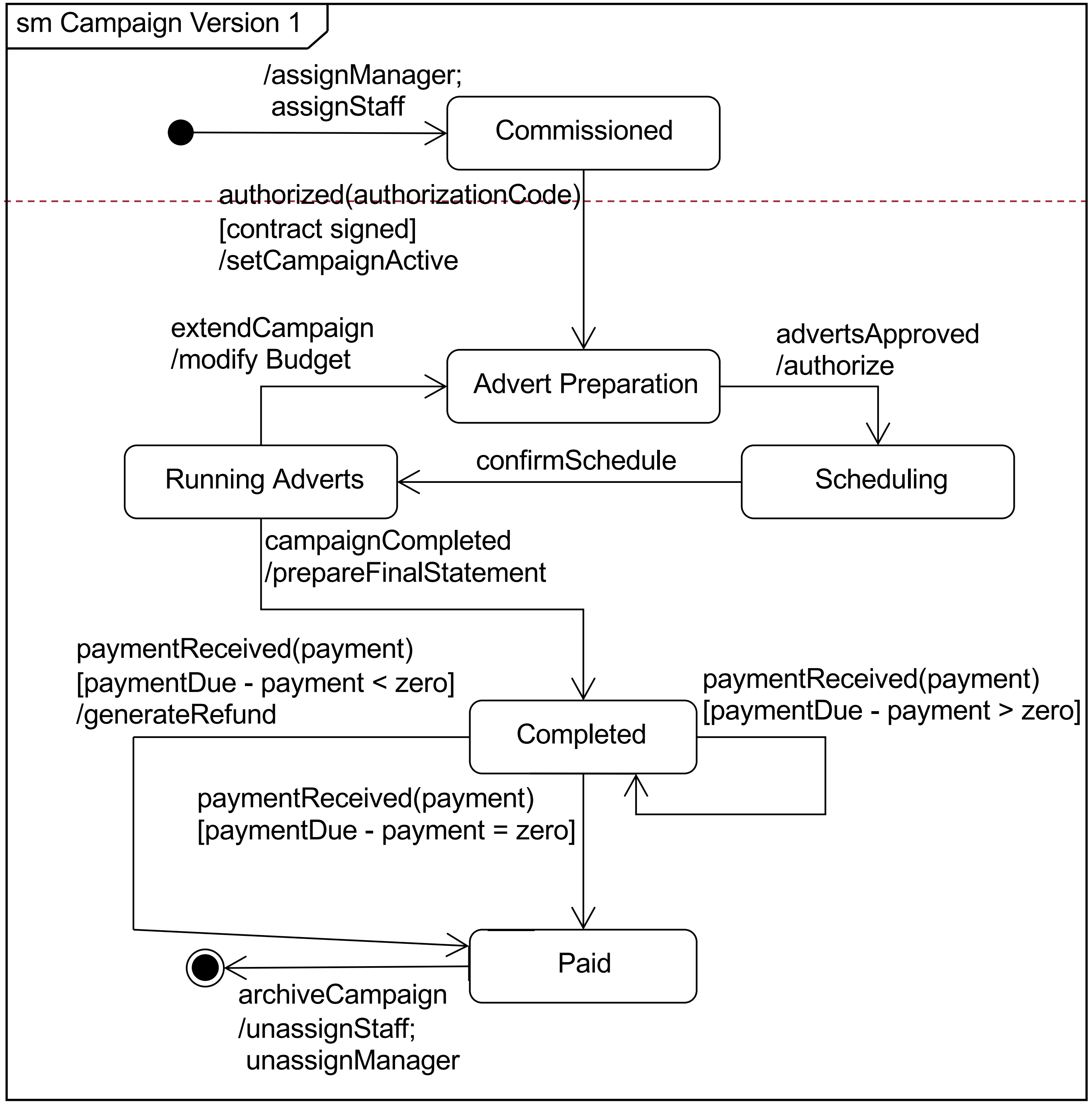
Sequence Diagram with States Shown

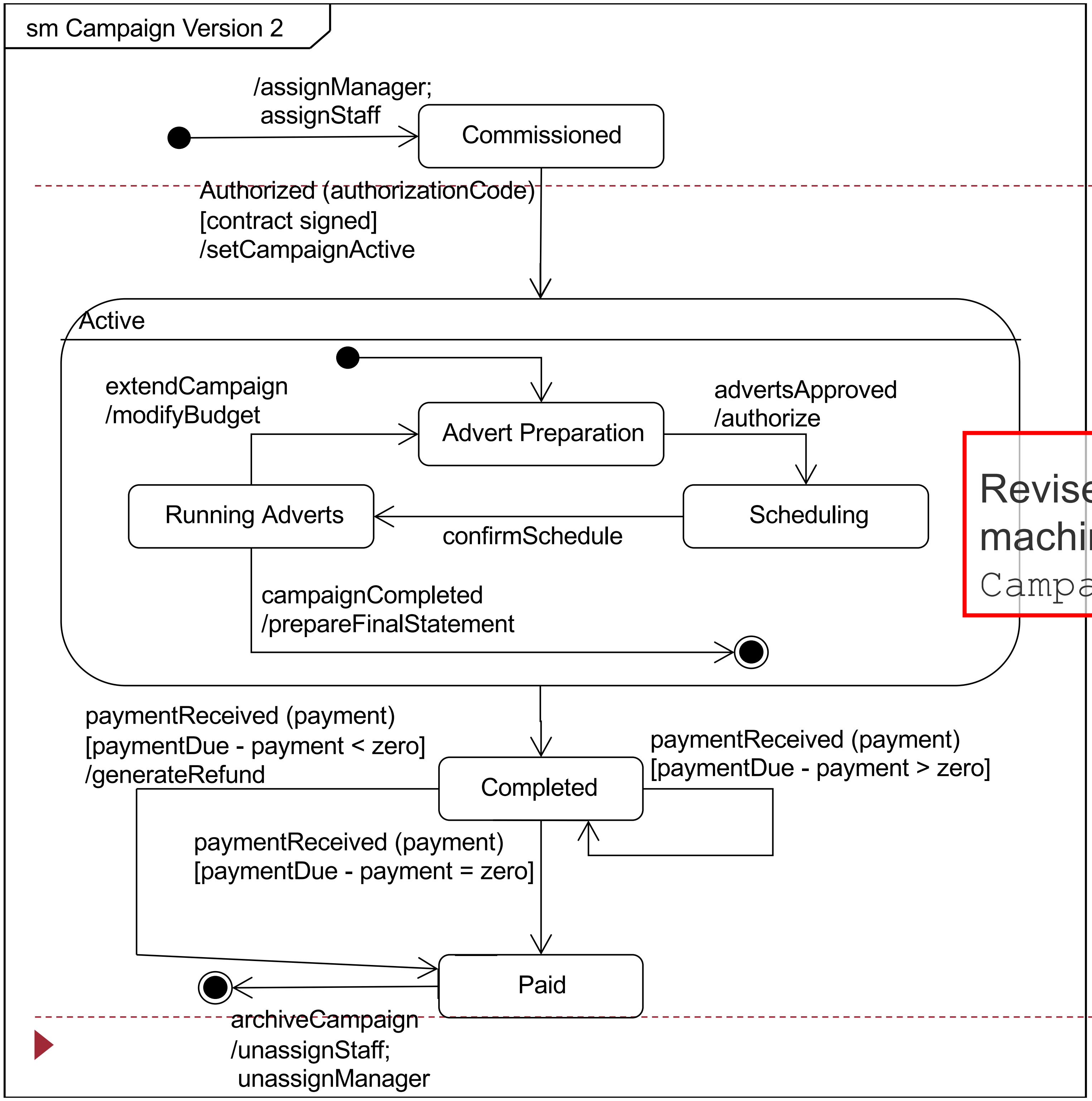


Behavioural approach example

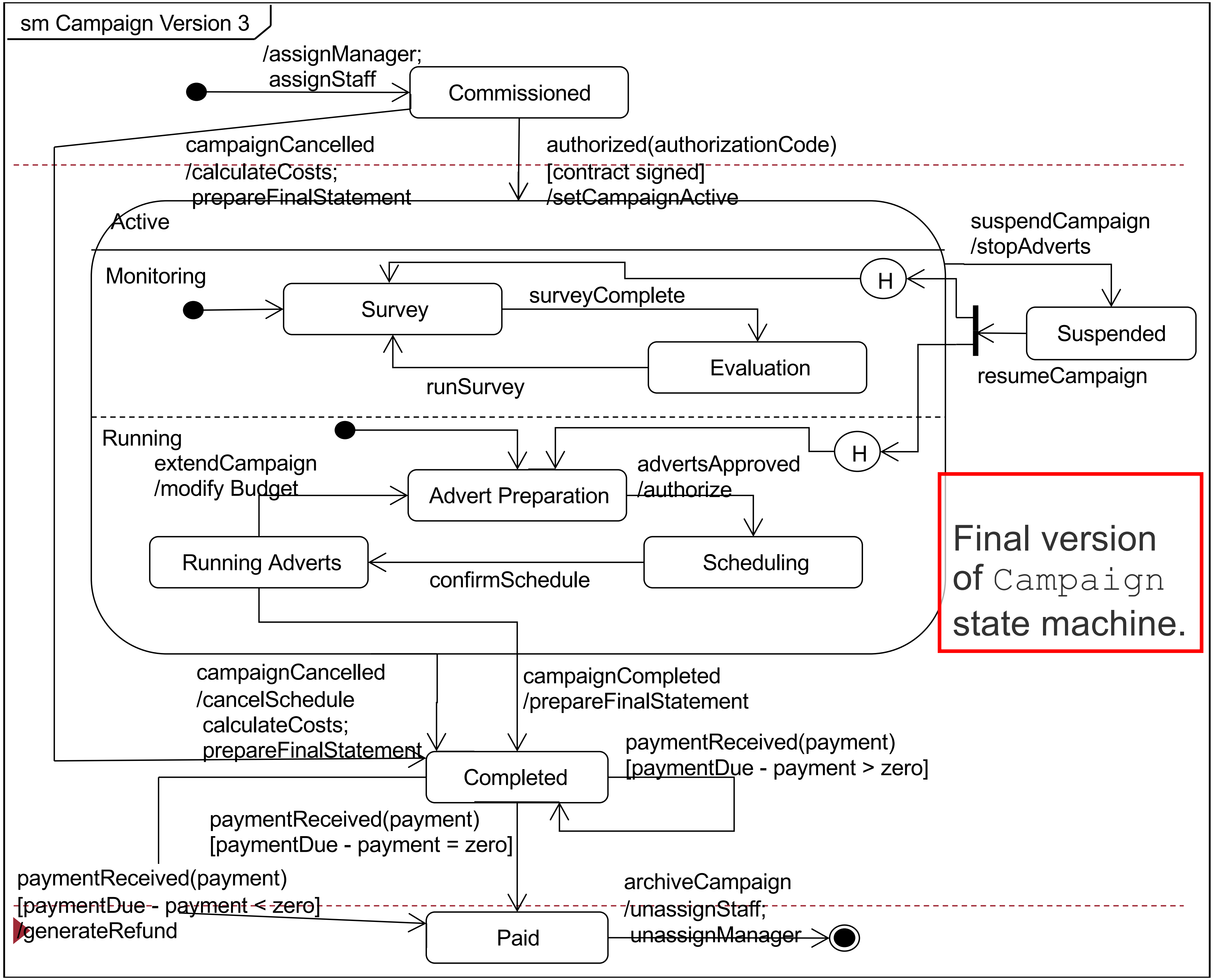
- The following slides show how a state machine might be developed
- More detail is added to the state machine as the analysis progresses
- Always start with an initial simple state machine and then elaborate from there

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




sm Campaign Version 2
/assignManager; assignStaff
Commissioned
Authorized (authorizationCode) [contract signed] /setCampaignActive
Active
extendCampaign /modifyBudget
Advert Preparation
advertsApproved /authorize
Running Adverts
confirmSchedule
Scheduling
campaignCompleted /prepareFinalStatement
paymentReceived (payment) [paymentDue - payment < zero] /generateRefund
Completed
paymentReceived (payment) [paymentDue - payment > zero]
paymentReceived (payment) [paymentDue - payment = zero]
Paid
archiveCampaign /unassignStaff; unassignManager

Revised state machine for the `Campaign` class.



Final version of `Campaign` state machine.

Consistency checking your state machine

- 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.

Summary

In this lecture you have learned about:

- how to identify requirements for control in an application;
- how to model object life cycles using state machines;
- how to develop state machine diagrams from interaction diagrams;
- how to model concurrent behaviour in an object;
- how to ensure consistency with other UML models.

References

- Chapter 11 Bennett
- UML 2.2 Superstructure Specification (OMG, 2009)
- Douglass B. P. (2004) Real-time UML: Advances in the UML for Real-time Systems (3rd Edition), Addison-Wesley.