



Modelling Object Lifecycles

The State Model

Objectives

- To understand the need for the state model
- Familiarisation with state transition diagrams (STDs)
- To understand states, events, transitions, actions and activities
- To be able to construct STDs from sequence diagrams
- To understand when to use STDs

The State Model

- ▶ An object is a *state machine*
 - ▶ One object has one lifecycle
 - ▶ It is born, it does stuff, it dies.

- ▶ A *state model* shows all possible lifecycles for all objects in a class:
 - ▶ What states an object instance can be in,
 - ▶ Which events it can respond to while in that state, and
 - ▶ How it responds to those events.

- ▶ Both represented using a *state diagram*
 - ▶ a.k.a. state transition diagram (STD)

All objects have a life cycle. After being created, an object will respond to events (for example, messages or replies received from other objects) and these responses might cause the object to change in some way. At the end of its life, the object is destroyed. Objects are sometimes referred to as *Finite State Machines* (FSM), or more simply as state machines.

Regardless of how short or long an object's life, it can have its life history represented by a State Transition Diagram (STD). The more interesting an object's life, the more likely we are to want to draw a STD to help us understand it.

Two or more instances of the same class might have different life histories. For instance, if you and your neighbour had identical models of DVD player, it is unlikely that you would use the controls at the same time or in the same order as each other. But the instruction manual would describe how to use the player at a general level – if the player is doing *this*, and you press *that*, this is *what* will happen, and this applies whether it is your player or next door's. In other words, an instruction manual applies to the whole *class* of DVD player. In the same way, a *state model* shows all possible lifecycles for a whole *class* of objects. A state model includes one or more STDs, with descriptions of each state and each activity an object can perform in its lifetime.

One way of thinking of a state model is as an instruction manual for any instance of a given class.

What is a State?

- A *state* represents the current mode or condition of an object
- An object is *stateful* if it can respond differently to the same message at different times
 - e.g. how a DVD player responds to a press of the 'play' button depends on what type of disk is in the tray (or if there is one!)
 - e.g. how a form on a web site responds to the 'Submit' button depends on what information has been entered
- Implemented using the values of an object's attributes
- An *event* can cause an object to change (*transition*) from one state to another

Some objects have relatively uninteresting lives – they might just hold some static information which they will give on request, but without undergoing any internal change as a result. Other objects lead very busy lives, constantly adapting their behaviour according to their circumstances – these are often described as *stateful*. It is important for an object to 'know' what state it is in, so that it can decide whether or not to respond to an event, and if it does respond, in what way.

An object's state is usually modelled by the values of one or more attributes – the class designer will usually decide how this is best implemented.

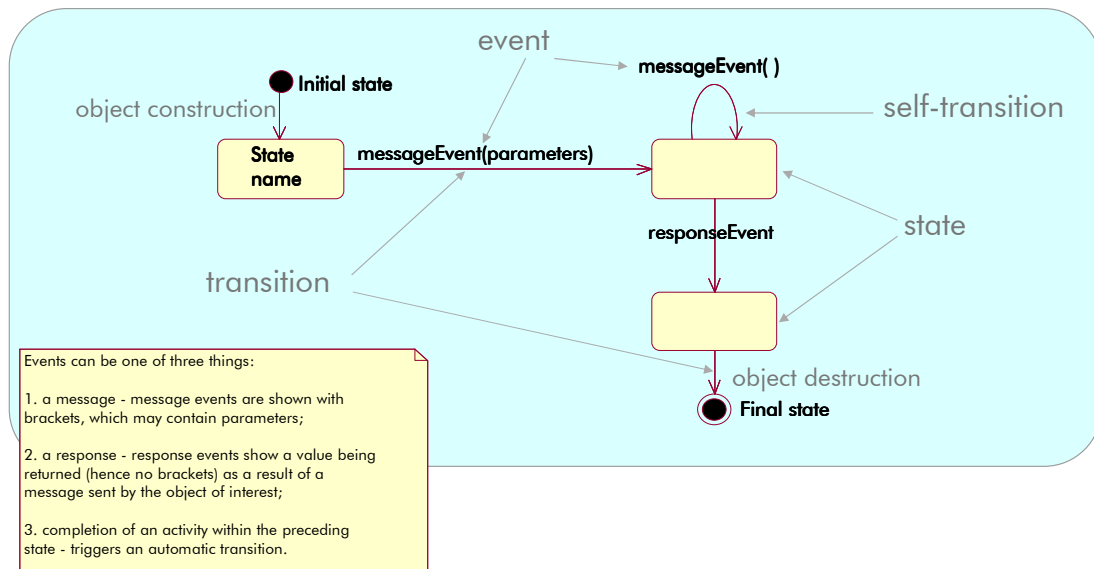
Another way of thinking of state is to consider it as the set of operations the object can perform at an instant in time. This will be a subset of the operations described by the object's class – those which are *not* available at this instant can be thought of as being 'greyed out', in the same way that a word processor's 'Cut' and 'Copy' options are greyed out if nothing is selected. If a 'Select' event occurs, the state of the system changes (or *transitions*) to a new state, in which 'Cut' and 'Copy' are now available in the 'Edit' menu.

State models are an important tool for the class designer, who must ensure a class provides attributes and operations to allow it to behave in the way it should to support the requirements of a system.

Business and Requirements Analysts can also use state models to represent the lifecycle of a whole system – a system is, after all, just a big object whose operations are regarded as *use cases*. At this level, a state can be regarded as a pre- or post-condition for a use case – the use case is the event which causes the system to transition from the precondition state to the postcondition state.

Basic State Diagram Notation

🎯 Notation devised by David Harel (1987)



The *initial state* represents the entry point of the diagram – where the object can be regarded to come into scope. Typically, this is used to show the state of the object on creation and usually there will be only one of these at a given level of diagram since classes tend to create their objects in the same state.

States are represented by 'roundangles' – rectangles with rounded corners. The state name is optional but useful when discussing a model.

Transitions are shown as arrows, and the event that causes the transition is written next to the arrow. Self-transitions can occur when an object's response to an event results in no change of state. Events can be

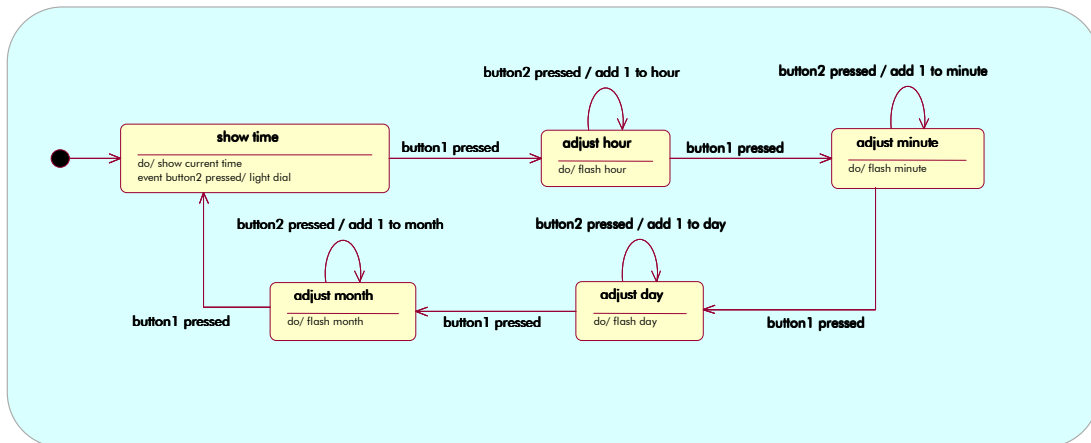
1. the reception of a message (shown with brackets, which might contain parameters);
2. The reception of a response to a message sent earlier by the object;
3. Completion of an activity. If an object is in a state which involves it *doing* something, the completion of that activity might cause the object to change state. E.g. if a VCR is in a 'rewinding' state, then completion of the rewind activity will cause the VCR to transition to a 'stopped' state (assuming automatic shut-off at tape end!).

The *final state*, shown by a bullseye, represents the point at which the object goes out of scope, e.g. object destruction. The preceding transition arrow will show the event that causes the object to move into the final state, and there might be more than one of these since objects might, for example, be destroyed under one of many circumstances.

This is not the full notation set for STDs, but can be regarded as the necessary minimum. Other aspects of the notation will be introduced later in the session.

Example State Diagram

A simple digital watch



This is a STD which could be used to show how to use a simple digital watch. The state of the watch could be thought of as 'what the watch face is showing' and the events are provided by pressing one of the buttons on the side of the watch casing.

Detail has been added to say something about what the watch does while in a particular state or in response to a particular event. Your instructor will talk through the model.

The Need for State Diagrams

- **An STD shows concisely and precisely how instances of a class behave dynamically**
 - which events they handle and when they handle them
 - how their response to events varies

- **The purpose of the STD is:**
 - To improve and share understanding of the class between users and developers
 - To identify the attributes and operations needed by a class in order to meet its responsibilities

- **State diagrams are only required for classes with significant dynamic behavior**

So, to summarise:

STDs model the dynamic behaviour of an object over its lifetime – what state they are in when they are created, how they respond to events throughout their life and finally, what causes their destruction. A STD could be drawn for every object and every class – in practice we tend to concentrate on only the classes whose objects have interesting (i.e. non-trivial) behaviour. STDs help the analyst understand more about how objects behave and help designers provide the class with the attributes and operations its instances will need in order to play their part in meeting the system requirements.

As we will see later, STDs also provide valuable input into the design of a test harness for each individual class.

State Diagram vs. Activity Diagram

Both represent *life histories*

An activity diagram shows the life history of a *process*

➤ Possibly involving many objects

➤ Activity defined by:

- what is being done (goal)
- by whom (swimlane)
- with what and (input object)
- what it produces (output object)



A state diagram shows the life history of an *object*

➤ Possibly involving many processes

➤ State defined by:

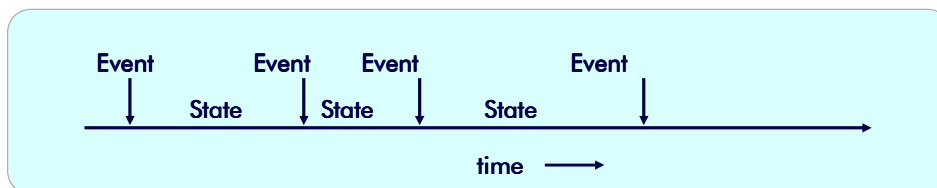
- set of attribute values for the object



It is easy to think of a STD as a flowchart – however, this is to misunderstand them. A flowchart represents the life history of a *process* – in UML we call them *Activity Diagrams*. Activity diagrams help us understand the sequence of tasks that achieve the goal of a process, together with the objects involved (as swimlanes and/or as objects manipulated during the process). In a State Diagram, however, the focus is on the life history of an *object*, which might in the course of its life participate in many processes.

The Duality of States and Events

- An event is *instantaneous*, a point in time
- A state is the interval between events
 - Objects are always in a state
 - An object remains in a particular state until an event which causes it to change state is received



- Between two states there is an event;
- Between two events there is a state.

When constructing STDs, it is helpful to bear in mind that objects spend all of their life in one state or another – i.e. states have *duration*. In other words, states can be interrupted. They are interrupted by events, which occur at instantaneous points of time.

For example, think of a telephone. It spends most of its time in a stand-by state – i.e. not ringing. A call comes in, and the phone transitions into a ringing state. How much time is there between the time it is *not* ringing and the time when it is? The two states abut each other – there is no gap between them.

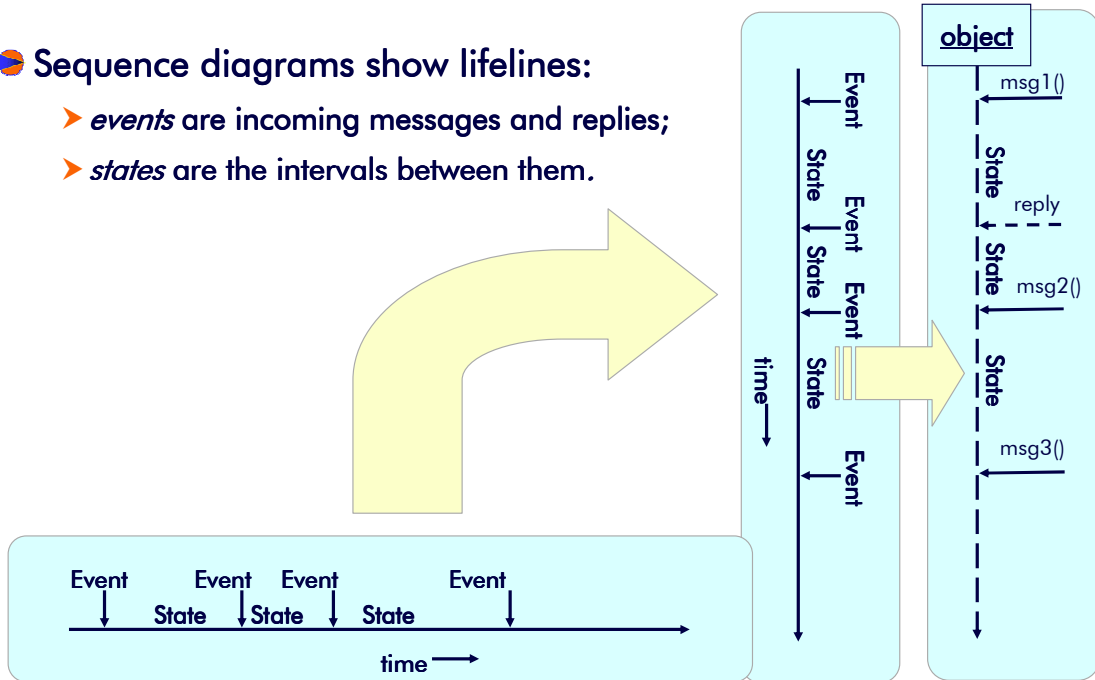
An event separates two states; a state separates two events.

It is this 'duality' that provides us with a useful technique for constructing STDs.

Building the State Diagram - 1

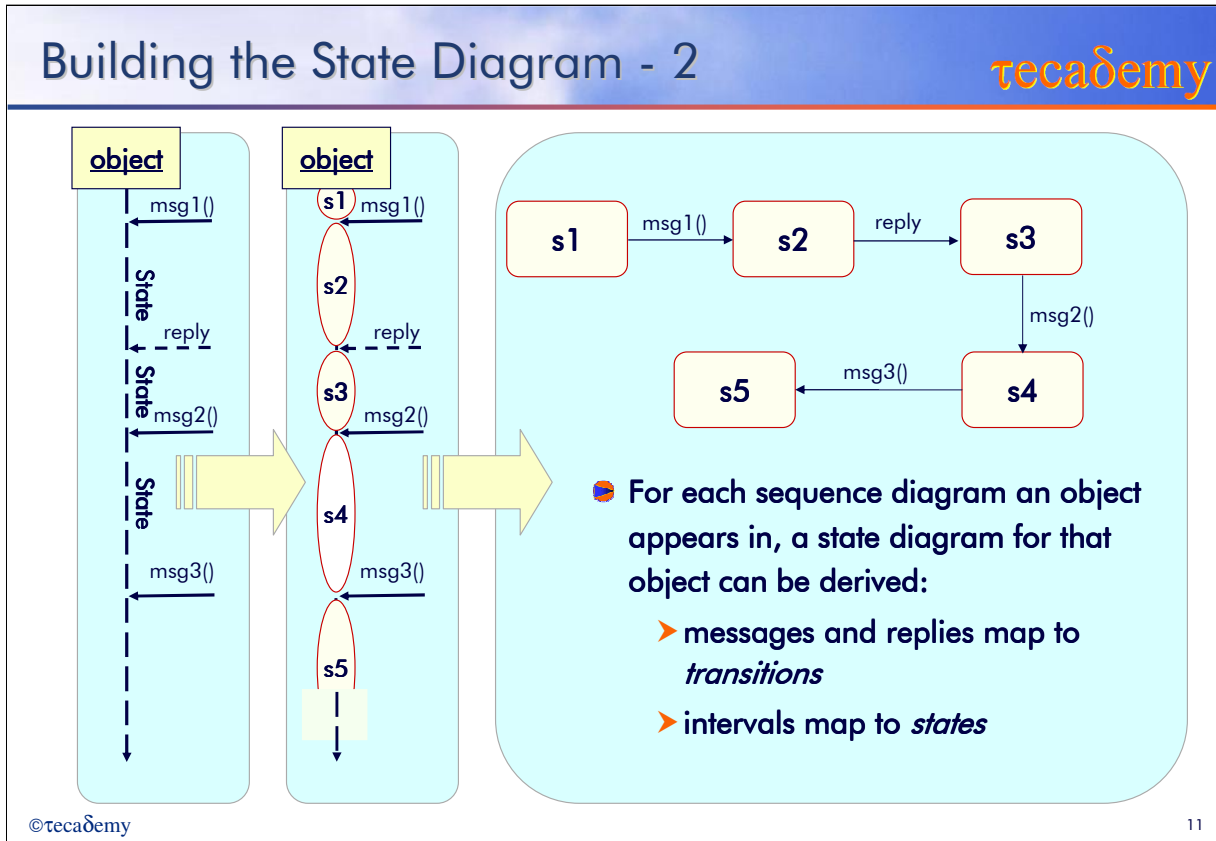
Sequence diagrams show lifelines:

- *events* are incoming messages and replies;
- *states* are the intervals between them.



Sequence diagrams are made up of object lifelines, arranged vertically, with the arrowheads showing each of the events an object receives in every scenario we have modelled. We can even see what kind of events they are, if the sequence diagram differentiates between messages (solid arrows) and responses (dashed arrows).

So, to construct the STD for a class, collect together every sequence diagram on which an instance of that class appears. Starting with the sequence diagram that represents the 'sunny day' scenario, isolate the lifeline of the object in question.



Using the 'duality principle', we can now identify the gaps between the arrowheads as states, and show it as a bubble on each side of an event. For now, don't worry about the half-bubbles at the top and bottom of a lifeline – treat them as single states.

Count the bubbles and begin your STD by drawing the same number of roundtangles, separated by transition arrows. Label each arrow with whatever is written on each event separating the bubbles on the sequence diagram.

Try to think of a name for each state. As a rule of thumb, good state names tend to be verb forms ending in *'-ing'*, to reflect the fact that objects in states tend to be *do-ing* something. You might get a clue by looking at any arrows *leaving* a bubble, as these can indicate activity; alternatively, look at the event that ends the state, because it could be that the object is simply *wait-ing* for that event to happen. You can always change state names as your understanding of them improves – in the meantime, it might help to number the states for reference regardless of the name you've given it.

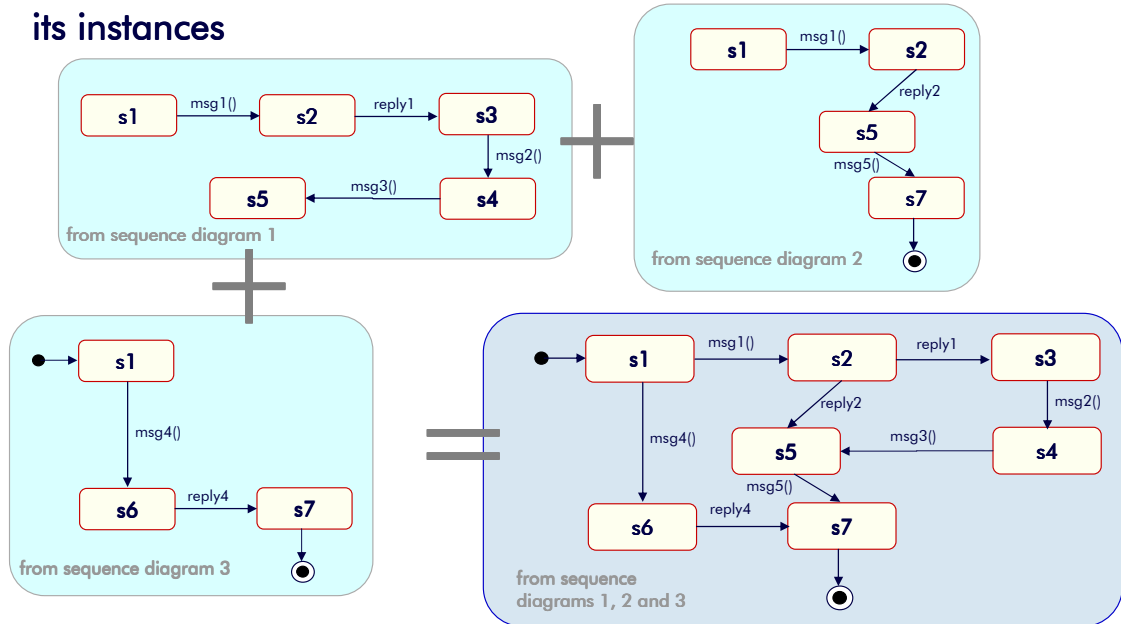
When you've finished, review the diagram and see that it makes sense. Check that you can map the STD back to the sequence diagram it came from. Watch out for duplicate states – sometimes two or more states that have different names turn out to be the same state. Two states are the same if you can apply the events leaving one state to the other, and end up transitioning to the same place.

If you find a problem, make sure you trace back to the sequence diagram/scenario/use case and re-work them to iron it out.

When you're happy you've done as much as you can, move on to the next sequence diagram...

Building the State Diagram - 3

● The state model for the class = the sum of the state diagrams for its instances



... and repeat the process. It is likely you will find the same states and events repeated on several diagrams. Any new states and new events can be identified and added to the STD so that, when all the sequence diagrams for a particular class of object have been processed, what you have is a STD that represents the sum of all of the individual life histories for each object of the class. This is the *State Model*.

The major advantage of this approach is that every state and every event can be traced back, via the sequence diagrams and scenarios, to one or more use cases. Problems discovered while building the state model can be traced back directly to the use case that needs to be re-worked, and helps to improve the correctness of the requirements model.

Exercise: Building a State Diagram

Objective

- To practise converting sequence diagrams into a STD

You are given:

- The following four slides show sequence diagrams for a user instructing a video recorder to perform the following tasks:
- Record and Record where the tape is write protected
- Play and Play allowing for user to press pause and fast forward

Task: Create a STD

- **Step 1:** For this exercise you are required to model the states of the VCR controller. This controller receives external events from a user and internal events from other VCR devices, then controls the rest of the system
- **Step 2:** Use the state model to identify possible missing events on the sequence diagrams. List the questions you would ask the customer.

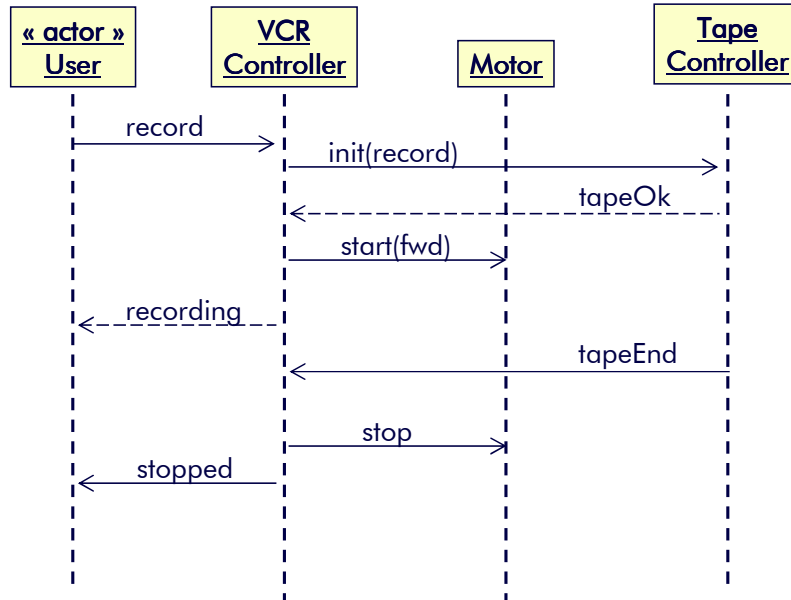
This exercise is based on a video cassette recorder (VCR). Our simple VCR comprises a Motor, Tape Controller and VCR Controller object.

For this exercise, we will model the states of the VCR controller object. This object accepts events from the an external user, which may be through some key pad interface or remote control, and also from the Tape Controller.

You will be presented with four sequence diagrams that all represent a different use of the VCR, but that all have the VCR Controller as a common object.

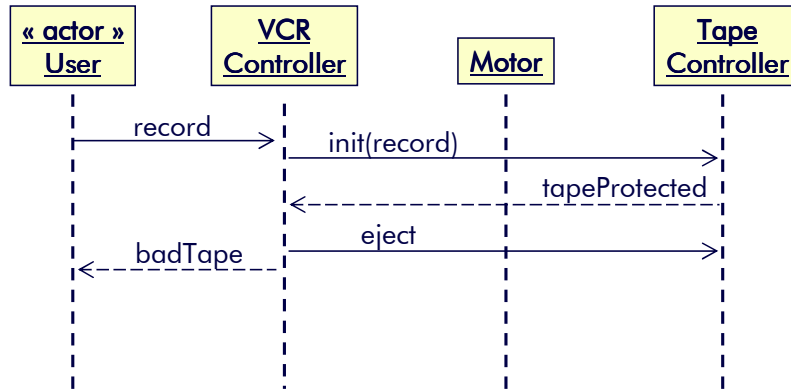
Having constructed the state transition diagram, it may become clear that there are potentially some missing events that are needed to make the VCR useable. Try to identify some of the missing events.

Exercise: Sequence Diagram 1



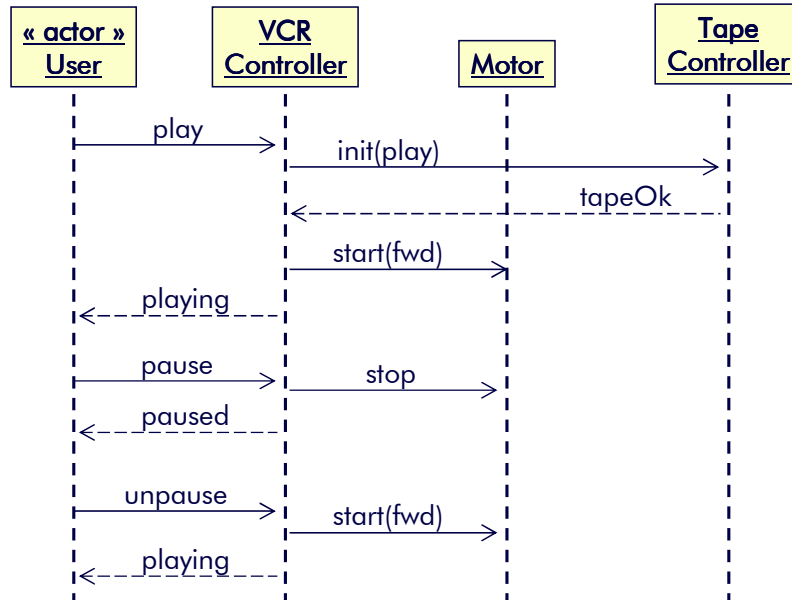
This sequence diagram is for basic recording.

Exercise: Sequence Diagram 2



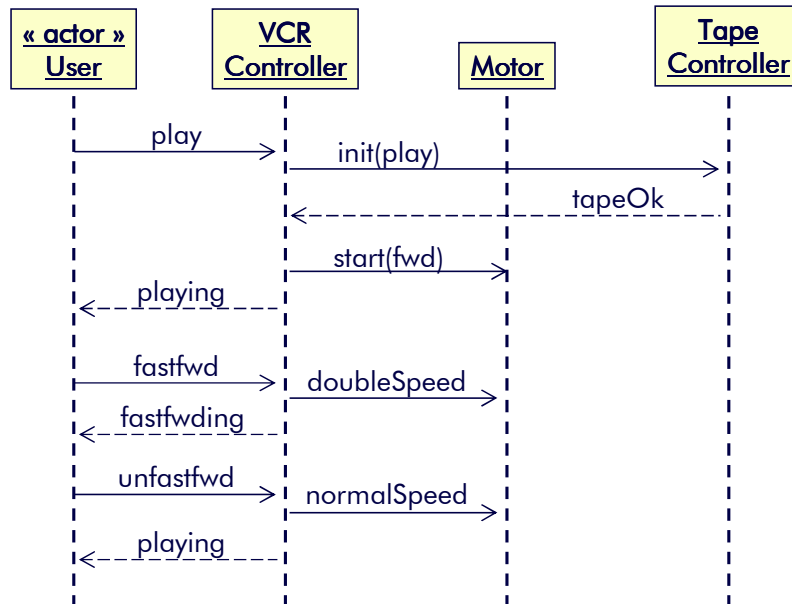
This sequence diagram is for recording, but where the tape is write protected to prevent it from being overwritten. The *Tape Controller* object is responsible for determining this information.

Exercise: Sequence Diagram 3



This sequence diagram is for playing the VCR. It shows a pause message stopping the motor, and an unpause message restoring it.

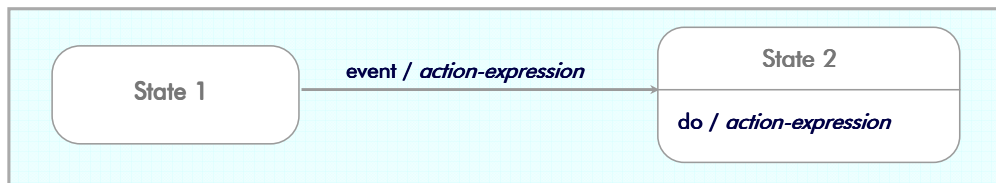
Exercise: Sequence Diagram 4



This sequence diagram is for playing the VCR. It shows a fast-forward message doubling the motor speed, and an unfast-forward message which returns the motor speed to normal.

Events and Behaviour

- When an event occurs, some behaviour may be performed as a result
- Behaviours are described by action-expressions

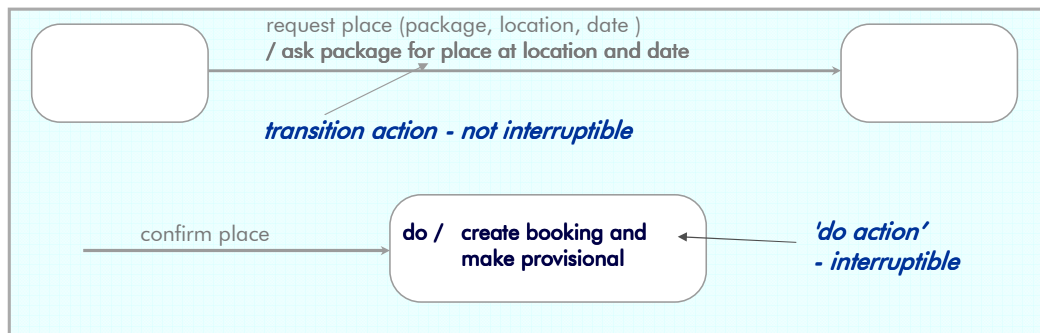


- An *action-expression* can be associated with the state transition and/or as an internal 'do' action on the new state

When an event occurs, some behaviour will be performed as a result. This behaviour is described in terms of *action-expressions*.

During the actual transition from one state to another, an *action* can be performed. When the object reaches its new state, an on-going action can also be performed.

Example: Action-Expressions



- *'Do actions'* can be considered to be instantaneous and are not interruptible
- *'Do activities'* take time to complete, may not finish and are interruptible

The main difference between *transition actions* and *'do' actions* is that a *transition action* is something that is fairly instantaneous and will not be interrupted, whereas a *'do' action* is something that is on-going while the object is in its current state.

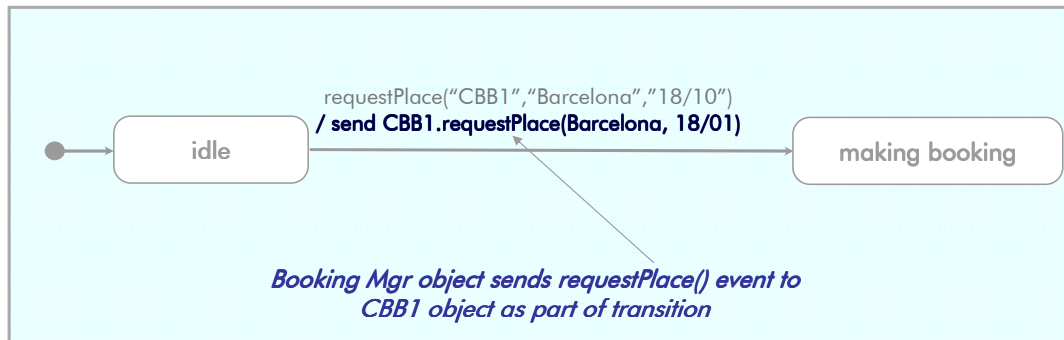
A *'do' action* may also not complete if it is interrupted by another event.

In the above example, a *withdraw* event will cause a message to be displayed before the object reaches its new state of *Process Withdrawal*. Once the object is in its new state, it will check the funds in the account, and maybe do several other tasks as well. It is possible that while it is performing these activities, the state could be interrupted by another event.

Note that during a self transition the do action is also interrupted and the action will be started again from the beginning.

Sending Events to Other Objects

- As part of a transition, or from within an action

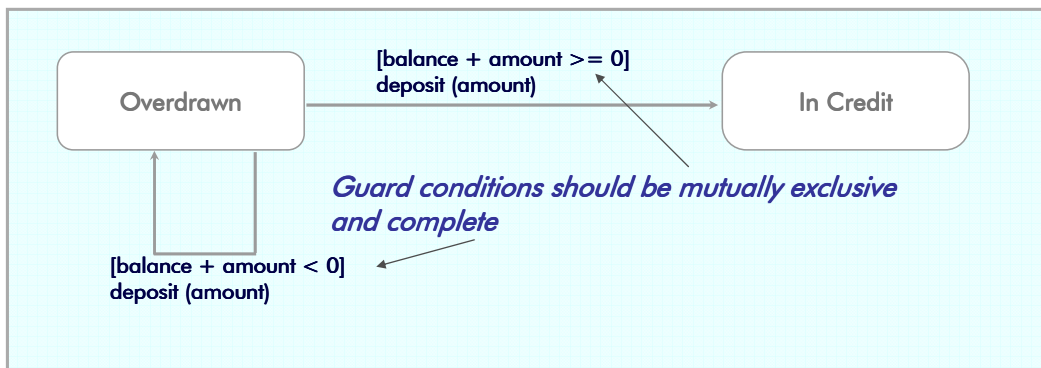


Objects can send events to other objects. In general, an event can be sent to any set of objects known to the sending object. Sending an event is an action that can be performed by an object, but UML provides a special syntax for it because it affects the flow of control so much.

The above slide shows the syntax for sending an event during a transition. It is also possible to send an event as part of a do action.

Guard Conditions

- A guard condition is a boolean expression that is used to determine which state transition should be performed when the event name is the same
- Syntax: [boolean expression] event name(parameters)



A guard condition is a boolean expression that is used to determine whether a state transition can occur. Guard conditions are represented on the transition(s) in question and refer to values of state variables in the controlling object.

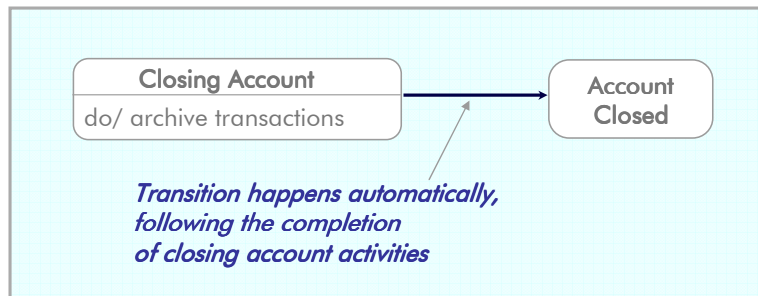
If an event occurs and the value of the expression is true, the transition occurs. Otherwise, it does not.

Guard conditions are most commonly used where the same event name can cause more than one transition from the same state, as shown in the above example. In this case, the guard conditions must be mutually exclusive and complete. In other words, only one guard condition can be true when the *deposit* event occurs.

Automatic Transitions

Automatic Transition

- Occurs automatically when the internal action of the current state is completed



Transitions can also occur automatically when a state's internal actions are completed, rather than having an explicit event cause the transition.

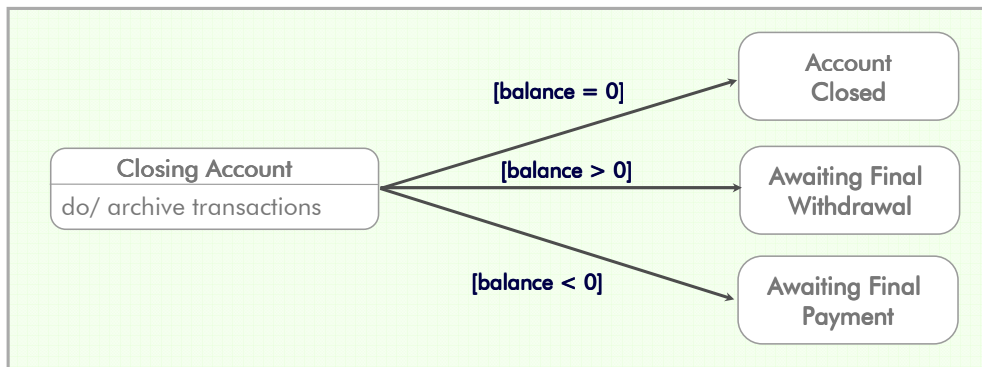
These automatic transitions are sometimes called lambda transitions, and are simply shown by excluding the event name from the transition.

Automatic transitions can also be combined with events on the same state. In this case, when the internal actions complete, the automatic transition will occur. However, if an event occurs before the internal actions have completed, the event will interrupt those actions.

By using only an automatic transition out of a state it will protect behaviour from being interrupted.

Guarded Automatic Transitions

- Automatic transitions can also be used with guard conditions
- One transition may be triggered when the internal action is complete, if the condition is true



Automatic transitions can also be used in conjunction with guard conditions.

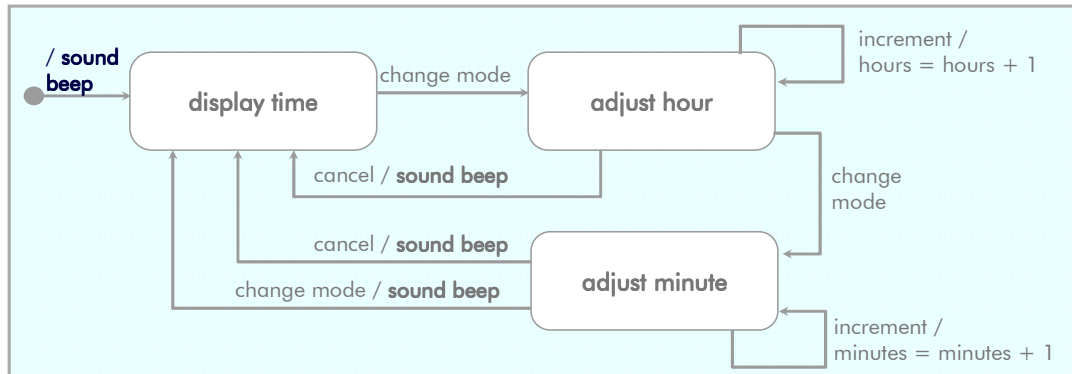
If a state has an automatic transition but none of the guard conditions are satisfied, the state remains active until one of the conditions is satisfied or until an event causes another transition to fire.

In the above example, when the internal actions have completed for *Closing Account*, an automatic transition will occur, which will take the state to one of three new states, depending on the value of the *balance* attribute.

Remember that guard conditions must be mutually exclusive, as we saw earlier. Therefore, only one of the guard conditions can evaluate to true on completion of the state.

Entry/Exit Actions

- Actions often need to be performed every time an object enters or leaves a particular state, rather than for a specific event



When every transition to a particular state requires the same action to be performed, it is convenient to record this as an action on the state rather than on the transition itself. This is called an *entry* action.

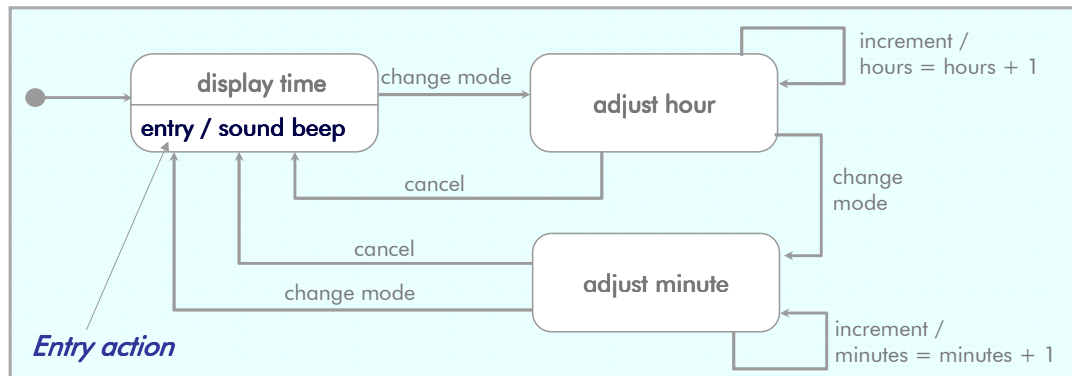
In the above example, all the transitions to Display Time require a *sound beep* action to be performed. We will see on the next slide that this can be moved to an *entry* action on the Display Time state.

It should be noted that an entry action means that every transition will cause this action to be performed. If this is not what is wanted, the actions should be left on the transitions that actually need it.

An *exit* action is a complementary action that is performed immediately before an object leaves a state. Later on in this chapter, we will summarise the order in which actions are fired.

Example: Entry Actions

- Useful when several events can cause a transition to the same state and require the same action to be performed

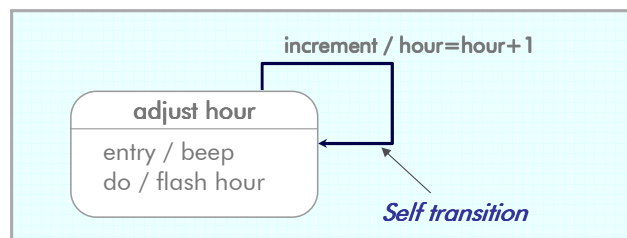


The *sound beep* action has been moved from the individual transitions to the *entry* action on the *Display Time* state.

Sound Beep will now be performed whenever this state is reached and will therefore also mean that the initial state will also cause this action to be fired when it transitions to *Display Time*.

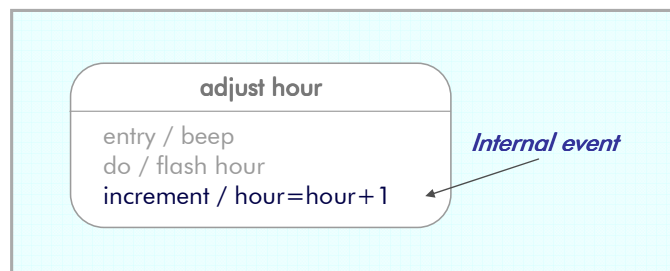
Self Transitions

- Are an instantaneous transition, object changes back into the state just exited
- Regarded as an interrupt – ‘do’ actions stop, ‘exit’ actions are fired, transition occurs, ‘entry’ actions fires, ‘do’ action re-starts



Internal Events

- Event occurs but does not result in state change
- *Not* regarded as an interrupt
 - event action occurs without stopping 'do' action
 - 'exit' and 'entry' actions are not fired



An event can cause an action to be performed without causing a state change. The event name is written inside the state box followed by a "/" and the name of the action.

When such an event occurs, its action is executed, but not the *entries* or *exits* for the state.

Note: Keywords *entry*, *exit* and *do* are reserved words within the state box.

Summary

- A state diagram graphically depicts the relationship between the states and events, states and activities, events and actions
- Actions can be represented:
 - On transitions as *event / action*
 - On entry to a state as *entry / action*
 - On exit from a state as *exit / action*
 - Within a state as *event / action*
- Activities are represented within a state as *do / activity*.
- Transitions may be automatic, self, and have guard conditions
- An STD enables us to determine if the behaviour of the class is cohesive

Case Study – Exercise 10

 **Objective**

➤ To practice drawing State Diagrams.

 Turn to and complete exercise 10 in the exercise booklet.

 Notes: