Lecture 6

Object-Oriented Software Engineering: Design Model - 2

Dr Neil Maiden
Dr Stephen Morris
Dr Wolfgang Emmerich

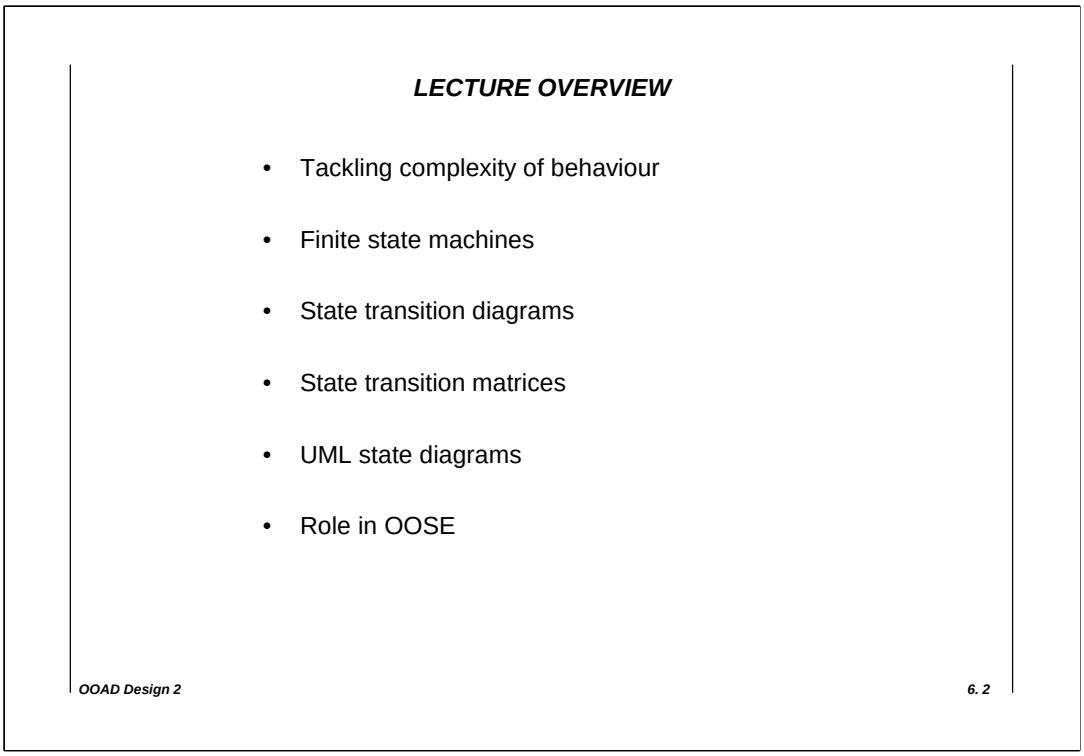
School of Informatics
City University

Last week, we have discussed the development of a class hierarchy that reflects constraints imposed by the implementation environment. The class diagram, therefore, has to be adjusted to take the programming language, the target operating system(s), the underlying database management system, the user interface management system, the available integration mechanism(s) and the development process into account.

We then indicated how sequence diagrams are used in order to design the control flow and the ordering of messages that are exchanged between objects in order to invoke operations specified in other classes. The sequence diagrams were then used to elaborate the class diagram with operations that were not existing before.

So far, we have only focussed on the static export interface of classes. We have not considered the operations that are internal to the class nor have we addressed the behaviour of instances of the class in response to the receipt of messages. Very often certain orders have to be enforced in order for an object to behave properly. Sequence diagrams represent particular orders for certain scenarios. These orders now need to be specified at the class level. A file, for instance, has to be opened before it can be accessed or modified and only if it is closed at the end will the changes be properly stored.

We are going to take these considerations into account and model the states that classes can be in. The question that we aim to answer in this lecture is, therefore: *How can we model the complex dynamic behaviour internal to classes?*



After a short indication where we are on our road map, we will discuss how complex behavioural specifications can get. We will have to keep this in mind when we devise solutions to the problem of specifying the class behaviour.

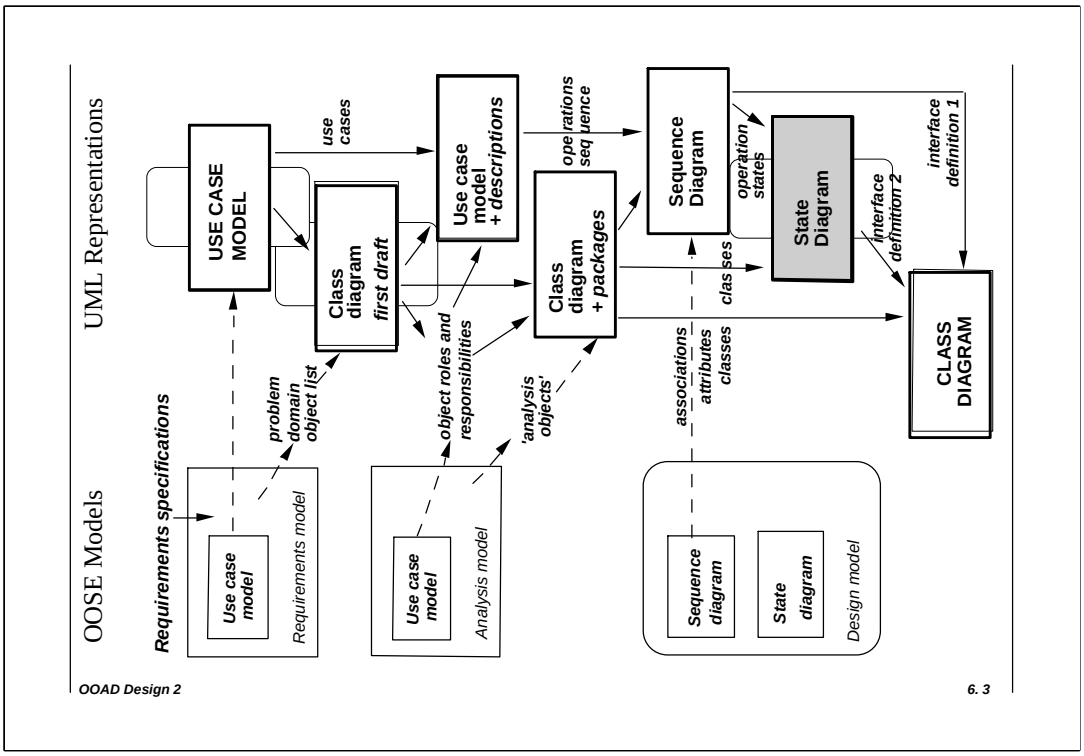
A technique for behavioural specification that has been around for very long are *finite state machines*. Finite state machines have been used for language specification (regular languages) and they are deployed for that purpose in every compiler. While they are fully appropriate for machine understanding they do not properly cope with abstraction and tend to get very complex.

Finite state machines are a mathematical concept and they have a graphical as well as a tabular notation. State transition diagrams depict finite state machines in a graphical form while state transition matrixes are a tabular representation.

David Harel has investigated the reduction of this complexity and has come up with the concept of *state charts* [Hare82]. Any finite state machine can be represented as a state chart that is considerably less complex than a state transition diagram.

The Unified Modelling Language provides language mechanisms for state charts. In UML Harel's state charts are referred to as *state diagrams*. tate charts have the same expressive power as finite state machine a more concise for finite state machines that is much more concise. We will spend most of this week's lecture on the definition of these state diagrams and discuss the pragmatics involved in their application.

OOSE incorporates a particular variant of state charts called ‘state transition graphs’, aimed in particular at aiding the transition from design to code.



Last week, we have discussed the notion of sequence diagrams and indicated how they are derived from class diagrams and use case model descriptions. The sequence diagrams had already an influence on the class diagram and we have seen how they are used to find the need of operations that we had not seen before.

During the course of this week's lecture we are going to introduce state diagrams. The purpose of these state diagrams is to determine the relevant internal states that objects that are instances of particular classes can have, the transitions that are allowed between these states and the events that cause the transitions to happen. Hence we will produce a state diagram for each class identified in the class diagram.

The modelling of these states might again lead to modifications of the class diagram. While the sequence diagrams led to the introduction of new operations that are publicly available from a class and therefore contained in the (export) interface of the class, state diagrams will most likely identify operations that are internal to a class and that will be used during the implementation of the exported operations. State diagrams will also allow us to carefully re-consider the definition of attributes as attributes have to be available to reflect the definition of the states that are identified for a class in the state diagram of that class.

Design elements already considered

Design outputs so far :

- set of ***sequence diagrams*** [diagram x use case]
- elaborated ***class diagram***

Next step: Change perspective from `outside´ to `inside´ of class

- Need for further understanding of each class
- Development of ***state diagrams*** to define class behaviour

OOAD Design 2 *6. 4*

The elements of the design that we have considered already are :

a) the set of sequence diagrams, each representing the temporal interaction of all the objects in a single scenario, i.e an instance of a particular use case, and,

b) the elaborations made to the class diagram, in the form of operations derived from the sequenced diagrams and incorporated into the interface of each class.

In preparing the next steps, we need to consider both the ‘system in use’ (as represented in the various sequence diagrams) and the ‘objects in the system’ and how each will evolve in response to external stimuli.

State diagrams (the subject of the this lecture) provide the essential means of describing the dynamic behaviour of a class, via the temporal evolution of an object in response to interactions with other objects inside or outside the system.

The state diagrams are a mathematical well defined language. They are based on the concept of finite state machines. Hence, we are going to introduce this concept first...

TACKLING COMPLEXITY

A system containing four buttons (B1 - B4) and two lights (L1 - L2)

10. Since the last powering on, if B2 has been pushed more often than B3, then L1 shall be lit.
11. Since the last powering on, if B2 has not been pushed more often than B3, then L2 shall be lit.
12. At no time shall more than one light be lit.
13. If either light bulb burns out, the other bulb shall flash on and off in 2-second increments regardless of the number of B2 and B3 presses. This flashing shall cease when B4 is pressed and restart when B1 is pressed. When the malfunctioning bulb is replaced, the bulb shall cease to flash, and the system shall return to its normal operation.

What is normal operation, if we don't know whether the system records B2 and B3 presses while a bulb is broken ?

OOAD Design 2 *6. 5*

This example and other material for this lecture have been taken from [Davis90].

The problem is that with the specification techniques that we have seen so far we cannot unambigously specify the behaviour of the system. Scenarios are not suitable as we might have to include a very high number of scenarios in order to completely describe the system.

Such questions often arise in dynamic or reactive systems in which input data continues to arrive during processing to effect the program's outcome, so-called 'real-time' systems.

Moreover, use cases are described in natural language (usually English in this country) and the use of a natural language often leaves room for different interpretations. This is particularly inappropropriate in situations where the behaviour of the system must be specified very precisely (think of the bulbs and buttons as part of an aircraft control panel). In these, so called safety-critical systems, unprecisely specified behaviour risks human lifes.

What means are their for specifying and representing behaviour precisely?

SPECIFYING COMPLEX BEHAVIOURS

Need to specify formally behaviour of :

- ***system*** in ***use***
- ***objects*** in ***system***

OOSE offers techniques

- use case diagrams, interaction diagram,
- state transition diagram

UML provides notations

- use case model, sequence diagram,
- collaboration diagram, state diagram

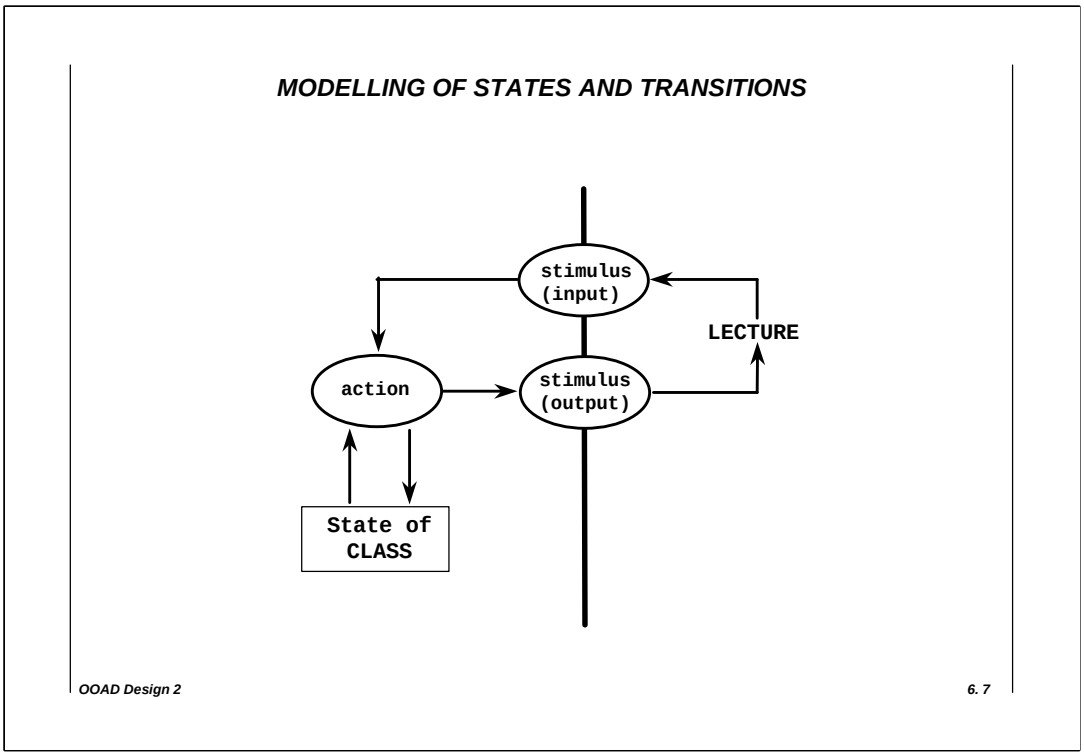
State diagrams provide essential means of showing how a class of objects evolves in response to external stimuli

Both use cases and sequence diagrams (called interaction diagrams in OOSE) are concerned with the system in use. They display the flow of messages according to different scenarios, potentially involving a number of objects.

UML also provides a ‘collaboration diagram’ which shows interaction between a set of objects as nodes in a graph, thus emphasizing relationships rather than temporal flow of behaviour shown in the sequence diagram. This has not been included because firstly it has no counterpart in OOSE and secondly the purpose of collaboration diagrams can equally well be met through sequence diagrams.

State diagrams come in various forms and provide powerful tools for the design of the behaviour in complex systems.

The concept of the ‘finite state machine’ underlies all state diagrams and provides the theoretical means of defining the state of a system and its reactions to new stimuli.



A finite state machine is a hypthetical machine which allows, for example, the modelling of the behaviour of this class in the context of this lecture.

The lecturer provides a series of inputs, causing actions of some kind of action in the class, which then generates an output and, if the lecture is effective, causes a permanent change in its state.

Let us know consider the mathematical definition of finite state machines...

FINITE STATE MACHINES

- A finite state machine FSM is a five-tuple FSM=(S,A,σ, s, F) where
  - i) S, is a finite set of states
  - ii) A is a finite alphabet of events
  - iii) σ: S×A $\rightarrow$ S, the partial function determining transitions
  - iv) s $\in$ S, a start state
  - v) F $\subseteq$ S, a set of ending states
- A configuration of an FSM is an element of SxA*
- If FSM=(S,A,σ,s,F) is a finite state machine then
  - i) A binary relation Γ is defined on configurations by (q,w) Γ (q',w') $\Leftrightarrow \exists \alpha \in$ A : (w=αw') $\wedge$ (σ(q,α)=q')
  - ii) Γ^* defines the transitive closure of Γ.
  - iii) A sequence of events is acceptable by the finite state machine if there is an ending state f $\in$ F such that (s,w) Γ^* (f,ε).

OOAD Design 2 *6. 8*

A finite state machine (FSM) includes a finite set of states (S) one particular element of that set is a starting state (s) and a subset F of S is designated as the set of ending states. Finite state machines in general work on alphabets of characters. For the purpose of this lecture, we can consider these alphabets to denote a finite set of events. The core of any FSM then is the transition function that defines for a pair consisting of a state and an event the successor state.

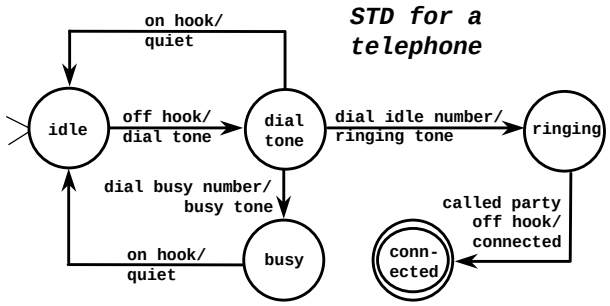
For the defnition of the semantics of an FSM machine, we need the concept of a configuration, which denotes the current state and a sequence of events that remain to be processed.

The relation Γ gives the semantics to the FSM. It defines a single step of execution. If the FSM is currently in state q and event α is the next event that is to be processed the new state will be q' if σ(q,α)=q'. The transitive closure of Γ denotes the set of reachable states and a sequence of events is acceptable only if the state the application of the sequence of events to the start state leads to an ending state.

This notation for finite state machines is mathematically sound. However, it is not an appropriate notation for humans to understand these machines. We will now use state transition diagrams and state transition matrices as notations whose semantics is formally defined based on finite state machines...

STATE TRANSITION DIAGRAM

circle	=	state
double circle	=	finishing state
directed arc	=	potential to make transition between the two connected states
label	=	input triggering transition / output from system



OOAD Design 2 6. 9

State transition diagrams are a direct representation of finite state machines. States are represented as circles and labelled arrows between states denote that the function σ is defined for the state where the arrow starts and produces the state where the arrow leads to if the event occurs that is recognised by the label.

All ending states are denoted as a double circle and the starting state is denoted as a circle where an open arrow head leads to (idle in the example above). Note, that there are also special forms of finite state machines that produce an output whenever a transition occurs. Outputs, if any, are given after the event definition and the two are separated by a slash.

The example displays states for a telephone. Initially, the telephone is in an idle state. If the receiver is taken off the hook a dial tone will be played. If the receiver is replaced the telephone will become idle again. If a number is dialled of a phone that is busy, the busy tone will be played and the only possible event is to replace the receiver onto the hook. If an idle number is dialed the ringing tone will be played and as soon as the other party takes the receiver off the hook the phone connection will be established.

An alternative notation to state transition diagrams are state transition matrices...

STATE TRANISITION MATRIX

A matrix with :

- all possible ***states*** labelling ***rows***
- all possible ***events*** labelling ***columns***
- cells identify ***next states***
- ***responses*** in separate column

EVENT / STATE	on hook	off hook	dial busy	dial idle	called party off hook	OUTPUT
idle		dial tone				quiet
dial tone	idle		busy	ringing		dial tone
busy	idle					busy tone
ringing					con-nected	rin-ging
connected						con-nected

OOAD Design 2 *6. 10*

State transition matrices are an alternative representation for finite state machines. In these matrices all possible states label the rows and all events that can occur label columns. For each row, the cells identify the successor state into which the machine transits if the event occurs that identifies the column of the cell.

In the example, the transition from 'state' that is triggered by event 'off hook' to state 'dial tone' is displayed as the first cell of that table.

Cells in the output column denote the output that occurs when a transition is done to the state identified by the particular row. There is an implicit assumption in this representation (and a weakness associated to it) in that it assumes that all transitions leading to one state have the same output.

For the specification of the behaviour of complex systems, however, neither state transition diagrams nor state transition matrices are appropriate...

MOTIVATION OF STATE DIAGRAMS

Realistic systems modelled with STDs get ***very complex.***

Reasons of complexity:

- System as modelled by STD can only be in one state
- State is influenced by ***many factors***
- All factors need to be considered leading to exponential proliferation of states and transitions

Harel's State Charts: concepts to make complexity manageable:

- Composition of states
- Concurrent substates
- Conditional transitions
- History states

OOAD Design 2 *6. 11*

If any realistic system is modelled with state transition diagrams these diagrams get **very** complex. The reason for this complexity is that finite state machines have exactly one active state and that state has to be explicitly modelled in the respective state transition diagram.

In reality, however, states can be influenced by a number of different factors. A telephone receiver can be either idle or it can be active. If it is active, it can be playing a busy tone, a ringing tone or a dial tone. State transition diagrams do not properly support the different levels of abstractions involved in this example.

This leads to an exponential growth in the number of states and transitions needed and makes the resulting diagrams unmanageable.

David Harel had to formally define the behaviour of a fighter aircraft component (on behalf of the Isralian air force). He found that engineers and pilots could easily understand state transition diagrams and searched for a way to cope with their complexity rather than introducing a completely different formalism nobody would be familiar with. The approach he took is, in fact, based on the same that we suggested in the first lecture: abstraction.

In the state charts that he suggested, he introduced facilities for considering states at different levels of abstraction. He then introduced different notions for composing abstract states from more concrete states which might have internal state transitions embedded. These internal state transitions, however, would be hidden at the more abstract level; hence he applied the principle of information hiding also to modelling of states.

The UML includes state diagrams as a notation for state charts. State diagrams will be used to model the behaviour of objects and in particular the state transitions of objects in response to events, internal or external to the object.

STATE DIAGRAMS

A ***state diagram*** is

- a ***directed graph*** of states connected by transitions
- a ***formal specification*** of the behaviour of a class

UML incorporates extensions to basic STDs made by Harel :

- decomposition of states
- default entry states
- concurrent states
- conditions on transitions

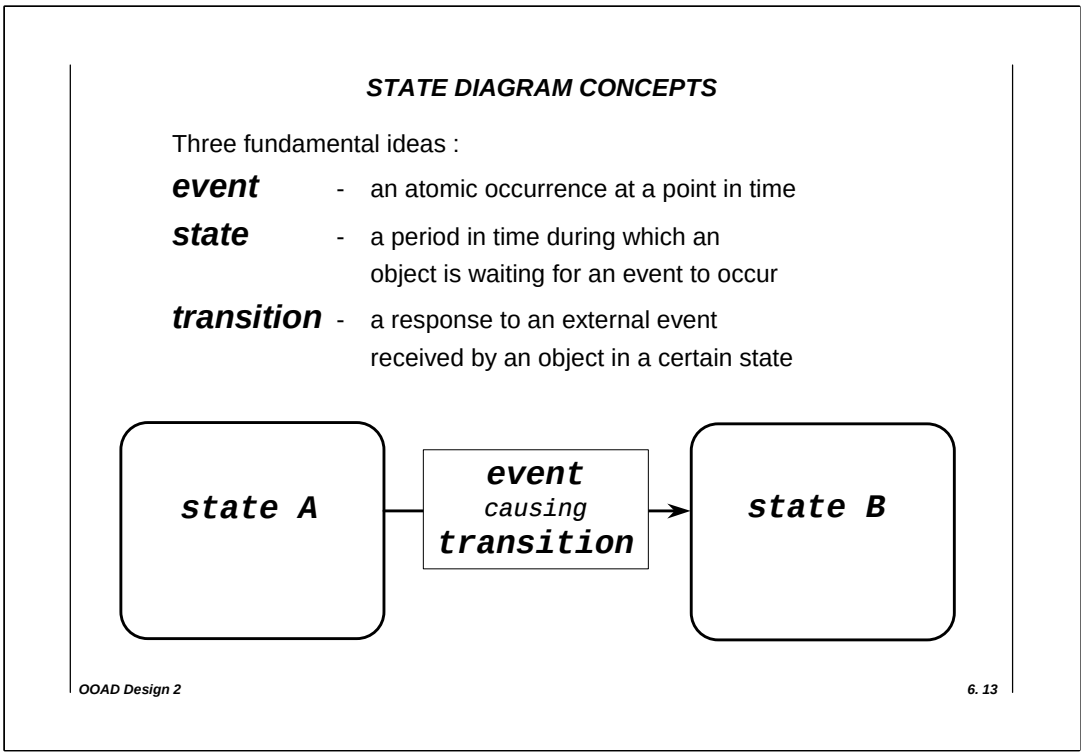
OOAD Design 2 *6. 12*

The representation of state charts in state diagrams takes the form of a collection of nodes (the states) and directed edges (the transitions).

A ‘scenario’, being an instance of a use case and an instance of the execution of a system, illustrates, but ultimately cannot define, behaviour. It is a ‘slice’ of system behaviour across state diagrams from multiple classes. The state diagram provides the means for describing the temporal evolutions of an object of a given class in response to interactions with other objects. Hence, the state diagram subsumes the different sequence diagrams that model scenarios from an object-level perspective.

Each diagram is associated with one class, or with some higher level state. Only a minority of classes undergo significant state changes necessitating diagrams.

The UML documention acknowledges the contribution of Harel. His important extensions to state transition diagrams provide a useful basis for examining the main elements of state diagram which can be modelled using the UML.



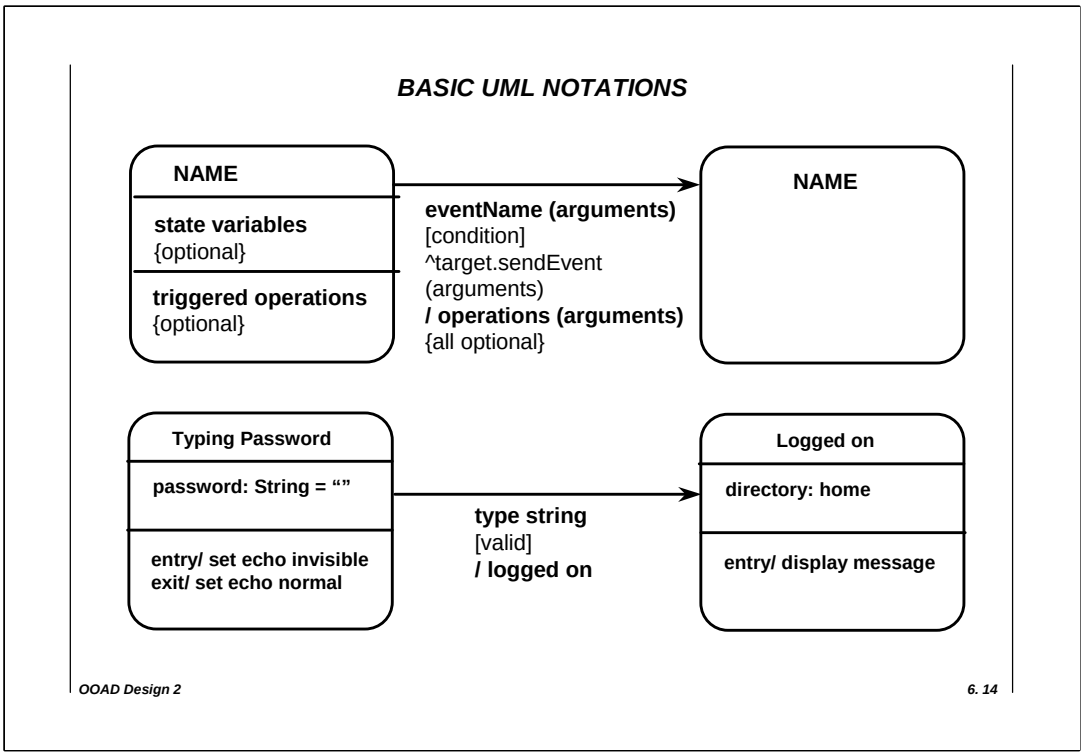
The basic concepts of state charts are applicable at all levels of abstraction, althouth state diagrams in UML (and the related notation in OOSE) are principally intended to describe the behaviour of objects at a type-level of abstraction, i.e. in classes (or design blocks in OOSE).

In defining an event, the emphasis is on its atomicity; it is a non-interruptible, one-way transmission of information from one object to another, proceeding independently (asynchronous).

The current state of an object is determined by the event that triggered its last transition and lasts until the next significant event.

A transition may both a) cause a change of state and b) invoke object operations. External transitions do a) and possibly b); internal transitions do b) but not a).

UML provides a multi-featured graphical representation for these concepts as detailed on the next slide...



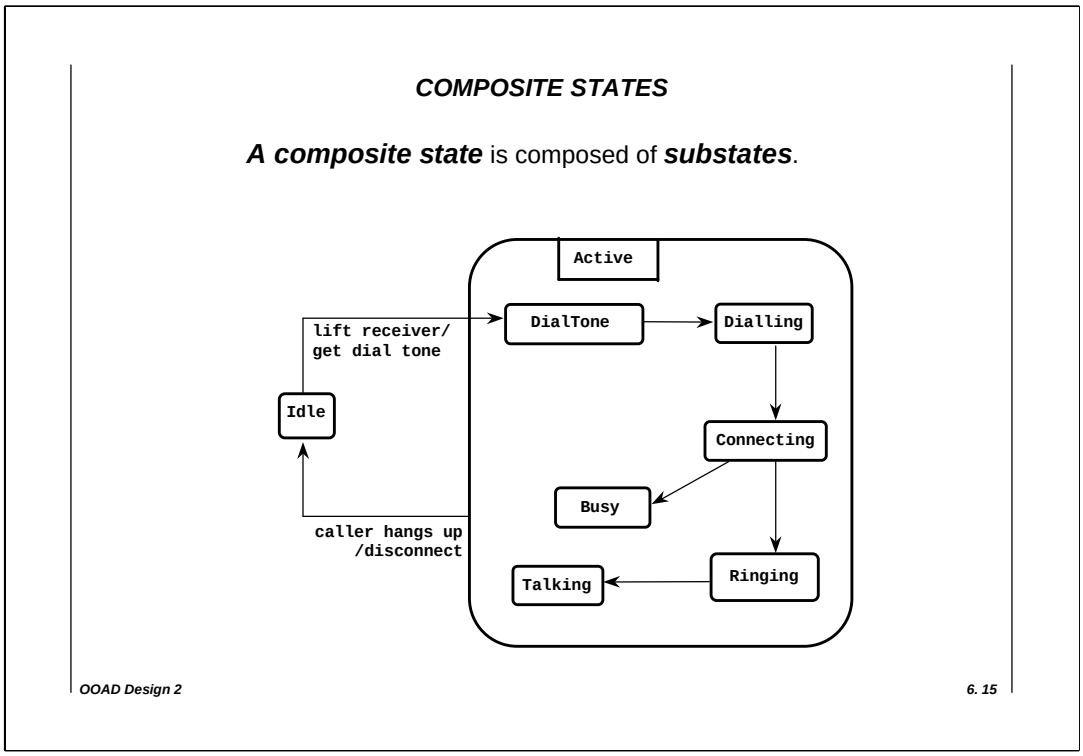
Each state appears as a rounded box with the name of the state and optional state variables and triggered operations.

A state variable is valid while the object is in the state and can be accessed and modified by operations within the state.

Operations, called ‘actions’, must be non-interruptible. They can be implemented as private methods (a method is the implementation of an operation in an object-oriented programming language) on the controlling object. Operations can be preceded by the pseudo-event names, ‘entry’ and ‘exit’, effectively making the state into a self-contained module.

The transition notation remains the same (directed arc), but the label has a more complicated syntax, of which the event name, with any associated parameters, is the principal component. Operations, possibly of other objects, can also be triggered by transitions and are included as the final component of the label. Other components are discussed below.

States can also be composed of other states. For these composite states the composition is drawn within the node representing the state. An example of these composite states is shown on the next slide...



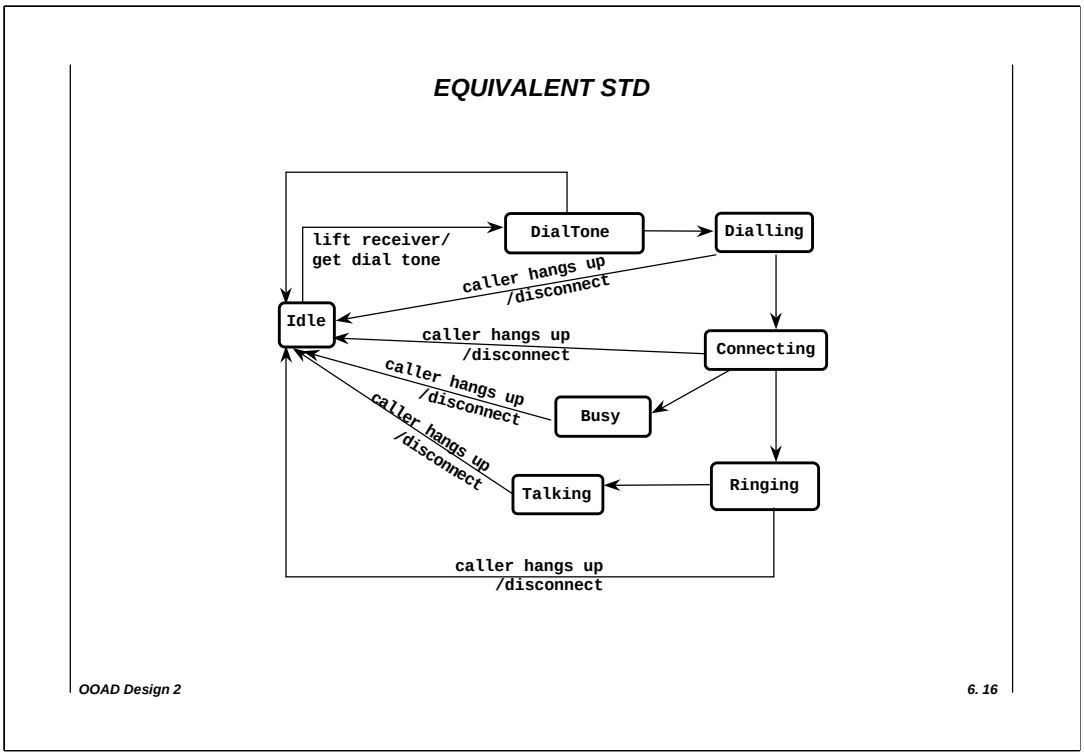
This slide displays the first example of the application of abstraction in state diagrams. At a high level of abstraction, there are only two states in the telephone, idle and active and two transitions between them. If a user lifts the receiver the telephone transits from idle to active and if the user replaces the receiver the system transits from active to idle.

At a more concrete level of abstraction, however, an active telephone can be active in different ways. These are displayed in the refinement of state active. The receiver can play the dial tone and then the user can start dialing. After that, the telephone either plays the tone for busy or it tells the user that the phone of the desired partner is ringing and if the partner responds the connection will be established and the parties can talk to each other.

The meaning of the composition of a state in this way is that the state 'active' is in exactly one of its substates.

A substate inherits the properties of its composite state, variables and transitions. More precisely, outgoing transitions are inherited. This means that if the caller replaces the receiver the telephone will become 'idle', irregardless in which active state the telephone is.

Note that state composition is the first way how Harel managed to reduce complexity. To clarify this and to explain the semantics of composition, the next slide displays a state transition diagram with the same semantics.

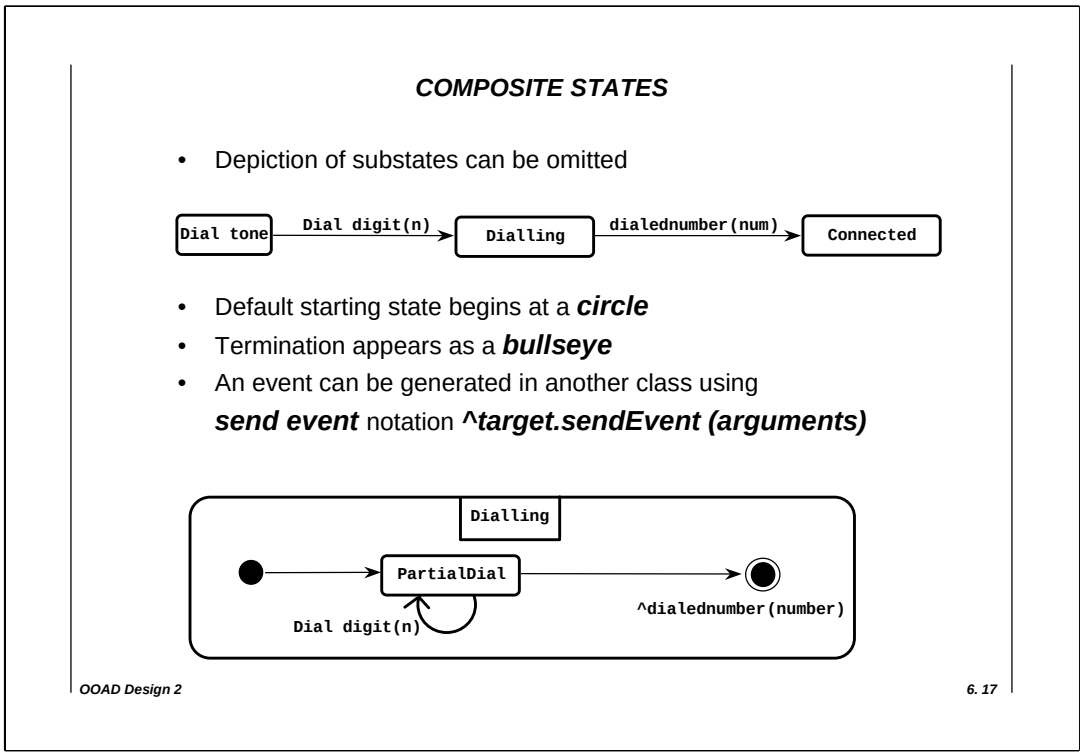


For reasons of simplicity, we have omitted the definitions of start and ending states in this diagram.

Note that a number of additional transitions are necessary in the state transition diagram. These transition lead from each active state to the idle state. They were subsumed in the state chart under a single transition leading from the composite state 'active' to the state 'idle'.

Hence composite states manage to reduce the number of transitions that are needed to model the behaviour of a class.

The next slide displays that we are able to hide the complexity of a composite state completely...



Besides the notation we have used so far for composite states, there is another notation where we omit the substates of the composite state completely. The composite states are indicated just as if they were regular states and their definition is given in a separate diagram.

This diagram needs to identify the substate that becomes active if the composite is activated. This is done by transition from a pseudo entry state that is represented as a filled circle and represents the activation of the composite state.

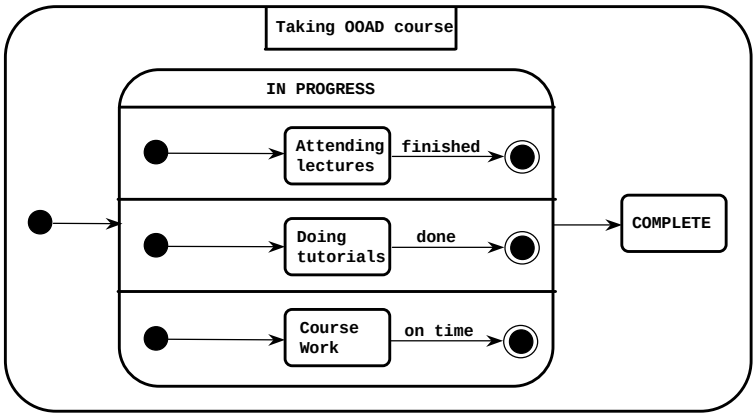
We can use a transition to another pseudo state that represents the deactivation of the composite state and is represented by a bullseye. If we omit the bullseye, the state transitions defined for the composite state are inherited by all substates.

On termination the composite state is shown sending an event to its higher-level self. In the ‘send event’ notation the ‘target’ is an expression designating a set of objects, which is not require in this example because it is fixed and well known.

The composite states we have just introduced enable us to reduce the number of transitions needed in a state chart as we can define transitions between composite states that are then inherited by all its substates. There is, however, further potential for reducing complexity if we can manage to reduce the number of states needed. As the next slide shows, concurrent states achieve that...

CONCURRENT SUBSTATES

When a state has multiple threads of control,
each concurrent substate appears as a separate region



OOAD Design 2 *6. 18*

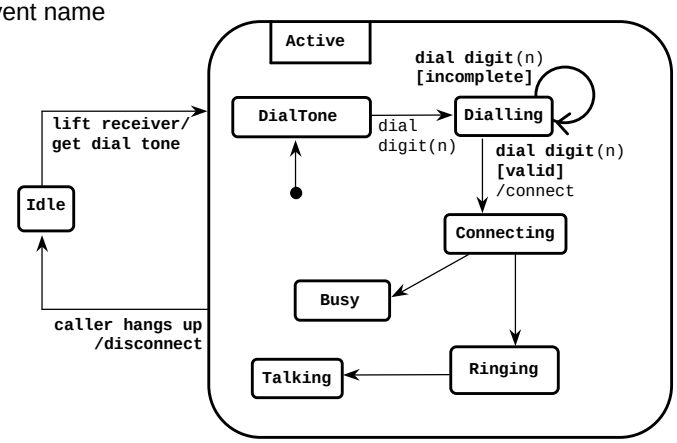
- The UML documentation suggests the following model of concurrency :

An atomic object can be thought of as a finite state machine with a queue for incoming events. New events go on the queue until the object is free to deal with them. Composite concurrent objects contain several atomic objects as parts, each of which maintains its own queue and thread of control.

> A detailed view of a telephone diagram illustrates some other features of the UML notation

CONDITIONS ON TRANSITIONS

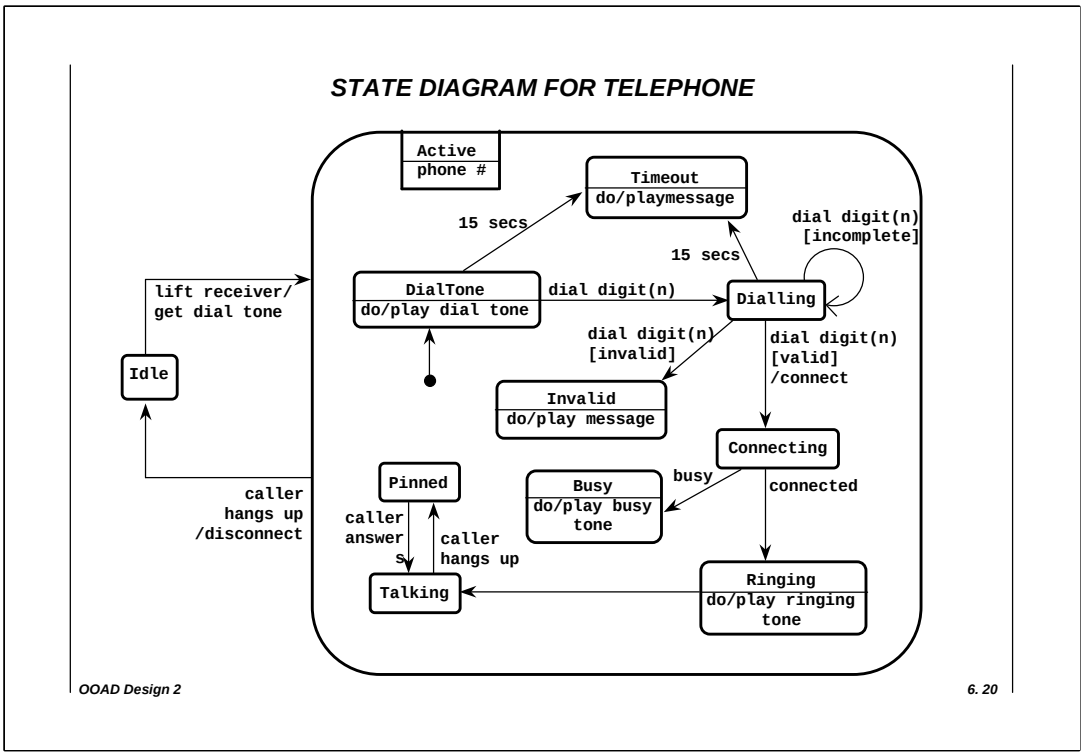
An optional guard ***[condition]*** may be attached to transitions after the event name



OOAD Design 2 *6. 19*

- A guard condition is a Boolean expression. If the event occurs and the expression is true, then the transition occurs, otherwise not. As in this example, two transitions can have the same name if different conditions are attached.

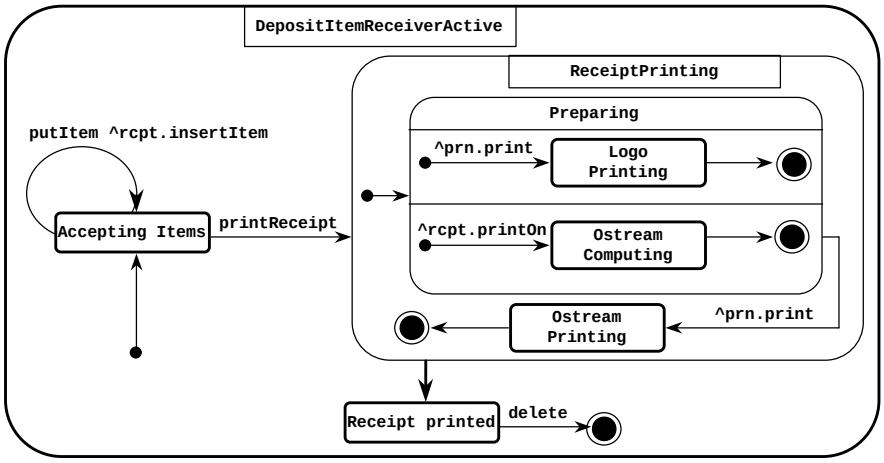
> The last of Harel's expansions of the original concept involved concurrency.



- ‘do/play dial tone’ denotes an ‘activity’.
- ‘15 secs’ is an ‘elapsed time event’.

STATE DIAGRAM FOR RECYCLING MACHINE

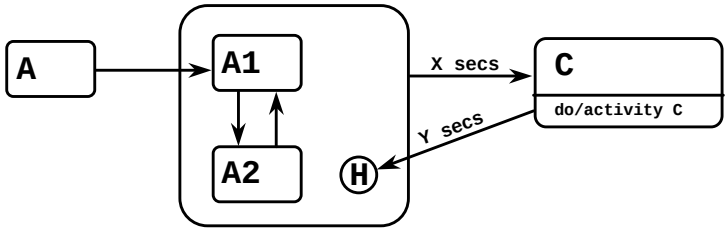
- One ***state diagram*** for ***each class*** contained in class diagram
- Example for Deposit Item Receiver



OOAD Design 2 *6. 21*

ADVANCED CONCEPTS

Activity: an ongoing operation within a state
Elapsed time event: an event occuring
a given time after entry into state
History state: a state resumed upon reentry



OOAD Design 2 *6. 22*

- An activity is an ongoing operation within a state that takes time to complete. It can be interrupted by an event that causes a state transition. An event causing exit forces its termination. An activity is indicated by a pseudo-event named ‘do’.

- In the case of the elapsed time event the sender is the “environment” rather than any individual object.

- The ‘history state’ (indicated by an ‘H’ within a circle) provides the means for a state to “remember” its substate when exited and to be able to resume the same substate on reentry into the state.

Role of state transitions in OOSE

- To increase understanding of design blocks (classes)
- To model the 'state-controlled' objects, rather than the 'stimulus-controlled'
- To help in the abstraction of the actual code
- To describe stimuli received and what happens consequently

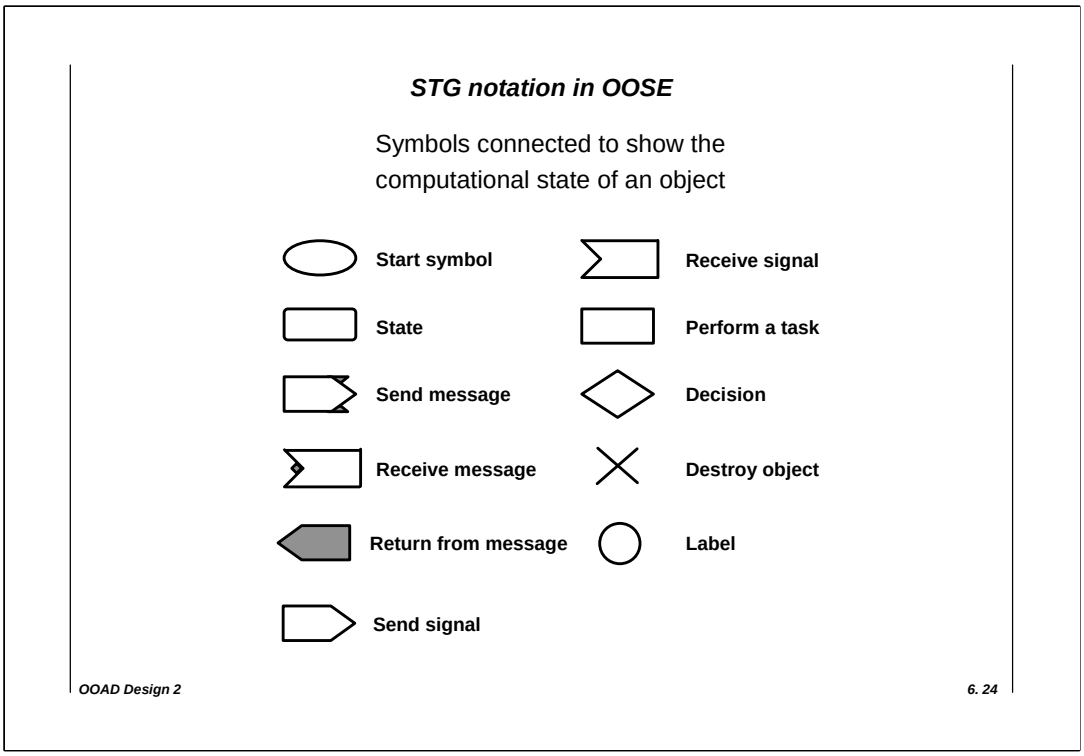
OOAD Design 2 *6. 23*

- During the block design stage of OOSE Jacobson recommends the examination of the states and state transitions of classes as a means of increasing understanding without going down to the actual code level.

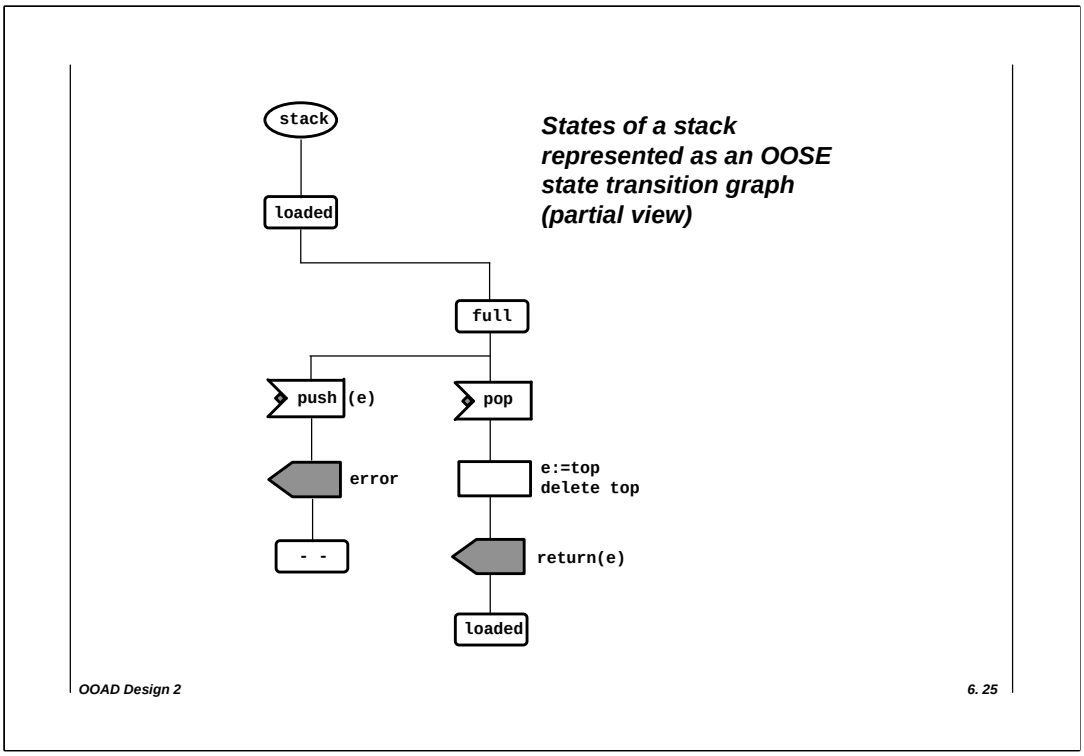
- Changes of state are important in those objects whose response to stimuli depend not only on the stimuli but also upon their state on receipt. Such objects are called 'state controlled' and are more likely to have been modelled as the 'control objects'. On the other hand the 'stimulus-controlled objects' will perform the same operation independent of state when a particular stimulus is received, e.g. the entity object 'Deposit Item'.

- Jacobson does not consider the actual technique used as critical, so long as it meets the objective of helping the abstraction of code.

- This implementation objective affects the characteristics to be described, the stimuli received and the reactions that occur on receipt, which in the notation used in OOSE employ a variety of graphic symbols far richer than those in the UML equivalent.

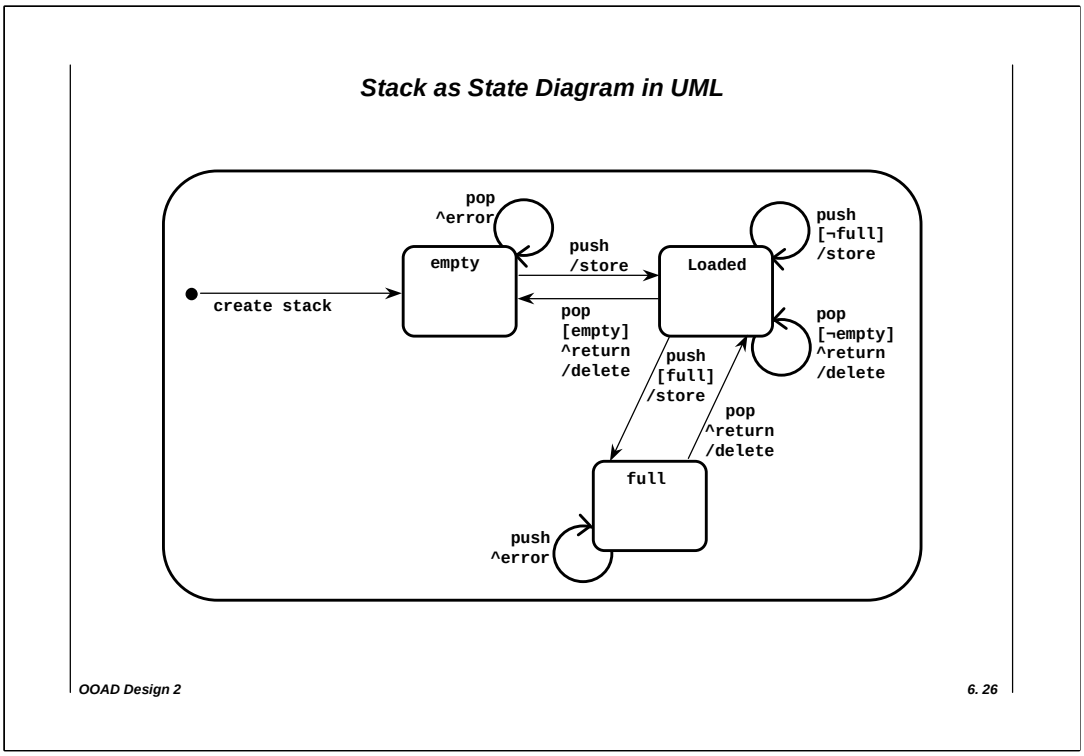


- These symbols are connected to describe the ‘computation state’ of an object, which describes how far we have come in the execution, as well as the potential future execution. Under lying these ‘computational states’ are ‘internal states’ that contain the information we use to move between the computational states. Under lying this distinction is a definition of a state as the union of all values describing the present situation, from which it is possible to fully recreate this situation. This complements rather that contradicts the position taken up in the UML where a state represents a period of time waiting for an event to occur, with associated state variables.



- For comparative purposes here is a part of state transition graph for a stack, a linear structure on which to store elements, sometimes called a LIFO (last-in-first-out) list. It is based on Jacobson's Figure 8.24

- Note that the state 'loaded' appears twice, making it less easy to identify all the possible transitions from this state, and that the state marked '- -' is the previous state, which is not specified.



- In the UML the presentation is graphically more succinct, but it is debatable whether it provides a better step towards detailed code.

- As it stands this diagram (like the previous representation of this example) might represent either a superstate with substates, or a composite state; it requires the addition of a termination transition (or transitions).

CONCLUSIONS

Modelling of class behaviour :

- is an essential part of design
- requires an understanding of states and state transitions
- can be accomplished with different notations
- can be the final step before detailed coding

OOAD Design 2 *6. 27*

- At the design stage in all o-o methods it is essential to understand the behaviour of class objects, particularly if they are ‘state controlled’. This understanding, and the design that it facilitates, depends on definition of all the potential states of an object belonging to a particular class and of the transitions that may take between these states. Representations of states can vary because, ultimately, there is not a fixed definition of what constitutes a state. The notations used bring the design closer to the detailed code by fixing all the events to which a class must respond.

SUMMARY

- Representations of complex behaviour as diagrams, matrices and tables
- FSMs as formal definitions of states and state transitions
- Harel-style state diagrams in UML
- Specific purpose in OOSE and alternative notation

OOAD Design 2 *6. 28*

- Additional material for this lecture has come from

Davis, A.M. Software requirements analysis and specification, London: Prentice Hall International Inc. 1990, which contains a useful review of many techniques.

- This lecture concludes examination of the specifics of design. At the beginning of the next there will be a review of the process so far, prior to considering the link to code.