Constructing the User Interface

- so far: good practices and guidelines to take into account when designing the interface
- now: the user interface as a piece of software
- we need a sound tool for modelling, as any other SW
- the objectives are the same:
  - SW that can be written quickly and easily
  - SW easy to test
  - SW easy to enhance, without side-effects
- complication: the interface is an "event driven" kind of SW

Event Driven Software

- systems usually consist of a main module and a number of sub-systems, each of them formed by other sub-sytems, etc.
- in many systems, after the program has been initiated, the main module controls the whole execution, passing control and parameters to its subparts
- the flow of control may vary from execution to execution, but the overall path is defined once and for all by the programmer
- in contrast, interfaces are a typical example of **event driven** software
- the various objects in the interface respond to external events, activated by:
  - the user (e.g. a mouse click)
  - the operating system (e.g. a printing error)
  - the application itself (e.g. the output of a sequence of code)
- it is this sequence of external events supplied to the application that determines the flow of control

Event Handlers

- to deal with the events occurring during execution, event handlers are needed
- these are procedures which "mediate" between individual user interface objects and specific events

  Note: we will assume that the system is designed using a OO paradigm.
- for example, if a user click on a object on the screen (e.g. a button) then the event handler `mouse_clicked` associated to the button will be involved
- each object in the interface can recognize several events, but not all of them will be "meaningful" for it (and will contain a code to execute)
- e.g. if a user presses, say, "T" when the cursor points to an empty region of the screen, nothing happens
- nevertheless, the interface should consider that this case may occur, and provide a `Key_T_pressed` event handler, which will have no code
- so, the user can control the sequence of events, but the designer of the application will determine those that are meaningful

Event-Action Paradigm for Interface Design

- Example: in a word processor, the event "Close document" affects:
  - the number functions available: all the formatting functions will be disabled, and their objects will disappear from the screen, or will change their status from abled to disabled

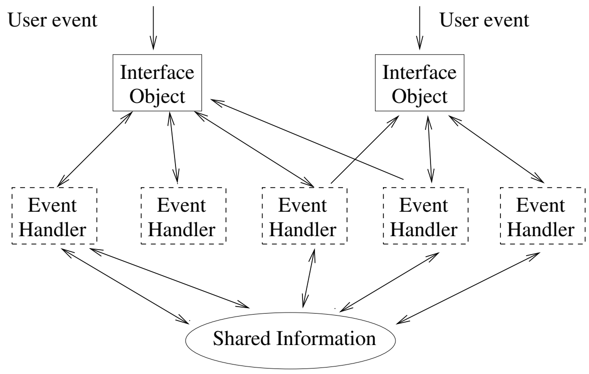
    the event handler for "close document" may determine other objects' status
  - the actions to execute in response: if there are any changes in the document and user didn't save it, a message will be prompted

    the event handler for "close document" should remember what happened before the event itself, and behave accordingly
- therefore, the event alone is not enough to describe the behaviour of the interface:
  - interface objects are not independent from each other
  - the history of events has to be taken into account

Event-Action Paradigm → Bottom-Up Design

- if you only think of the single events (mouse clicks etc.) you'll end up building the interface with a "bottom-up" approach
- that is, you start describing the basic actions (e.g. "close document") and then, add up new actions and event handlers when needed
- this leads to several problems:
  - no abstract view of the software
  - massive use of global variables → the event handlers cannot be seen in isolation
  - no OO software, because there is information shared between objects
  - the contexts are not explicit
  - small, peripheral errors are difficult to spot (e.g. it is easy to forget to add code to disable not allowed options)
  - SW difficult to check and modify

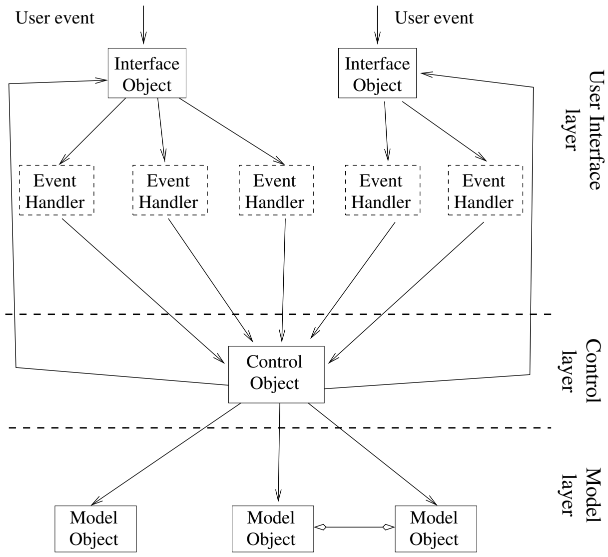
So, we'd rather not have:



Because:

- we want to avoid the user of global variables
- we don't want the interface object to interact directly with the application
- we don't want the application to be concerned with the ways its output is presented to the user, or its input is requested to the user

User Interface-Control-Model architecture



The control objects maintain the **state** of the user interface as a whole.

Solution: an Event-State-Action Paradigm

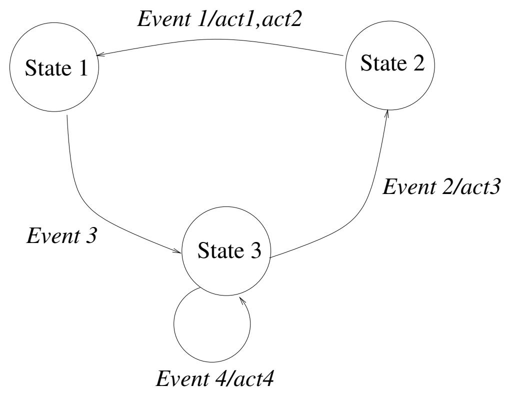
- in the event-action paradigm it is assumed that each event the user supplies determines which actions are executed
- this description is too semplicistic: actions are determined also by the contex in which the event occurred
- an event-state-action better represents this situation
- when the user supplies an event, the interface move from state to state
- each state also defines the set of possible events that the user can supply

How to Represent this Paradigm?

- we've now decided the paradigm to use
- we need a suitable notation for designing the control objects
- the interface objects must be coordinated to work together as a whole
- the notation should show explicitly the states in which the interface can be
- the notation should also show the dynamic nature of the interface, which moves from one state to another
- problem: the number of states tends to be huge

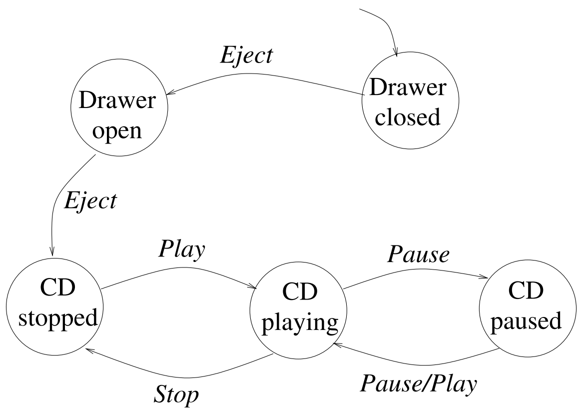
State Transition Diagrams

- STD is a notation for representing finite state machines
- a finite state machine is a model of a system which can only be in one of a given number of states at any moment of time
- such a machine accepts events in input that cause an output (action) and a change in state
- a representation of a finite state machine in a STD is:



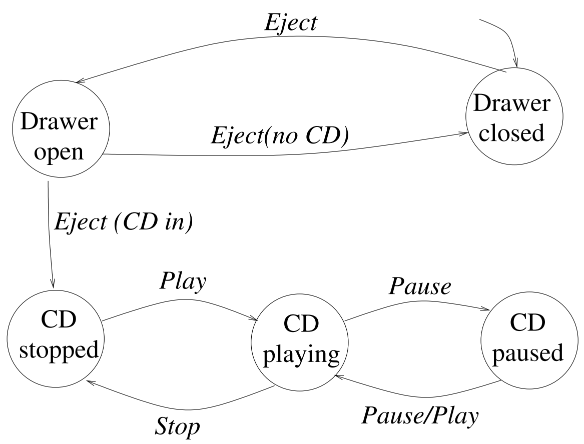
An example: CD player

- user interfaces are an example of finite state machines
- let us then try and use the STD notation to represent thir behaviour
- example: describe the behaviour of the interface of a CD player
- (remember: we are NOT interested in how the real CD player works!)



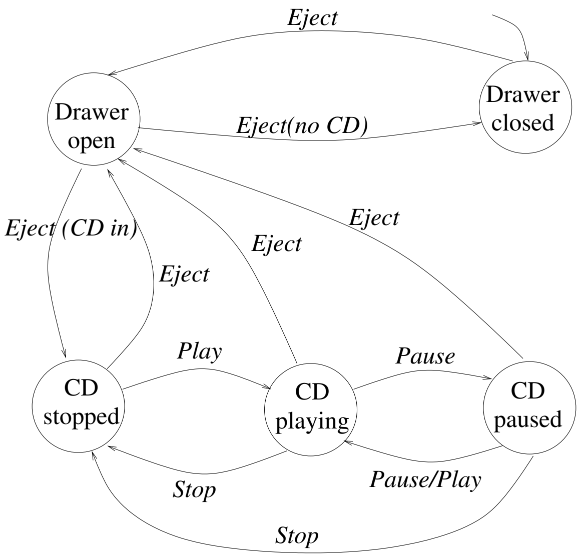
CD player - II

- the description so far is simple, and does not account for all the cases
- what if, for example, the user does not put a CD in the player?
- the STDs should have "conditions" on the arrows...



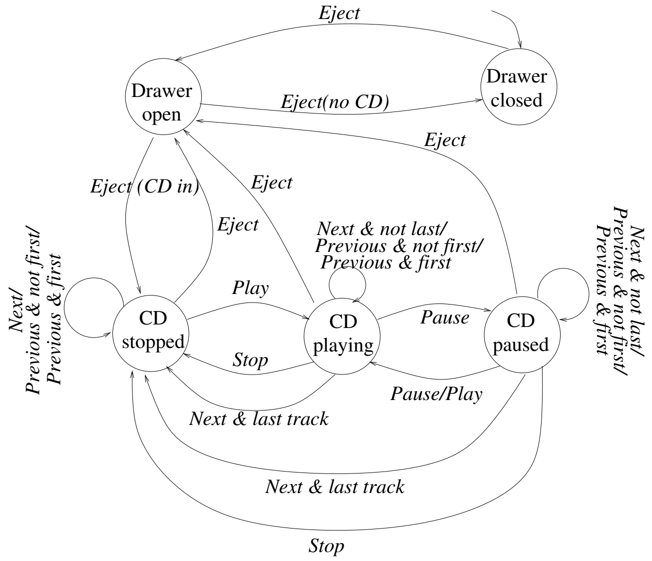
CD player - III

- the user may well vary the “normal” course of action
- for example, pressing “Eject” when the CD is playing:



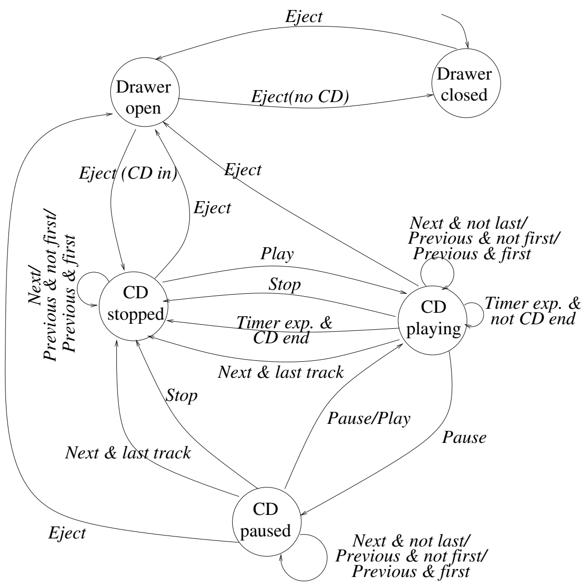
CD player - IV

- so far we have described the behaviour of the user
- what about the behaviour of the CD player itself?



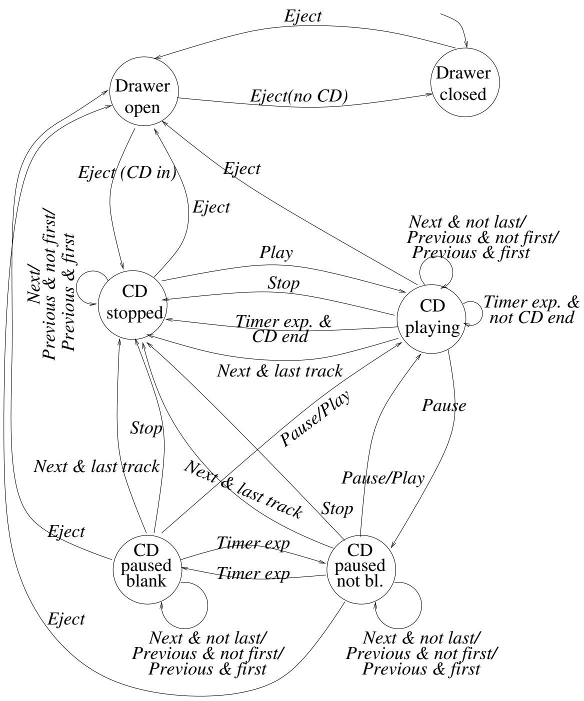
CD player - V

- there is still something missing: the CD player will have a display, showing, for example, the current track time



CD player - VI

Suppose we want to keep record of the fact that when the CD is pausing the numbers on the display are shown for a sec., then hidden, then shown againg after a sec. etc.:



And so on...

- we can keep on adding lots of events and states
- for instance, the events that are not allowed in particular states should cause the interface object to vary (e.g. disabling the buttons)
- each event causes a number of arrows to be added
- and in some cases, states too need to be added
- moreover, in some occasions the same event or state may appear in different places of the chart
- this is likely to cause errors or inconsistencies

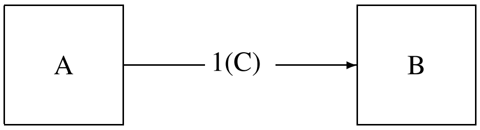
Problems with STD

- we needed to extend the basic notation by adding conditions on the event arrows
- even a modest rise in the complexity of the system causes a very rapid growth in the number of states
- the probability of duplicating events and states is very high
- the STD is too large, and difficult to read
- STD are not scalable

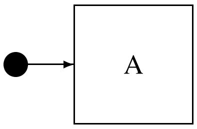
solution? State Charts notation

The StateChart Notation

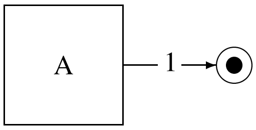
- A simple transition: *If event 1 occurs in state A and condition C is true at that time, then make the transition to state B*:



- Representing a default start state:

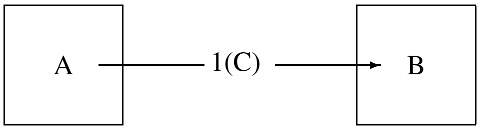


- End state: *If event 1 occurs in state A then go to the end state (the application will close down)*:



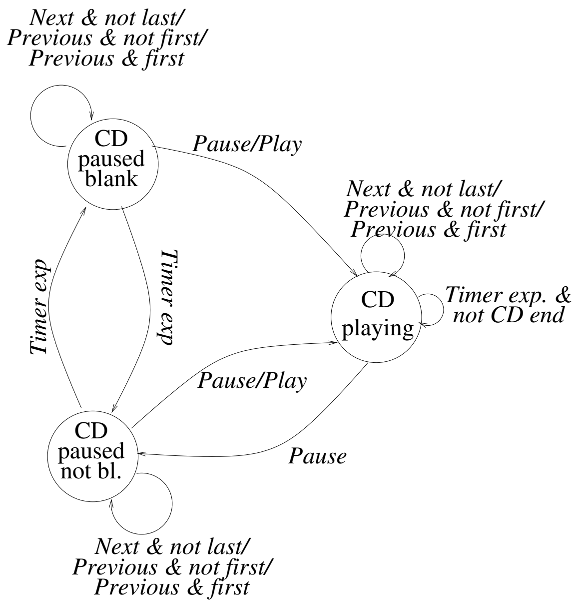
Refinement

- a big problem in the STD notation was the presence of too many states at the same level
- in the StateCharts notation states are organized in *levels*
- higher levels represent abstractions of a part of the system
- and then each level is refined on a separate chart
- to show that a state has lower level of details, the notation uses arrows starting and ending “inside” a state:

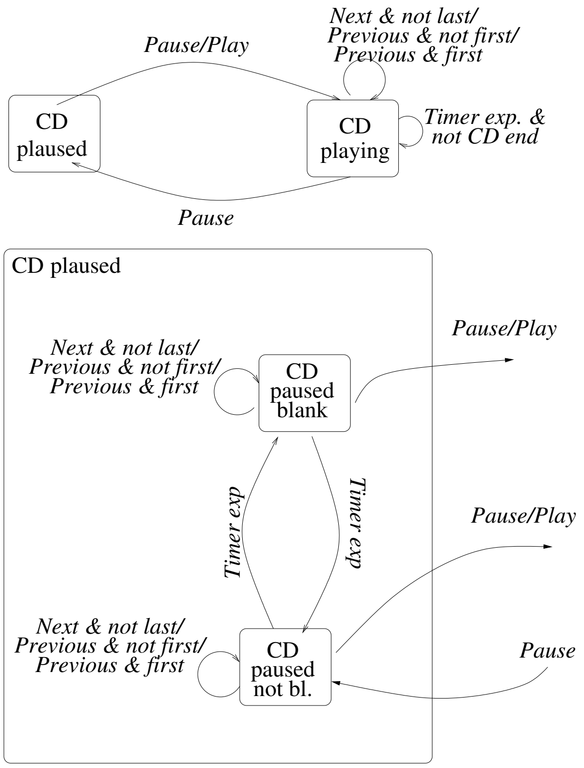


Refinement in the CD Player Example

Consider this bit of the STD diagram for the CD player:

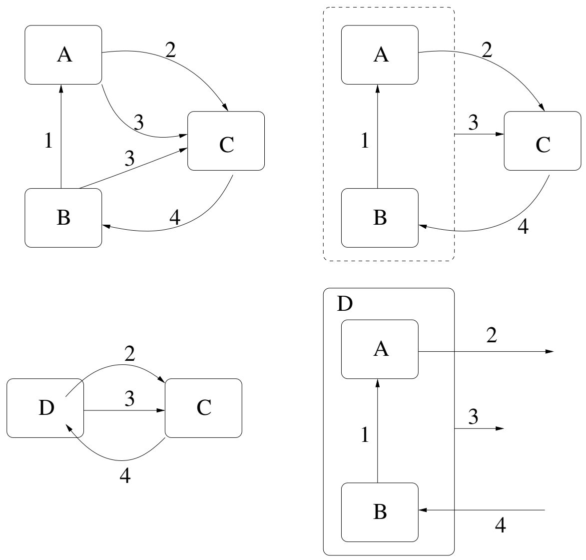


Here is how a refinement technique can be applied to produce the statecharts:



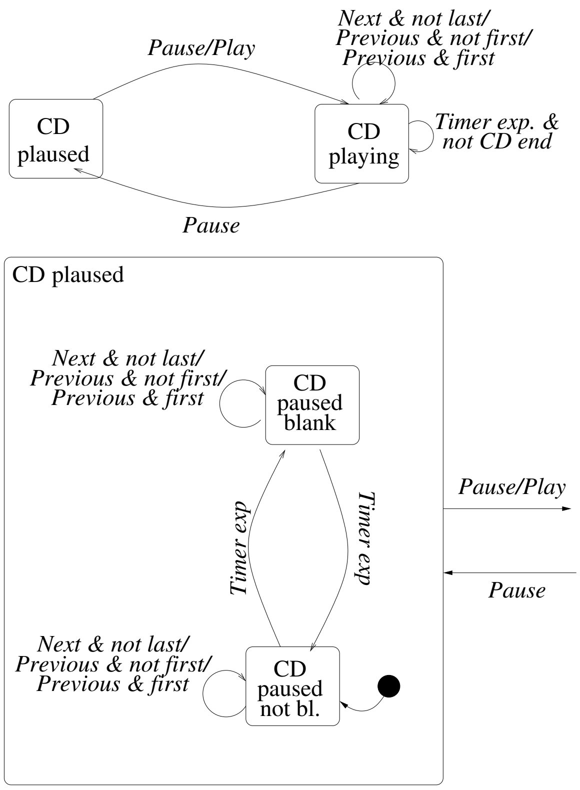
Clustering

- deep levels of states can also be used for clustering
- groups of states are clustered together to reduce the number of event arrows, for example:



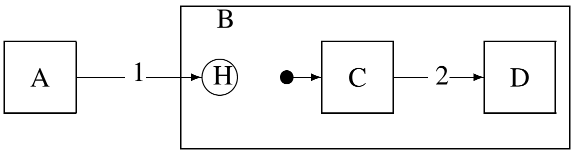
- event 3 can occur in any of the states in D (A or B)

Clustering in the CD Player Example



History

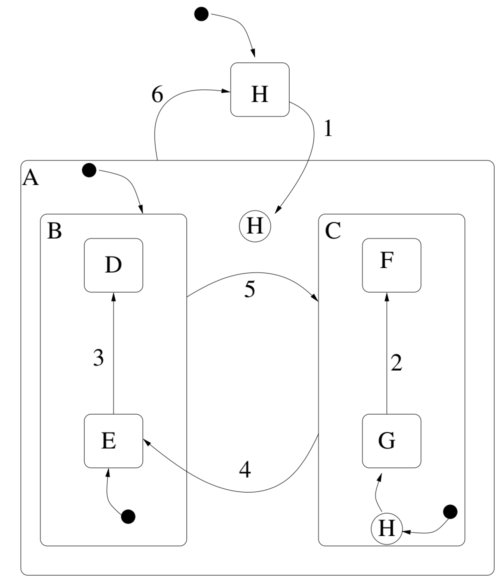
- the history mechanism provides a way of entering a group of states based on the history of that group
- it is a way to express that “the state to be entered is the most recently visited state of the group”
- for instance in following situation, once event 1 occurs, the system will enter either in state C or in state D, whichever was the most recently visited:



(if first time in B, the default state is C)

- the history mechanism is applied only at the level in which it appears
- to apply it at lower levels, it is necessary to use a new history symbol

For example:



(notice the default attached to the history symbol in C: this indicates that if state C has not been visited before, then G is entered by default)

The Complete CD Player Example