[Handout for L11P1]
Modeling your way out of complexity: other useful models

A *model* is anything used in any way to represent anything else. For example, a class diagram is a model that represents a software design and is drawn using the UML modeling notation. Models are a great help in providing a simpler view of a much more complex entity because a model often captures only some aspects of the entity while abstracting away other aspects. For example, a class diagram captures the class structure of the software design but not the runtime behavior. Therefore, we often have to create multiple models of the same entity if we are to understand all relevant aspects of it. For example, in addition to the class diagram, we might create a number of sequence diagrams to capture various interesting interaction scenarios the software undergoes. Models are useful in several ways:

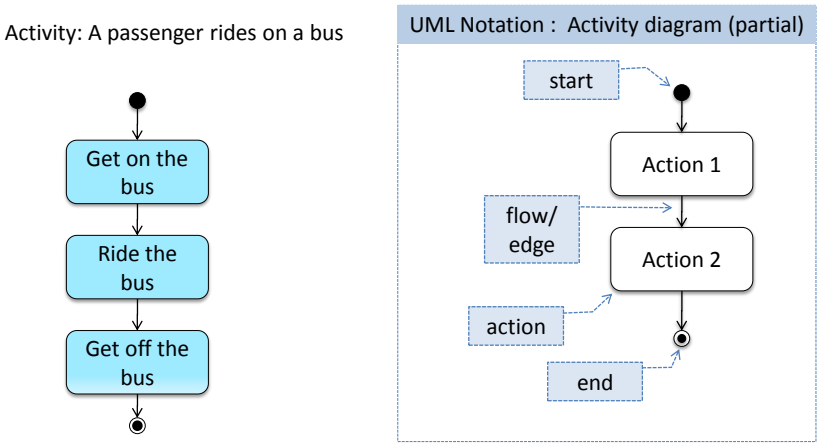
a) **To analyze** a complex entity related to software development. For example, we can build models of the problem domain (i.e. the environment in which our software is expected to solve a problem) to help us understand the problem we are trying to solve. Such models are called *domain models*. Similarly, we can build models of the software solution we plan to build to figure out how to build it.
b) **To communicate** information among stakeholders. We can use models as a visual aid in discussions and documentations. To give a few examples, at architect can use an architecture diagram to explain the high-level design of our software to developers; a business analyst can use a use case diagram to explain to the customer who can use which functionality of the system; we can reverse-engineer a class diagram from the code to explain to a new developer the design of a component.
c) **As a blueprint** for creating software. We can use models as instructions to build software. *Model-driven development (MDD)* is an approach to software development that strives to exploits models in this fashion. MDD uses models as primary engineering artifacts when developing software. That is, we first create the system in the form of models. After that, we convert models to code using code-generation techniques (usually, automated or semi-automated, but can even be manual translation from model to code). MDD requires the use of a very expressive modeling notation (graphical or otherwise), often specific to a given problem domain. It also requires special and sophisticated tools to generate code from models and maintain the link between models and the code. One advantage of MDD is that we can use the same model to create software for different platforms and different languages. MDD has a lot of promise, but it is still an emerging technology. The emergence of MDD is one more reason for this module to emphasize on models.

In this handout, we look at some of the models we can use to model the problem domain so that we understand the problem well before we start to solve it. But note that some of these models can be applied when designing the solution as well.

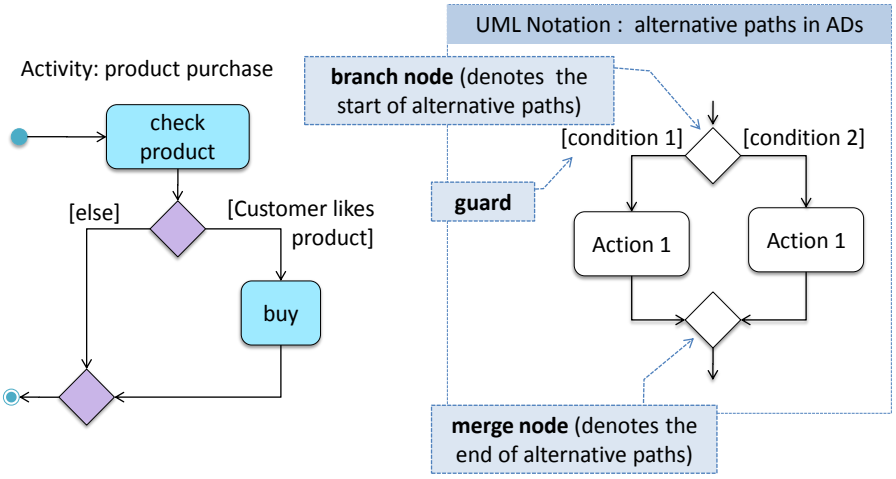
Modeling workflow

Workflows define the flow or a connected sequence of steps in which a process or a set of tasks is executed. Understanding the workflow of the problem domain is an important if the problem we are trying to solve is connected to the workflow.

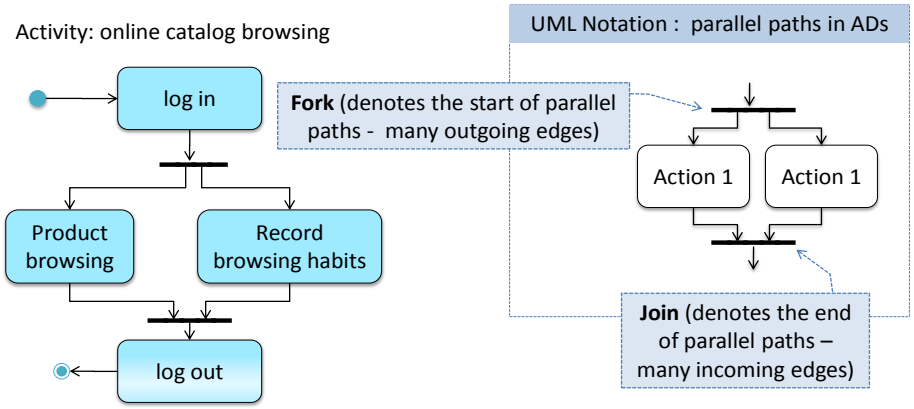
We can use UML *Activity diagram* (AD) to describe a workflow. An activity diagram (AD) consists of a sequence of actions and control flows. An *action* is a single step in an activity. It is shown as a rectangle with rounded edges. A control flow shows the flow of control from one action to the next. It is shown by drawing a line with an arrow-head to show the direction of the flow.



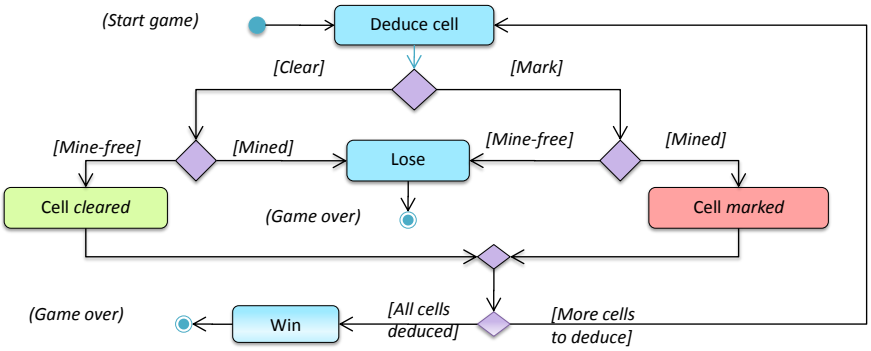
Branch nodes and *merge nodes* have the same notation: a diamond shape. They are used to show alternative (not parallel) paths through the AD. The control flows coming away from a *Branch node* will have guard conditions which will allow control to flow if the guard condition is satisfied. Therefore, a *guard* is a boolean condition that should be true for execution to take a specific path.



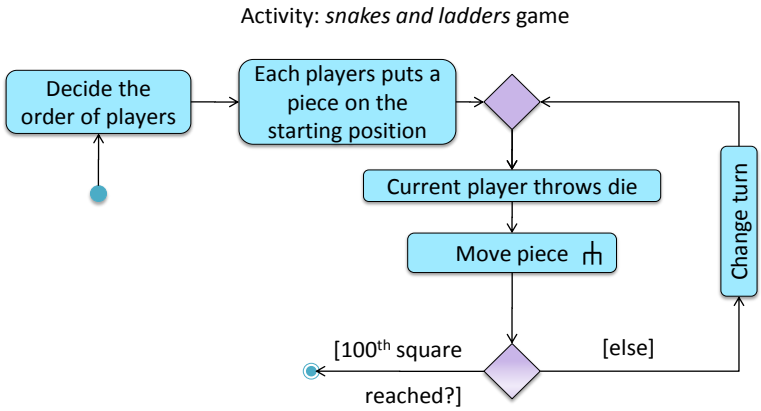
Forks and *joins* have the same notation: a bar. They indicate the start and end of concurrent flows of control. The following diagram shows an example of their use. For *join*, execution along all incoming control flows should be complete before the execution starts on the outgoing control flow.



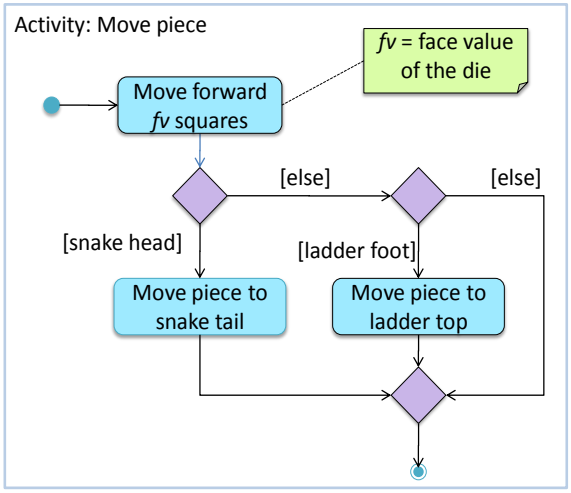
Here is the AD for the Minesweeper.



Here is the AD for a game for a 'snakes and ladders' game.



We use the *rake* symbol (in the "Move piece" action above) to show that an action is described in another subsidiary activity diagram elsewhere. That diagram is given below.



Note : Only essential elements of ADs are covered in this handout.

Modelling state dependent behaviour

Consider how a CD player responds when the "eject CD" button is pushed:

- If the CD tray is already open, it does nothing.
- If the CD tray is already in the process of opening (opened half-way), it continues to open the CD tray.
- If the CD tray is closed and the CD is being played, it stops playing and opens the CD tray.
- If the CD tray is closed and CD is not being played, it simply opens the CD tray.
- If the CD tray is already in the process of closing (closed half-way), it waits until the CD tray is fully closed and opens it immediately afterwards.

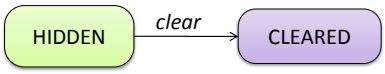
What this means is the CD player's response to pushing the "eject CD" button depends on what it was doing at the time of the event. More generally, the CD player's response to event received depends on its internal state. We call such behaviour *state-dependent behavior*.

Often, state-dependent behaviour displayed by an object in a system is simple enough that we need not pay extra attention to it; we think of such behaviour as simple 'conditional' behaviour such as 'if x>y, then x=x-y'. Occasionally we encounter objects exhibiting state-dependent behaviour that is complex enough to capture into a separate model. We can use UML *state machine diagrams* (SMD for short, sometimes also called 'state charts', 'state diagrams' or 'state machines') to model such state-dependent behaviour.

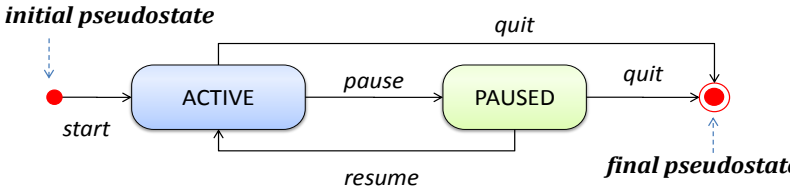
An SMD views the life-cycle of an object as consisting of a finite number of states where each state displays a unique behaviour pattern. An SMD captures information such as the states an object can be in, during its lifetime, and how the object responds to various events while in each state and how the object transits from one state to another. As we have seen in the earlier sections in this chapter, sequence diagrams capture one scenario at a time. In contrast, SMDs capture the object's behaviour over its full life cycle.

The following partial SMD for a Cell object from the Minesweeper illustrates some of the basic elements of an SMD:

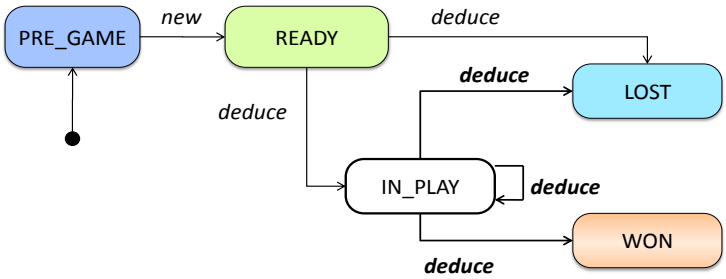
- *state*: A phase in the objects life-cycle that shows a unique behaviour pattern (shown as rounded rectangles) e.g. HIDDEN, CLEARED
- *transition*: A movement from one state to another (shown as a directed arrow).
- *trigger*: A single event that causes a potential transition e.g. clear



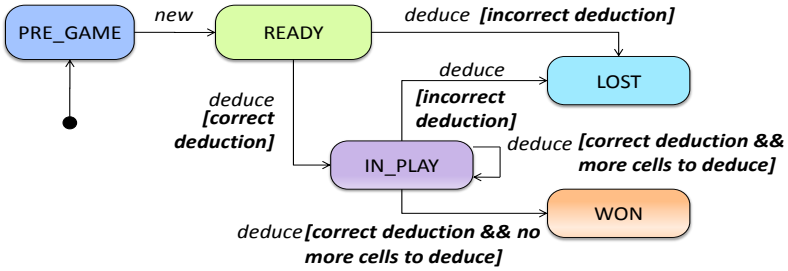
Given below is an SMD for an unidentified application. It illustrates the symbol for the *initial pseudostate* which indicates the starting state of the SMD and the *final pseudostate* which indicates the end of the lifecycle. Note that the final pseudostate can be omitted (for simplicity) if there is no specific way to end the object's lifecycle other that shutting down the system.



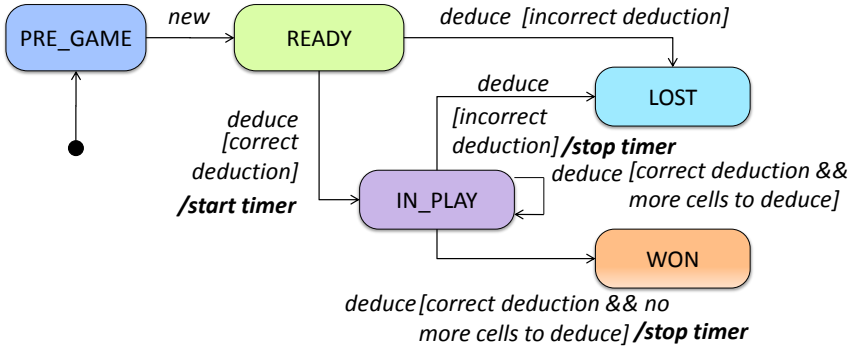
Given below is the (partial) SMD for the Minesweeper game. Here, we assume that the lifecycle of the whole game is managed by one object. Note that PRE_GAME means we have started the game but no Minefield has been created yet while READY means the Minefield has been created and Minesweeper is ready to play.



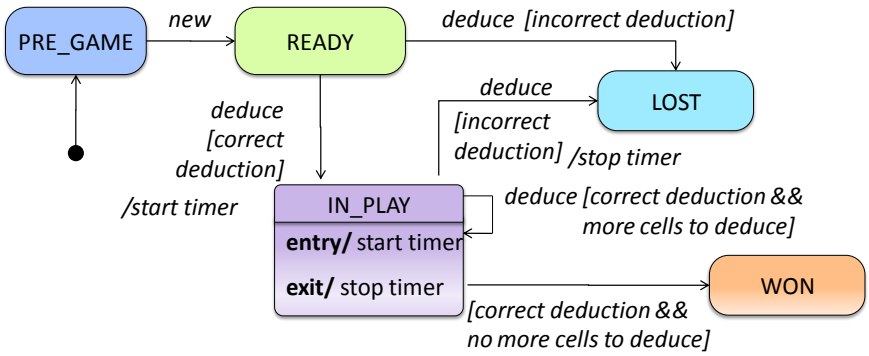
Note how the SMD is not precise on which transition to follow if the deduce event happened while in IN_PLAY state. i.e. a given event in a given state will not always produce the same transition. In simpler terms, it exhibits "random" behavior. Such SMDs are said to be *non-deterministic*. Non-deterministic SMDs are not much use to us. We can make an SMD *deterministic* (i.e. make it exhibit "predictable" behavior) by coupling non-deterministic transitions with *guards*. A guard is a boolean condition that must be true for the transition to take place. Guard conditions are shown using square brackets, as illustrated below. Guard conditions can be given in informal language or using the syntax of the programming language (e.g. [size() >= 5]) or using a specialized formal language such as the Object Constraints Language (OCL). If all guard conditions are false for a particular occurrence of an event and there is no unguarded transition specified, the event will be ignored.



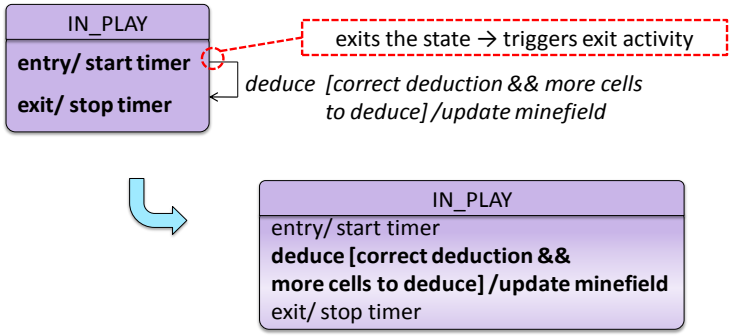
An *activity* is a behavior that takes place *during* a transition. The UML syntax is `event[guard]/activity` where all 3 components are optional. Note that in the context of SMDs, 'activity' 'action' and 'effect' are used interchangeably. In the following SMD, we assume that MS keeps track of the time taken for a game.



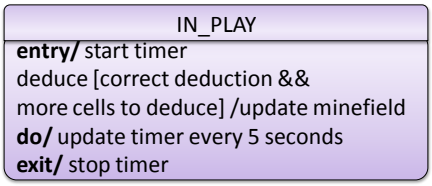
Alternatively, we can put start timer and stop timer as *internal activities* in the IN_PLAY state. An *entry activity* is an internal activity that is executed whenever object enters the state. Similarly, the *exit activity* is an internal activity that is executed whenever object exits the state.



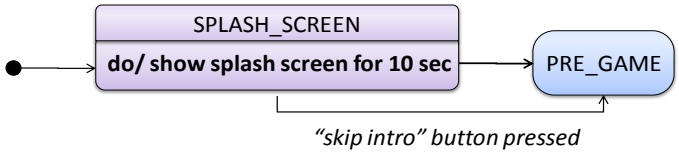
However, the above SMD does not give us the behavior we want because the *deduce* event triggers both entry and exit activities every time a new cell is deduced! That means the timer will start from zero every time a cell is deduced. To rectify this, we can make it an internal activity, as shown below.



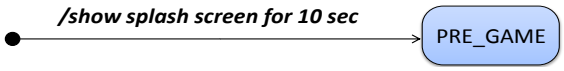
What if we need to update the "elapsed time" variable every 5 seconds during the IN_PLAY state. Note that this activity is not triggered by any external event. We can model it using an internal *do activity*, as shown below. Note that entry, do, and exit are UML keywords.



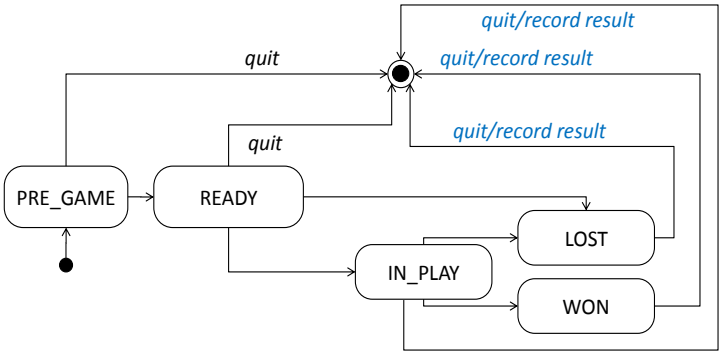
An *activity state* is a special type of a state that has a do activity and an outgoing transition that does not have any event associated with it. Once the do activity is over, the transition without any event occurs. The difference between an activity state and a regular activity is that an activity state can be interrupted (e.g. using "skip intro" button pressed event) while a regular activity cannot be interrupted. The SPLASH_SCREEN state given below is an activity state.



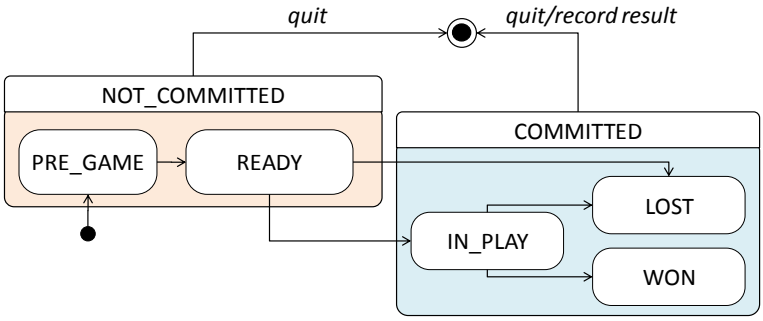
If the show splash screen for 10 sec activity cannot be interrupted, we can show it as a regular activity, as shown next.



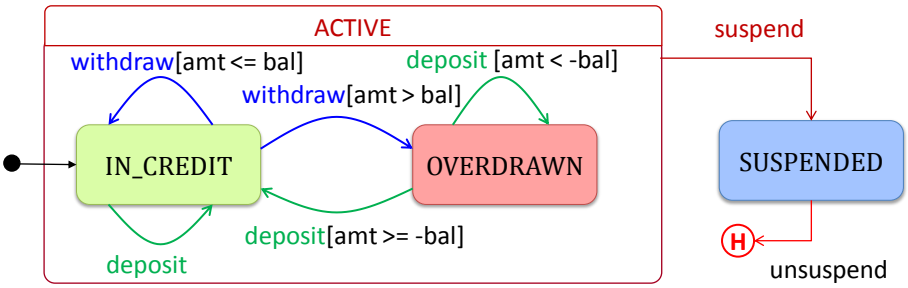
The following SMD assumes that we keep a record of the player's win/lose status. Quitting during IN_PLAY is recorded as a loss.



Apparently, multiple states handle the quit event the same way. For such a situation, we can use *superstates* e.g. COMMITTED to simplify an SMD.



A history state (an 'H' inside a circle) is used to show that state goes back to the sub state it was in when it left the super state before. The example below shows an SMD for a bank account. The unsuspend event makes the account go back to IN_CREDIT or OVERDRAWN, whichever state it was in before the account was suspended.



amt -> the amount withdrawn or deposited
bal -> account balance before withdrawal/deposit

Here are the steps we can follow when modelling state-dependent behaviour.

- Identify classes that show complex state-dependent behaviour. (Most classes do not fall into this category) e.g. Cell
- Identify states e.g. for Cell class: HIDDEN, MARKED, CLEARED
- Identify events that can be received by objects of this class (i.e. public operations) e.g. for Cell class: mark, unmark, clear
- Filter out events that are treated the same way in all states e.g. for the Logic class: getWidth(), getHeight(), …

- For each state, identify how to respond to each of the possible events.
- Simplify using internal activities, superstates, etc.

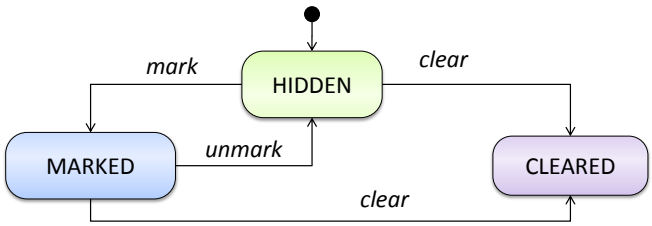
We model state-dependent behavior when it is complex enough to warrant a separate model. If the state dependent behavior becomes part of the solution, it makes sense to systematically translate those SMDs into code rather than implement the class in a way disconnected from the model we created. In this section we look at such systematic ways of implementing state-dependent behavior.

There are three approaches commonly used to implement state-dependent behavior.

- Using switch statements
- Using the state pattern
- Using state tables

First, let us learn the switch statement technique which is the simplest of the three.

The switch statement technique
Consider the Cell class and its lifecycle behavior is given in the following SMD.



Here are the steps we follow in this approach:

Step 1: Define states

- Enumerate the states. We can use an enum construct for this.
- Define a variable to store the current state (for ease of reference, let us call this variable the 'state variable').

The code below illustrates how it is done using Java.

```
public class Cell {

  enum CELL_STATE {CLEARED, MARKED, HIDDEN}

  private CELL_STATE state ;


  public Cell(int x, int y) {
     state = CELL_STATE.HIDDEN;
  }
  ...

} //end class
```

Step 1: Define states
A variable to hold current state

Step 2: Set Initial state in constructor

Step 2: Record the object's current state

In the constructor, set the state variables to the initial state.

The code above illustrates this step as well.

Step 3: Implement each event as an operation

An event in the SMD corresponds to an operation in the class. Here, we take one event at a time and implement the corresponding operation to match the SMD. The logic of the

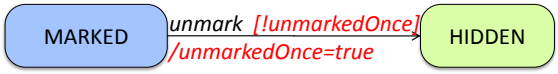
operation is handled by a switch statement that is controlled by the state variable. In the diagram below, we show how we apply this technique to the mark() operation and the clear() operation. It also shows (in the center) the generic form of such an operation.

```
public void mark () {
    switch (state) {
      case HIDDEN:
        state = CELL_STATE.MARKED;
        break;
      case CLEARED:
        break; //do nothing
      case MARKED:
        break; //do nothing
    }
  }
```

```
public void event (...) {
    switch (state) {
      case STATE_N:
        //behavior for event
        //      when in STATE_N
        break;
      ...
    }
  }
```

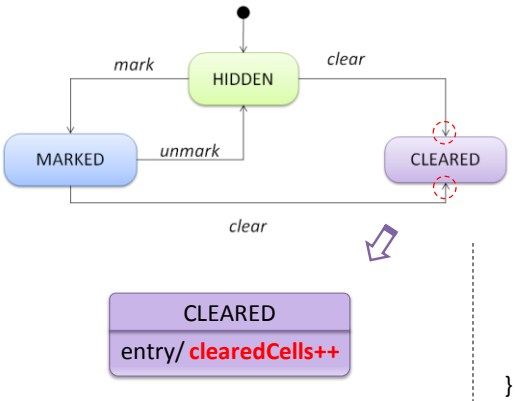
```
public void clear () {
    switch (state) {
      case HIDDEN:
        state = CELL_STATE.CLEARED;
        break;
      case MARKED:
        state = CELL_STATE.CLEARED;
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

Suppose we want to increase the difficulty of the game by not allowing more than one unmark per Cell. Given below is how we can implement the resulting guards and activities.



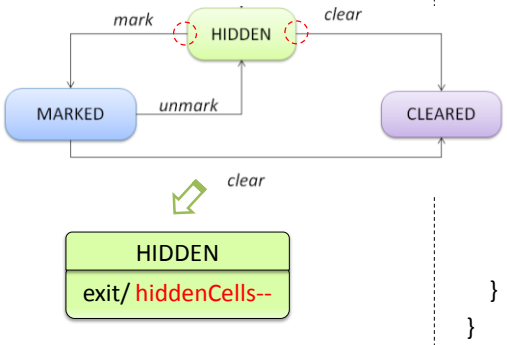

```
public void unmark () {
    switch (state) {
      case HIDDEN:
        break; //do nothing
      case MARKED:
        if (!unmarkedOnce) {   state = CELL_STATE.HIDDEN; unmarkedOnce=true;}
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

Now, assume we have the entry activity clearedCells++ in the CLEARED state. We insert the entry activity in each place where the state is being changed from any other state to CLEARED.




```
public void clear () {
    switch (state) {
      case HIDDEN:
        state=CELL_STATE.CLEARED;
        clearedCells++;
        break;
      case MARKED:
        state=CELL_STATE.CLEARED;
        clearedCells++;
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

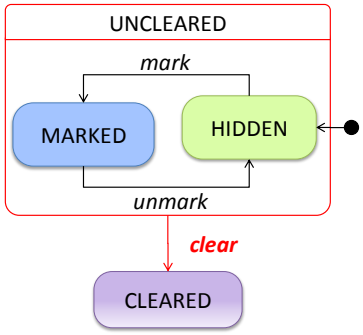
Similarly, we insert exit activities where the state variable is being changed from the state in concern (i.e. the one that has the exit activity) to another state. The example below shows how to insert the HIDDEN state's exit activity into the clear() operation. Note that according to the SMD, there are two ways to exit the HIDDEN state (clear() operation and the mark() operation), both of which need to incorporate the exit activity.



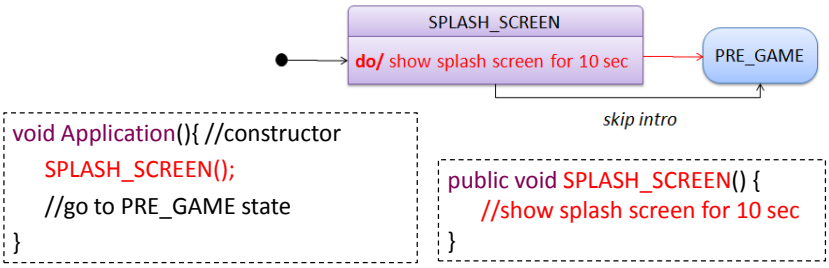

```
public void clear () {
    switch (state) {
      case HIDDEN:
        hiddenCells--;
        state=CELL_STATE.CLEARED;
        break;
      case MARKED:
        state=CELL_STATE.CLEARED;
        break;
      case CLEARED:  break; //do nothing
    }
  }
```

When superstates are involved, we simply group all the sub-states together in the switch statement. Note that the switch statement does not have a separate case for the superstate. Neither do we declare the superstates as a member of the enumeration.

```
public void clear () {
    switch (state) {
      case HIDDEN:
      case MARKED:
        state = CELL_STATE.CLEARED;
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

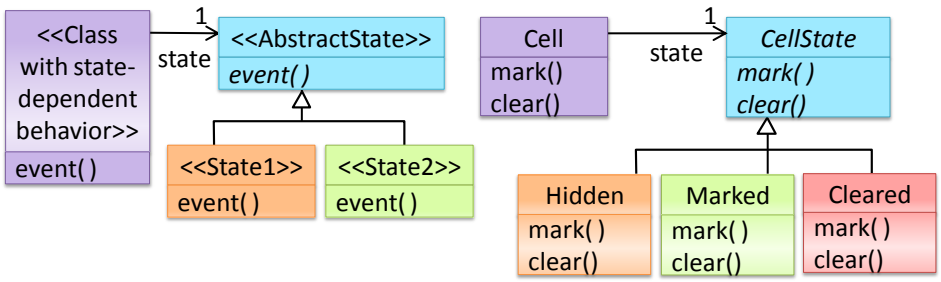


Activity states can be implemented as operations. When we want to go into an activity state, we simply call the operation that implements the activity state.



Using the State pattern

In this approach, we use inheritance and polymorphism to implement an SMD. You can find more about this pattern from many other places where state pattern (a GoF pattern) is documented. The diagram given below is simply to give you a rough idea only.



Using state tables

In this approach, we specify the state transition data in a table format (e.g. in a text file or an excel file). Here is an example.

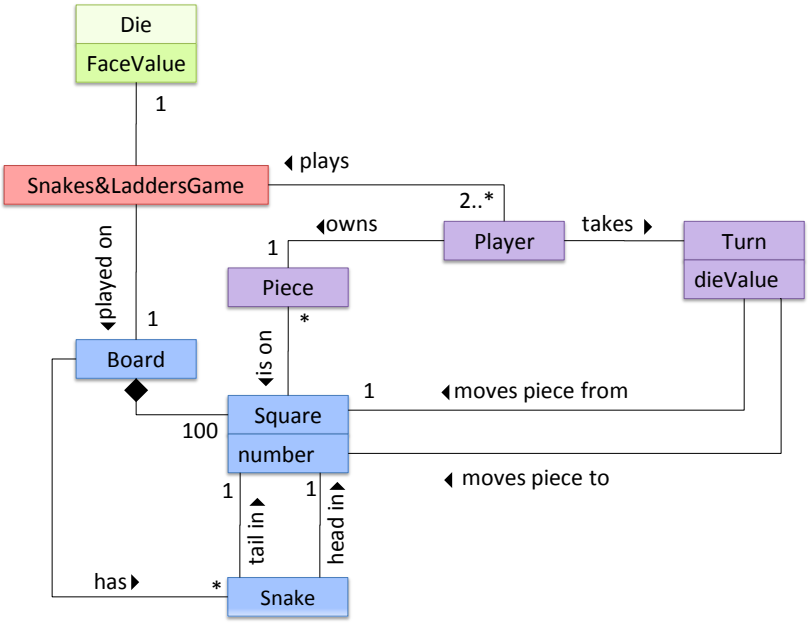
Current state	Event	Guards	Activity	Next event
PRE_GAME	new	-	-	READY
READY	deduce	Correct deduction	Start timer	IN_PLAY
...	...	...	...	...

Then, we can make our object read this table, interpret it, and behave accordingly. Note that there are code generation tools and libraries that help to reduce implementation workload when using this approach. One advantage of this approach is we can alter the object's behavior by simply altering the state table, even during runtime.

Modeling objects in the problem domains

Previously, we used UML class diagrams to model the structure of an OO solution. We can use class diagrams to model objects in the problem domain (i.e. to model how objects actually interact in the real world, *before* we emulate them in our solution). When used to model the problem domain, we call such class diagrams *conceptual class diagrams* or *OO domain models*. Usually we don't show operations or navigability on OO domain models. As an example, given below is a OO domain model of a *snakes and ladders* game.

> Description: *Snakes and ladders* game is played by two or more players using a board and a die. The board has 100 squares marked 1 to 100. Each player owns one piece. Players take turns to throw the die and advance their piece by the number of squares they earned from the die throw. The board has a number of snakes. If a player's piece lands on a square with a snake head, the piece is automatically moved to the square containing the snake's tail. Similarly, a piece can automatically move from a ladder foot to the ladder top. The player whose piece is the first to reach the 100th square wins.



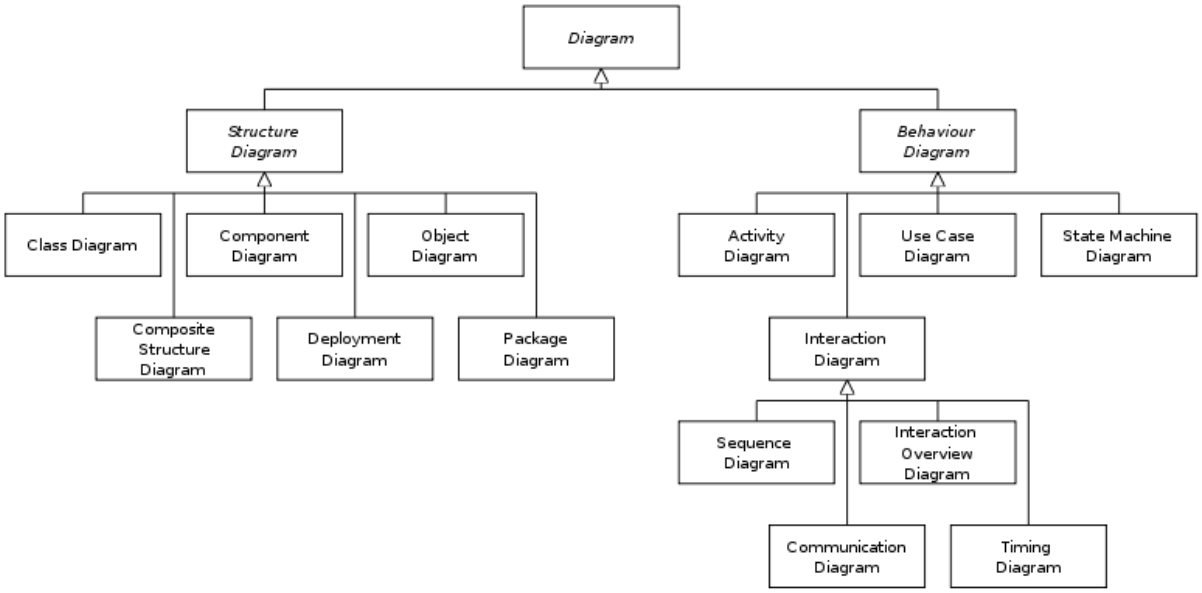
The above OO domain model omits the ladder class for simplicity. It can be included in a similar fashion to the Snake class.

Note that OO domain models do not contain solutions-specific classes (i.e. classes that are used in the solution domain but do not exist in the problem domain). For example, a class called DatabaseConnection could appear in a class diagram but not usually in an OO domain model because DatabaseConnection is something related to a software solution but not an entity in the problem domain.

Also note an OO domain model, just like a class diagram, represents the class *structure* of the problem domain and not their behavior. To show behavior we should use other diagrams such as sequence diagrams.

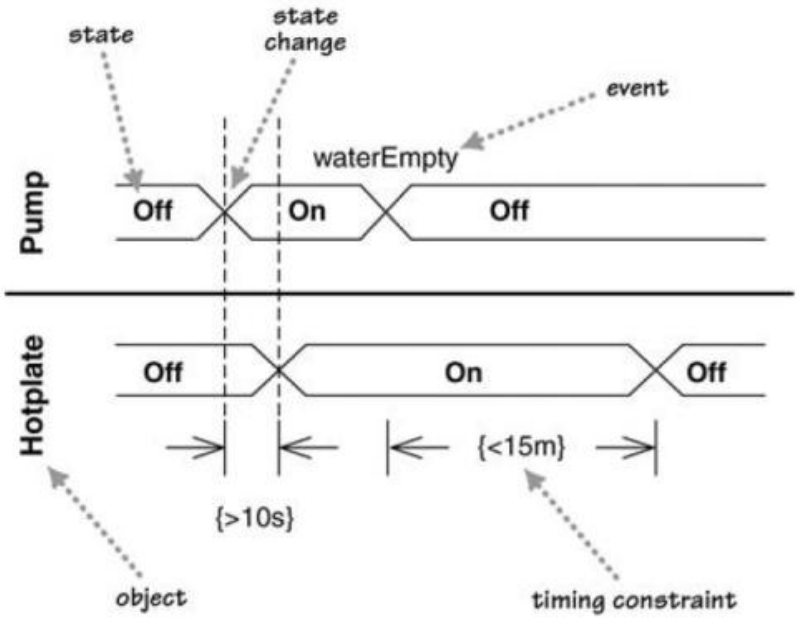
Other UML models

So far we covered following UML models: Class diagrams (including OO domain models), object diagrams, activity diagrams, use case diagrams, state machine diagrams and sequence diagrams. As shown by the domain model of UML diagrams given below, there are seven other UML diagrams: Component diagrams, Composite Structure diagrams, Deployment diagrams, Package diagrams, Communication diagrams, Interaction overview diagrams, and Timing diagrams.

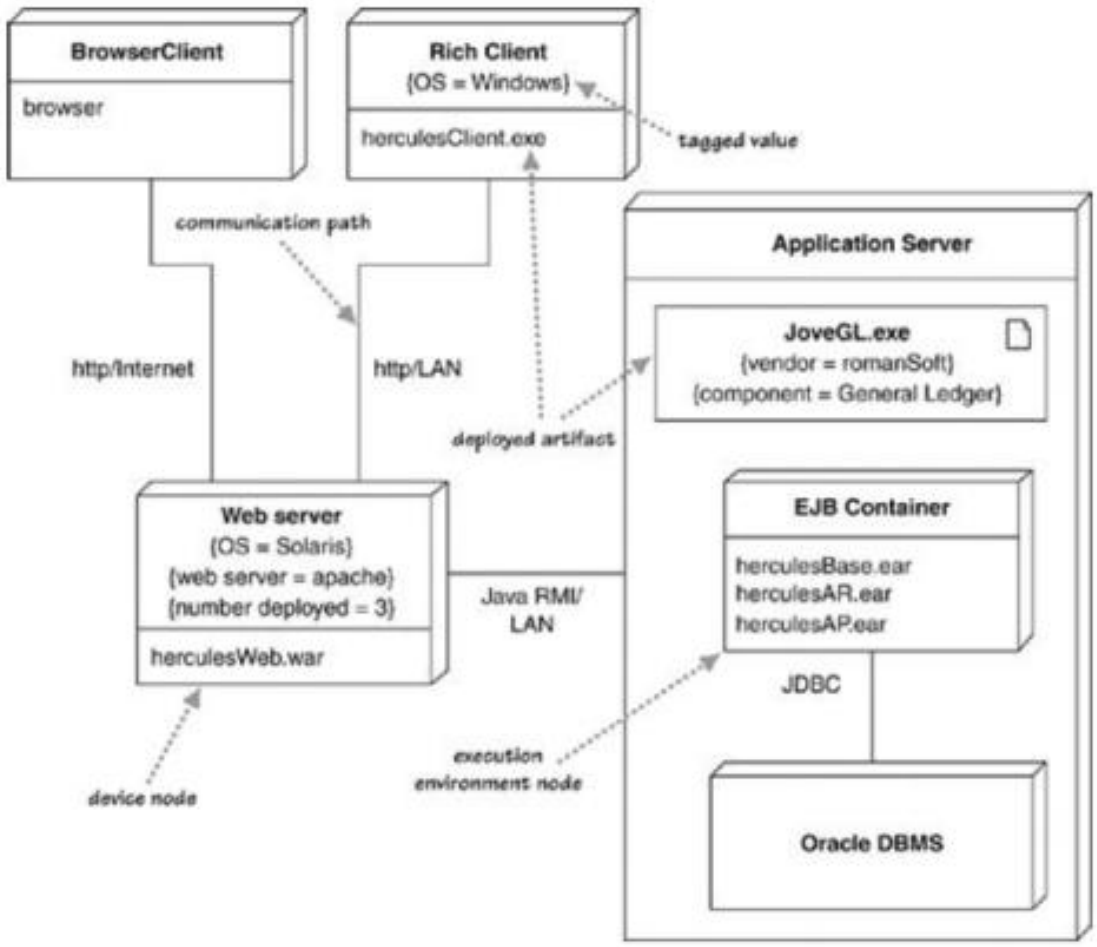


A brief overview of those seven are given below [diagrams adapted from the excellent book *UML Distilled* (3e) by Martin Fowler]. These are given here for completeness sake and are not examinable.

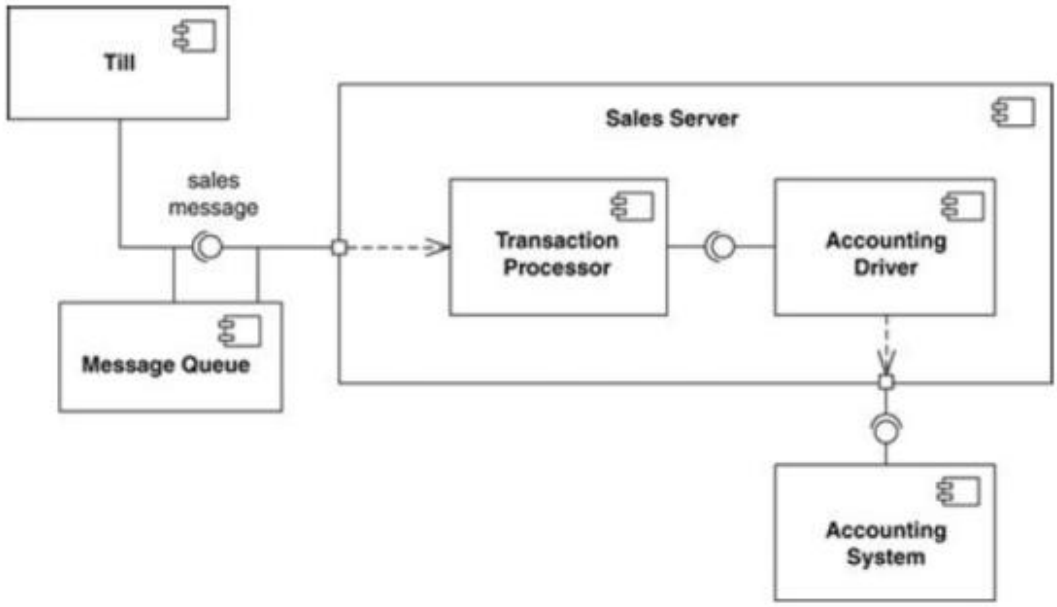
Timing diagrams focus is on timing constraints.



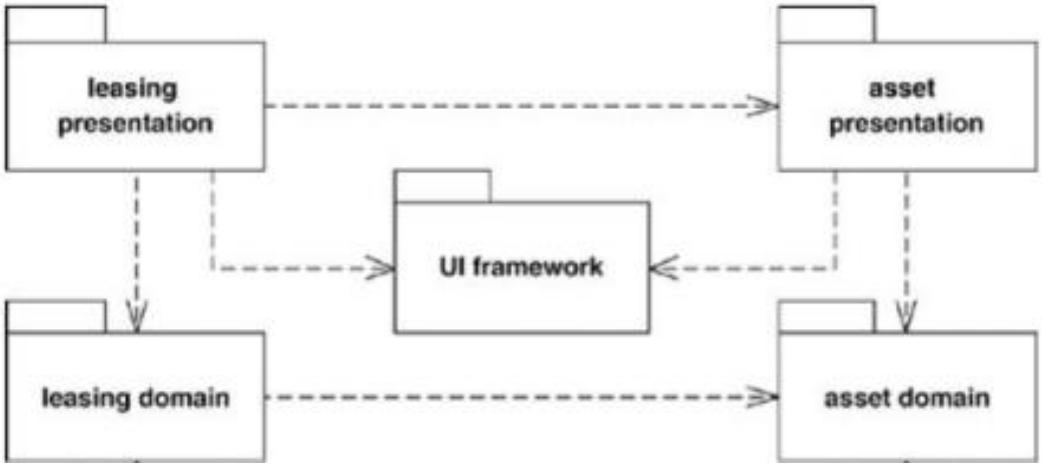
Deployment diagrams show a system's physical layout, revealing which pieces of software run on which pieces of hardware.



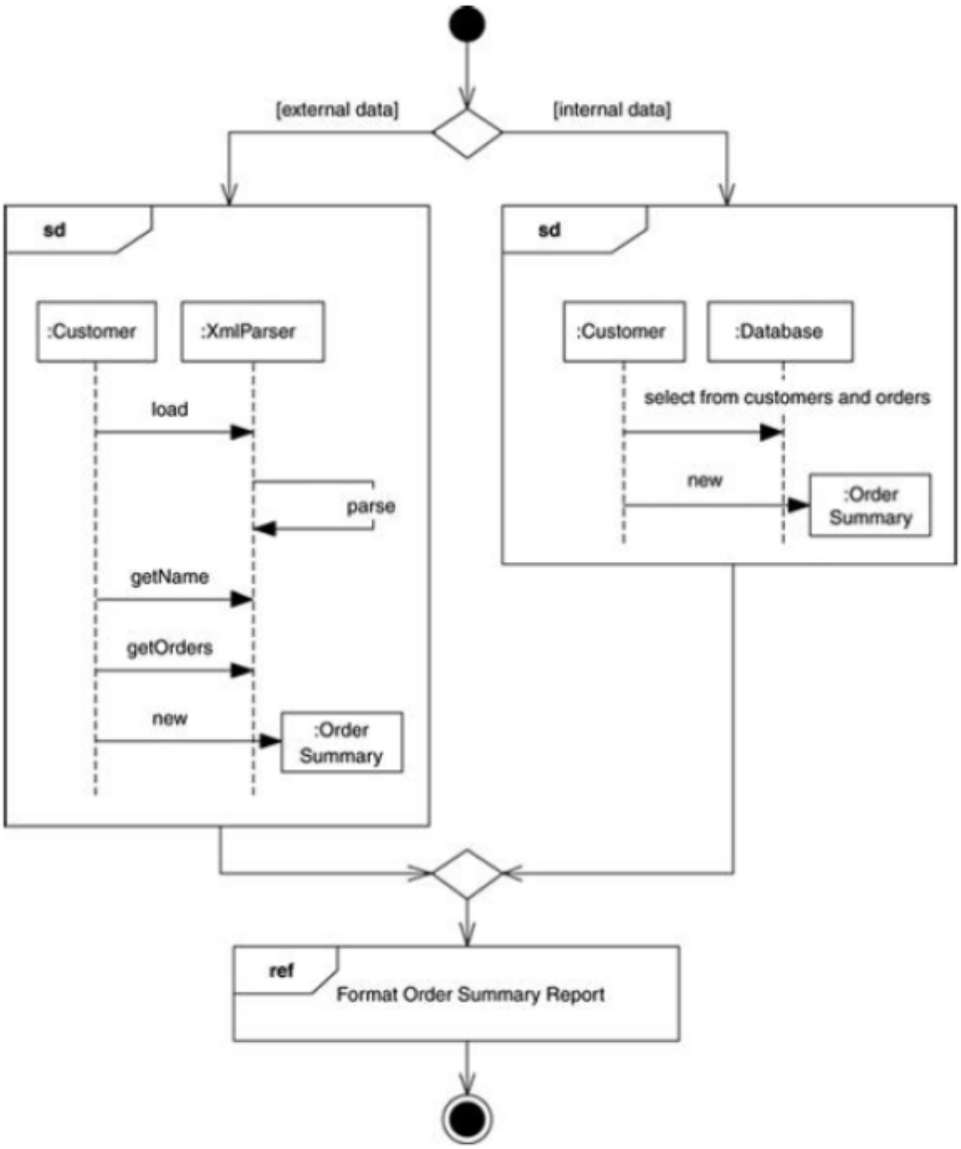
A **component diagram** is used to show how a system is divided into components and how they are connected to each other through interfaces.



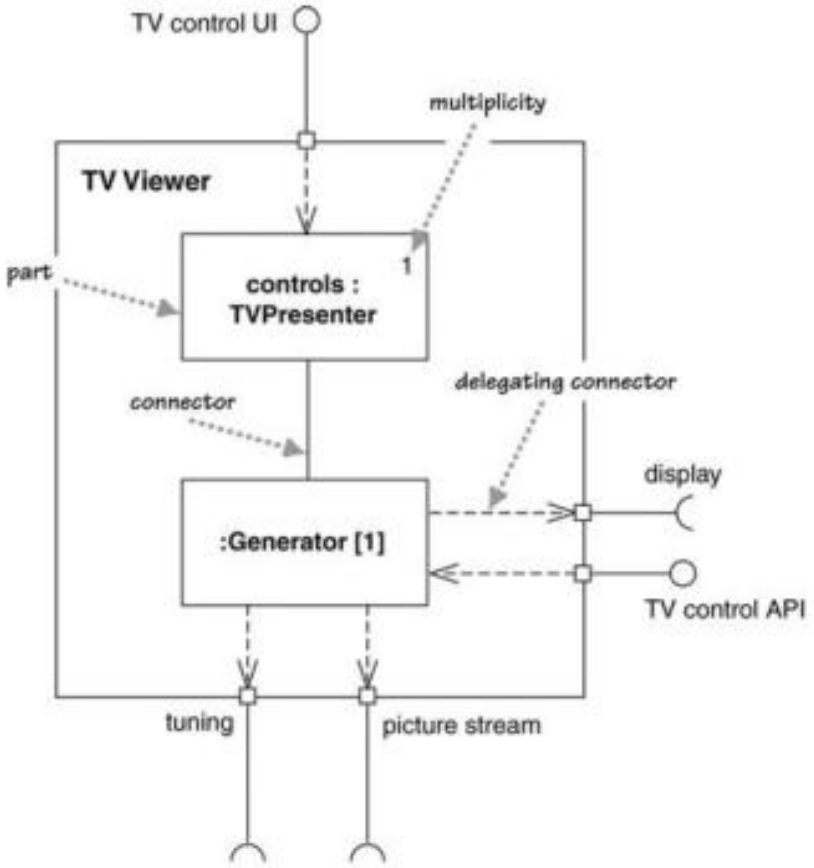
A package diagram shows packages and their dependencies. A package is a grouping construct for grouping UML elements (classes, use cases, etc.).



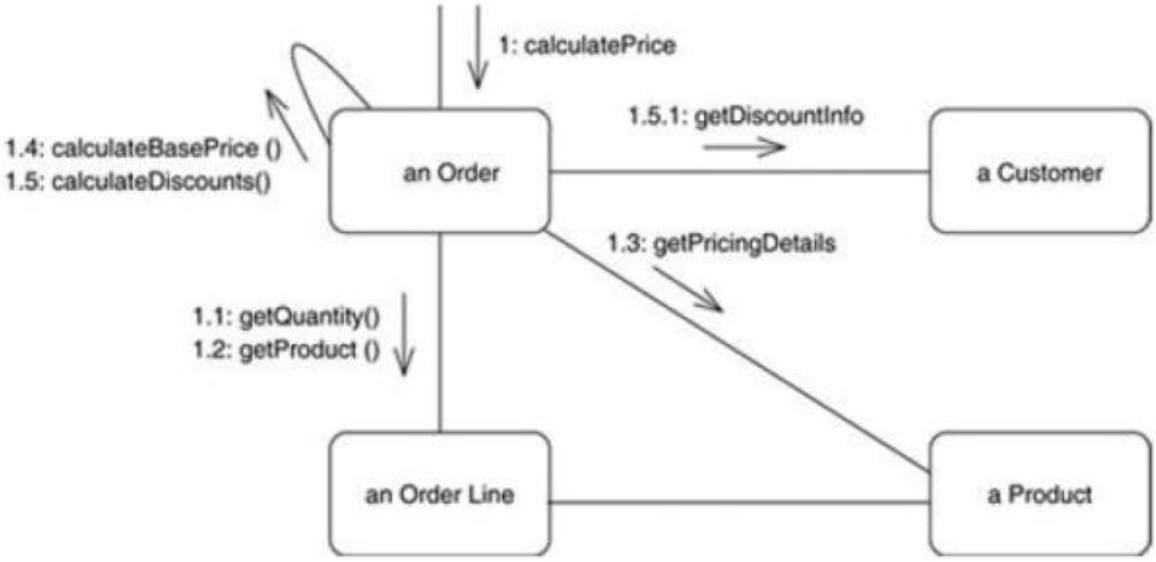
Interaction overview diagrams are a combination of activity diagrams and sequence diagrams.



A **composite structure diagram** hierarchically decomposes a class into its internal structure.



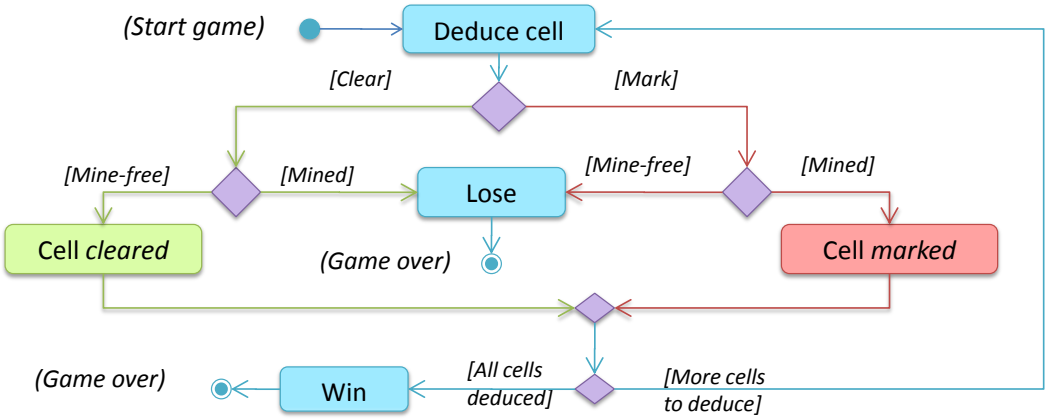
Communication diagrams are like sequence diagrams but emphasize the data links between the various participants in the interaction rather than the sequence of interactions.



Worked examples

[Q1]

Given below is the high-level game logic of the Minesweeper, drawn from the point of view of the player.



Incorporate the following new features to the above AD.
(a) timing

Description: The game keeps track of the total time spent on a game. The counting starts from the moment the first cell is cleared or marked and stops when the game is won or lost. Time elapsed is shown to the player after every mark/clear operation.

(b) standing_ground

Description: At the beginning of the game, the player chooses five cells to be revealed without penalty. This is done one cell at a time. If the cell so selected is mined, it will be marked automatically. The objective is to give some "standing ground" to the player from which he/she can deduce remaining cells. The player cannot mark or clear cells until the standing ground is selected.

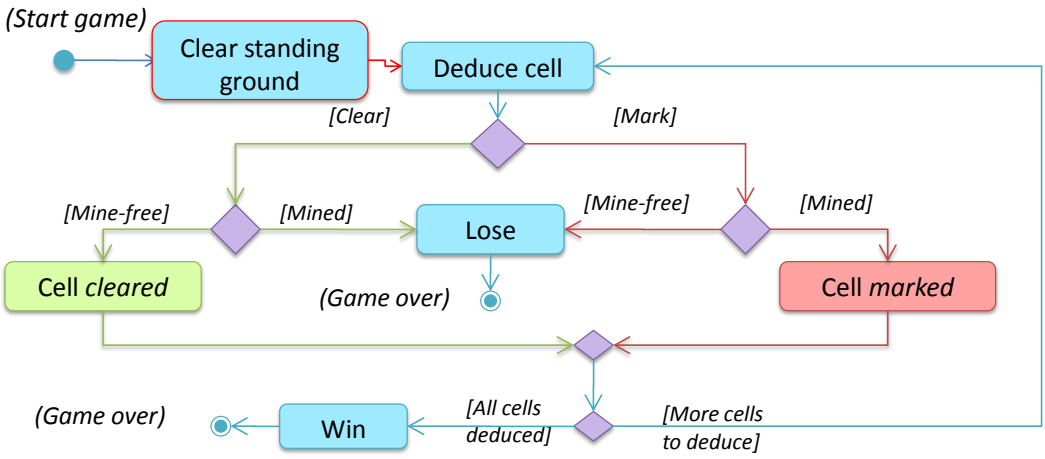
(c) tolerate

Description: Marking a cell incorrectly is tolerated as long as the number of cells does not exceed the total mines. Marked cells can be unmarked. The player is not allowed to mark more cells if the total number of marked cells equals the total number of mines.

[A1]

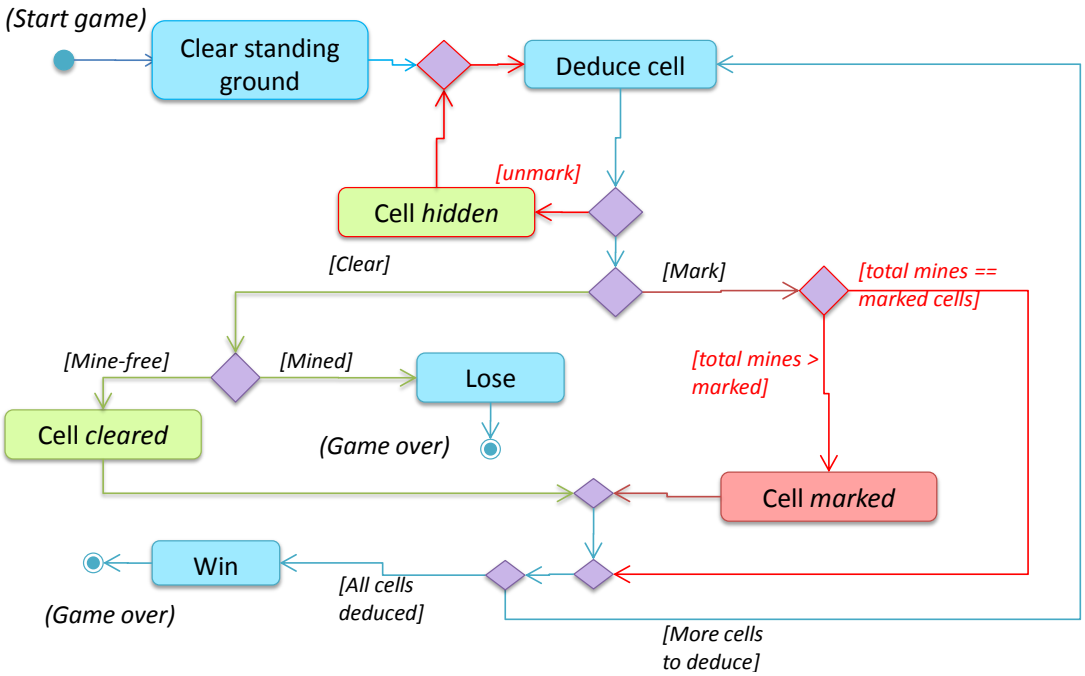
(a) No change to the AD

After incorporating (b)



Given above is a minimal change. It is OK to show more details of the 'clear standing ground' action or show it as a separate AD.

After incorporating (c)



Note that some actions/paths have been deleted. The above diagram uses a diamond as either a branch or a merge (but not both). It is ok to use a diamond as both a merge and branch, as long as it does not lead to ambiguities.

[Q2]

You are designing software for an emergency phone to be installed in security posts around a high security zone. It has a mouth piece, a speaker, and the following three buttons:

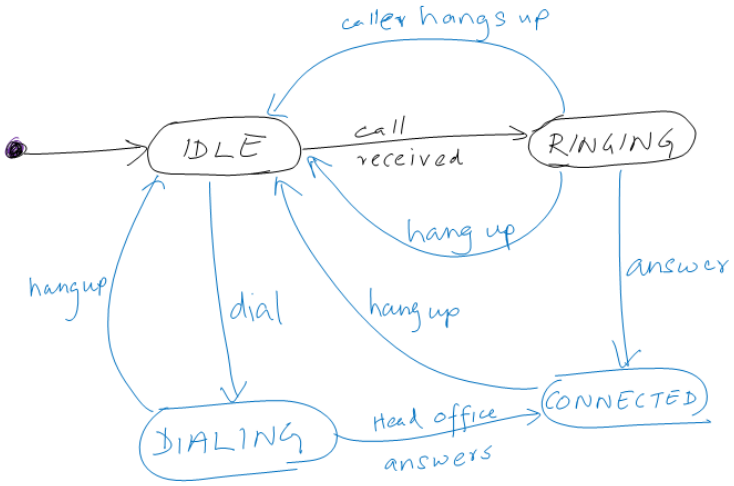
- *answer* button – answers an incoming call
- *hangup* button – terminates an ongoing call

- *dial* button– dials head office.

The phone rings when it receives a call. It can receive call from anywhere, but can only make calls to the head office.

Model the behavior of this phone using a state machine diagram. Use the partial state machine diagram given below as your starting point. It should capture information such as how the telephone will respond when the answer button in pressed while the phone is ringing.

[A2]



Note that we assume the phone does not go from CONNECTD to IDLE unless the 'HANGUP' button in pressed, even if the phone at the other end was hung up.

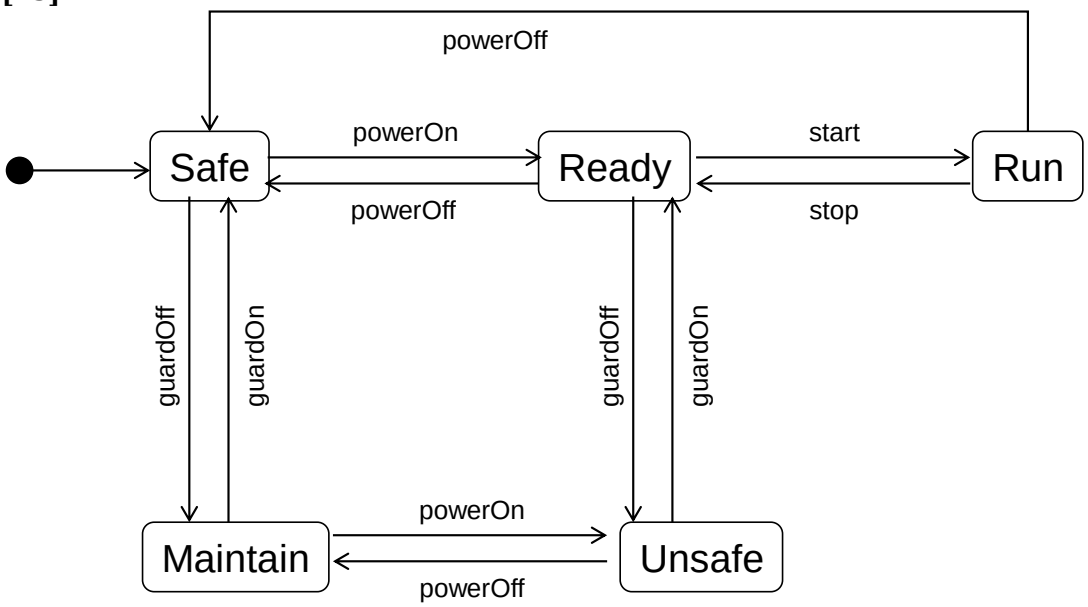
[Q3]

An XObject is controlled by events: ***powerOn, powerOff, Start, Stop, guardOn, guardOff***

Information about states of XObject are as follows :

- *Safe* – power should be off, and guard should be on
- *Ready* – power to be on, and guard to be on
- *Maintain* – power to be off, and guard to be off
- *UnSafe* – power to be on , and guard to be off
- The object transits to a ***run*** state only when power and guard are on, and a **start** event is triggered. It continues to be in ***run*** state until the **stop** event or **poweroff** event are triggered.
- Assume initial state of the object is ***Safe*** state.

[A3]



---End of Document---