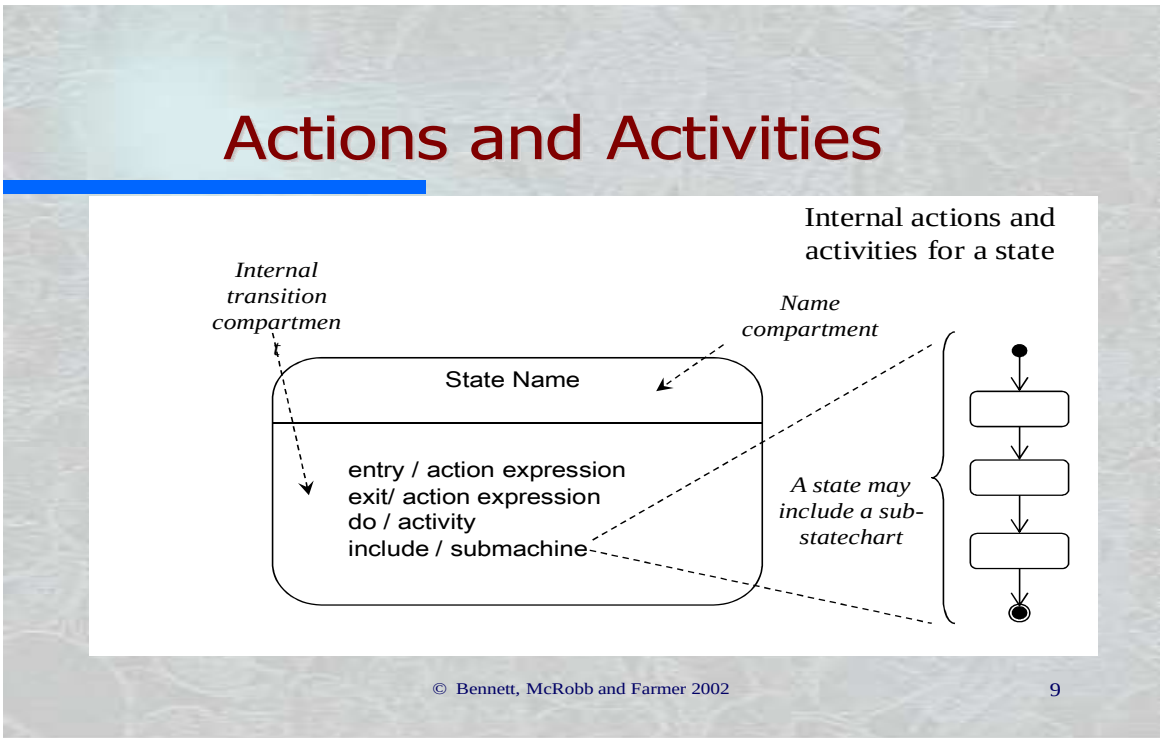
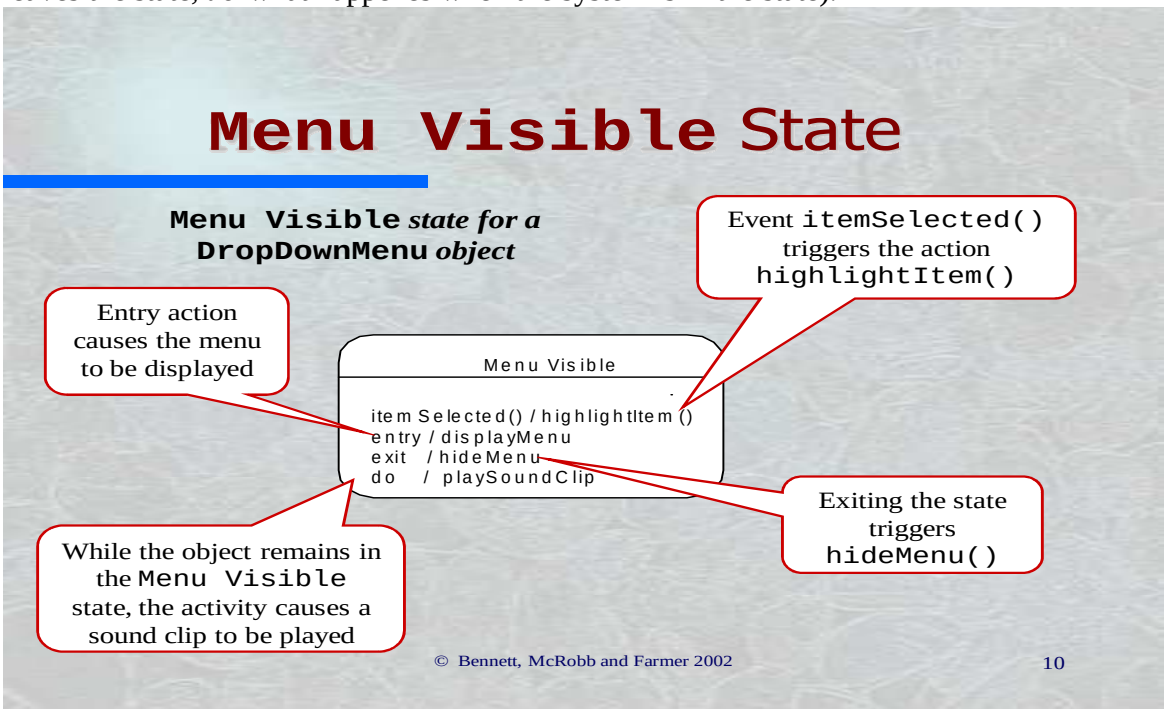
STATE DIAGRAM



(***entry***-what happenes when the system enters the state, ***exit***-what happenes when the system leaves the state, ***do***-what happenes when the system is in the state).



State Diagram – Example 1

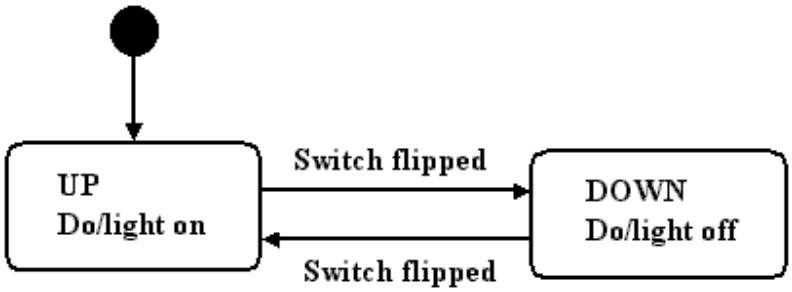
Example: ordinary two-position light switch

A light switch will have two states: up and down. (We could call them "on" and "off" if we liked.) In a UML state diagram, each state is represented by a rounded rectangle.

A light switch only has one possible event: the switch gets flipped. We could call this two different events (gets flipped up, gets flipped down) but the net effect is the same. In a UML state diagram, each possible event that can happen to cause an object or system to change from one state to another is represented by an arrow from the original state to the resulting state, labelled with the name of the event.

For our example diagram, we can pick an arbitrary start state, represented by a solid black circle with an arrow to the starting state.

Here is a sample state diagram that would work for our two-position switch:

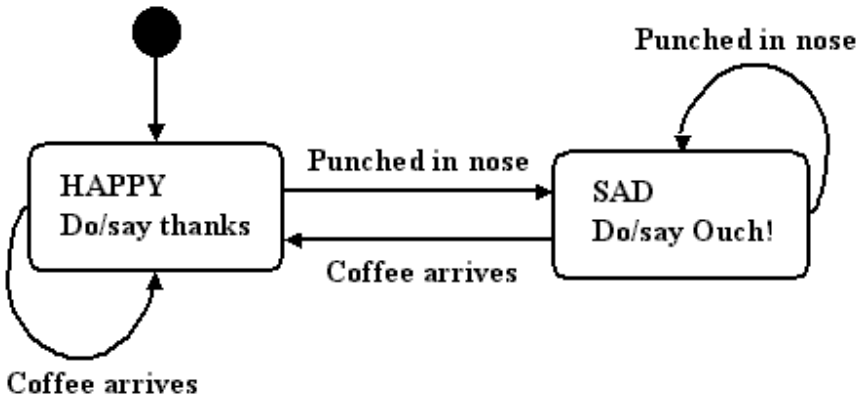


One could argue that our light switch really has hundreds of thousands or millions of states, since every time we flip the switch it wears out just a little bit, and eventually will fail if we flip it enough. However, in the world of state diagrams, we fortunately do not include this type of "event." We only include the states of the system when it is working in its normal course of operation. Although the actual number of possible events that could happen to any system (such as our light switch) approaches infinity (e.g., such as the switch being smashed by an asteroid hitting the earth), we do not include this type of event in the state diagram. (This is a good thing or all state diagrams would be so large as to be unreadable!) The only possible events that are shown on the state diagram are the ones the system is specifically built to handle and still keep working.

Example: simplistic Teaching Assistant (TA) :)

The states in a state diagram represent the states of being of the system. For example's sake, suppose that your TA only has two states of being: happy and sad. Further suppose that your TA is basically cheerful and starts out happy. :) Also suppose that your TA, not having much of a life, can only respond to two events: getting coffee (which makes the TA happy) and getting punched in the nose (which, needless to say, makes her sad).

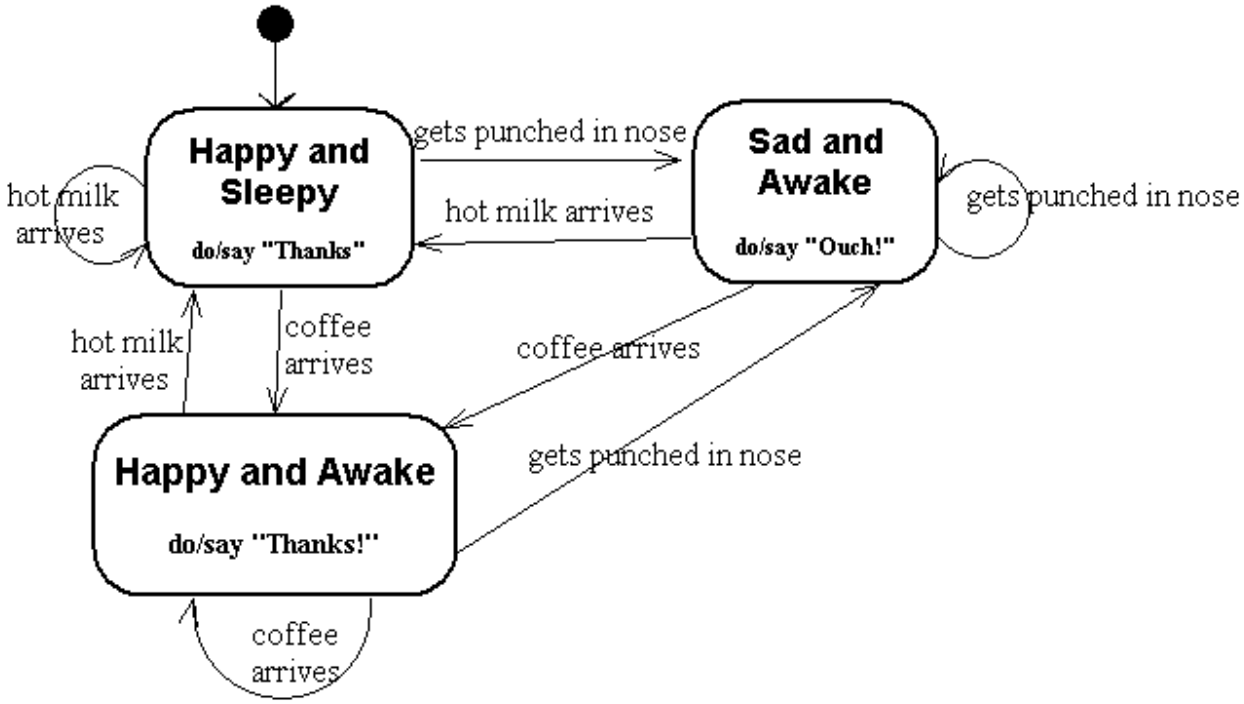
Here is a state diagram that would represent our simplistic TA:



Note that although the TA need never get punched in the nose in her entire life--and we hope she doesn't!--we still need to include this event in our state diagram, just in case it happens. A state diagram must be able to represent any possible input sequence, not just input sequences that have already happened, or are likely to happen. As long as the event (such as getting punched in the nose) is an event the TA is built to recognize and respond to (by becoming sad and saying "Ouch!"), then we have to include it in her state diagram.

Also notice that some of the arrows in the TA's state diagram loop back to the state from which they came. That is, if the TA is already happy, and coffee is brought, she stays happy (and says "Thanks!"). If the TA is already sad, and gets punched in the nose again :(, she stays sad, and says, "Ouch."

Here is a slightly more complex TA, with three states. This TA also responds to the arrival of hot milk, as well as to the arrival of coffee and to being punched in the nose. This TA starts out happy and sleepy, and only wakes up when coffee arrives, or by being punched in the nose. She also gets sleepy and happy again each time hot milk shows up.

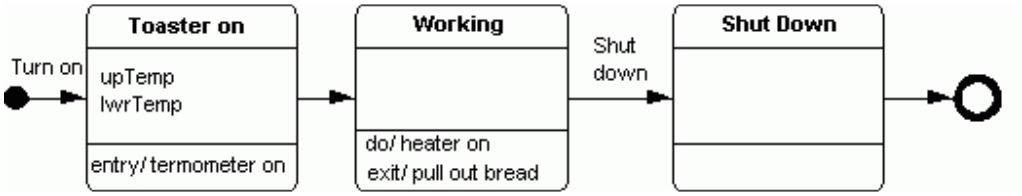

Happy and Sleepy
do/say "Thanks"
gets punched in nose
Sad and Awake
do/say "Ouch!"
hot milk arrives
hot milk arrives
gets punched in nose
hot milk arrives
coffee arrives
coffee arrives
Happy and Awake
do/say "Thanks!"
gets punched in nose
coffee arrives

State Diagram – Example 2

Suppose you're designing a toaster. You would build a plenty of UML diagrams, but here only state diagrams will be of our interest.

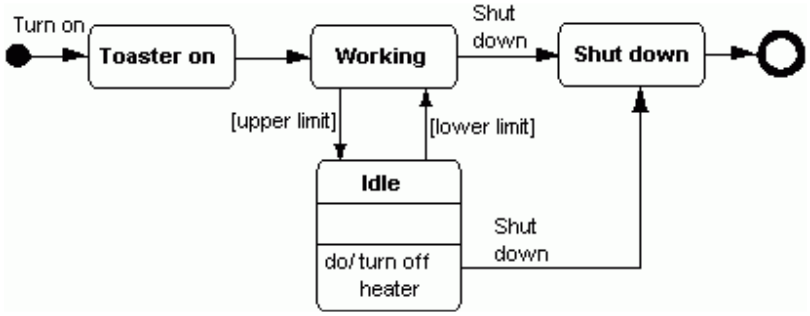
- What are the steps of making a toast?

First of all we must turn on the toaster, put in the bread and wait for several minutes to bake it. The initial state diagram is shown below:



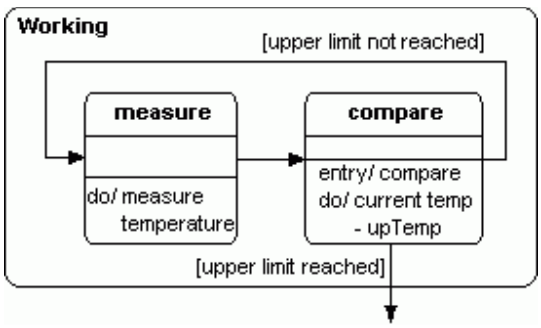
The initial state diagram for making a toast

But this is not the final state diagram. To prevent burning out the bread, heater of the toaster must produce heat in temperature interval (upper and lower temperature limits). For this purpose thermometer measures the temperature of heater, and when the upper limit of temperature is reached then heater must go into idle state. This state resists until heater's temperature decreases to lower limit, and then working state is again aimed. With this new state, extended state diagram will be:

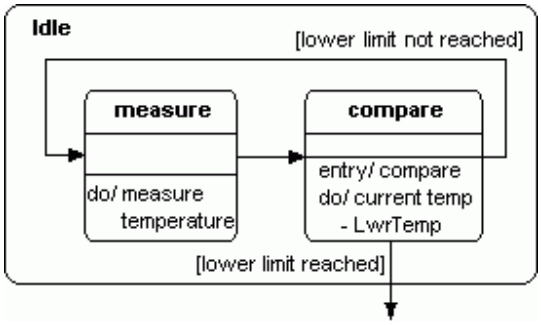


The extended state diagram for making a toast

Transition between working and idle state is not presented in details. To do this substates must be added.



*Substates in **Working** and **Idle** states*



Substates in working and idle states are very similar. both had measure and compare states, but differentiates in process of temperature comparison. Working state must compare current temperature with upper temperature limit (if it is reached, working state goes into idle state), and idle state compares current temperature with lower temperature limit (idle state is replaced with working state when temperature falls under lower limit).

It's necessary to have state diagrams because they help analysts, designers and developers understand the behavior of the objects in a system. Developers, in particular, have to know how objects are supposed to behave because they have to implement these behaviors in software. It's not enough to implement an object: Developers have to make that object do something.

SCENARIO

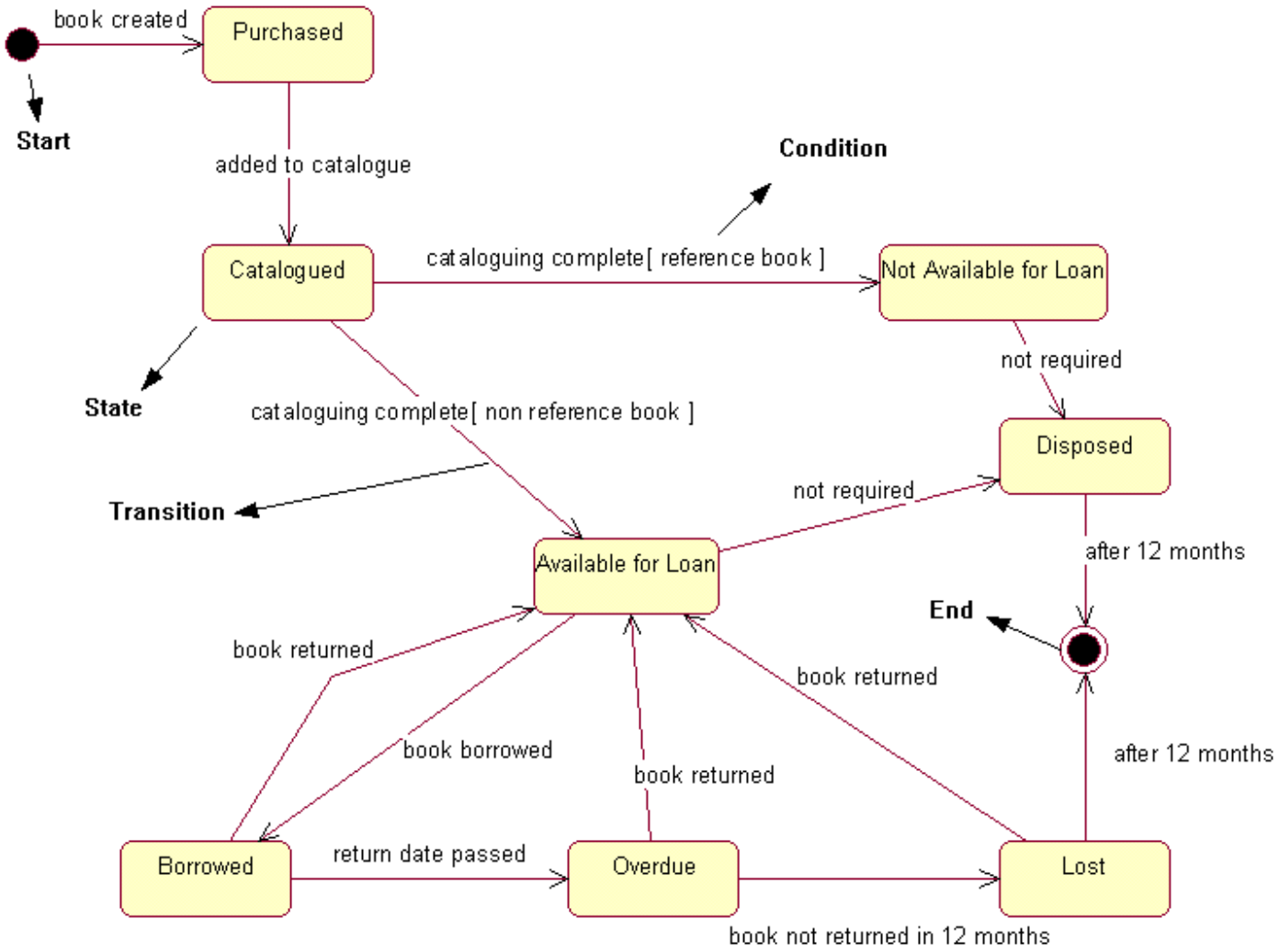
State diagrams describe the **lifecycle of a given object of a class**. They show all the possible **states** that the object can get into and the **transitions** which show how the object's state changes as a result of events that reach the object. They are useful to describe the behaviour **of an object** across several use cases. We use state diagrams only for classes that exhibit interesting behaviour.

State diagrams are not good at describing behaviour where several objects are interacting. If too many concurrent behaviours exist in an object's state diagram, it is an indication that there are probably more than one object interacting.

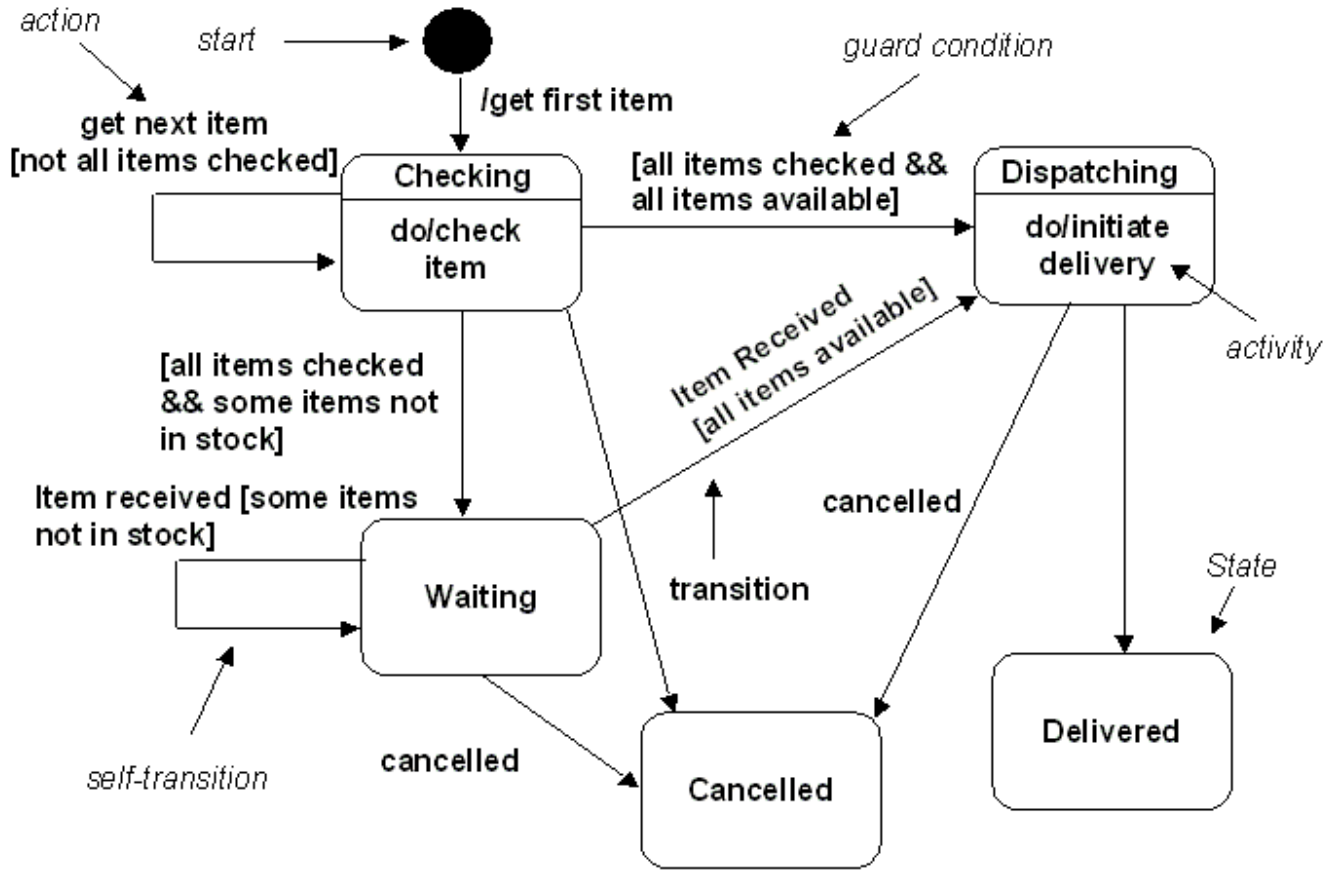
The following is a basic state diagram of an object of class book.

Scenario (CASE STUDY)

Librarians categorise the library books into loanable and non-loanable books. The non-loanable books are the reference books. However, the loanable books are the non-reference books. After cataloguing the books, the books are available for loan. Students who borrow the library books should return them back before the due date. Books that are 12 months over the due date would be considered as a lost state. However, if those books are found in the future, they must be returned back to the library. When the books are found not required in the library or have been damaged, the book would be disposed.


book created
Purchased
Start
added to catalogue
Condition
Catalogued
cataloguing complete[reference book]
Not Available for Loan
not required
State
cataloguing complete[non reference book]
Disposed
Transition
not required
Available for Loan
after 12 months
End
book returned
book returned
book borrowed
book returned
after 12 months
Borrowed
return date passed
Overdue
Lost
book not returned in 12 months

Another Example :



We begin a state diagram at the start point and show the initial transition to the first state. The syntax for a transition label has three parts, all of which are optional: `Event[Guard]/Action.` A state has a name and an optional activity associated with it, indicated by a label with the syntax `do/Activity`. Note, that `Actions` are associated with states and `Activities` are associated with transitions. Actions are quicker than Activities and cannot be interrupted. An Activity can be interrupted by some event. A `guard` is a a logical condition that will return only "true" or "false". A guarded transition occurs only if the guard resolves to "true. Only one transition can be taken out of a given state, hence the guards attached to the transitions coming out of a state should be mutually exclusive for any event.