[L10P1]

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. Often, multiple models of the same entity have to be created in order to understand all relevant aspects of it. For example, in addition to the class diagram, a number of sequence diagrams need to be created to capture various interesting interaction scenarios the software undergoes. In software development, models are useful in several ways:

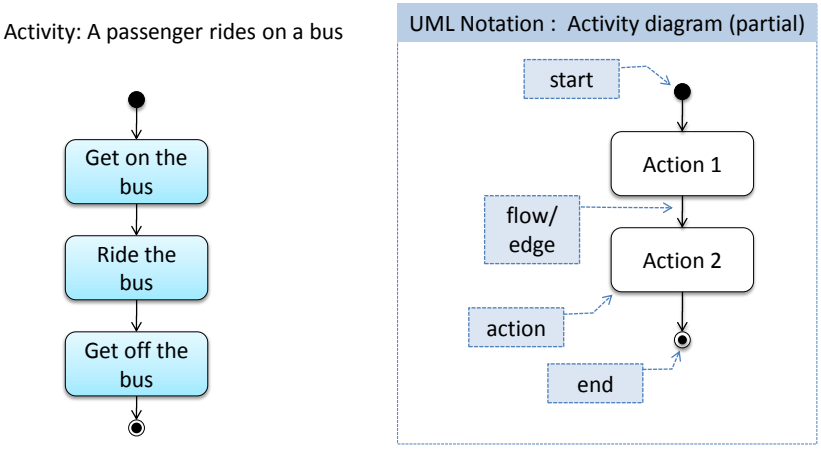
a) **To analyze** a complex entity related to software development. For example, models of the problem domain (i.e. the environment in which the software is expected to solve a problem) can be built to aid the understanding of the problem to be solved. Such models are called *domain models*. Similarly, when planning a software solution, models can be created to figure out how the solution is to be built. An architecture diagram is such a model.
b) **To communicate** information among stakeholders. Models can be used as a visual aid in discussions and documentations. To give a few examples, an architect can use an architecture diagram to explain the high-level design of the software to developers; a business analyst can use a use case diagram to explain to the customer the functionality of the system; a class diagram can be reverse-engineered from code so as to help explain the design of a component to a new developer.
c) **As a blueprint** for creating software. Models can be used as instructions for building software. *Model-driven development (MDD)*, also called *Model-driven engineering*, 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, the system is first created in the form of models. After that, the models are converted to code using code-generation techniques (usually, automated or semi-automated, but can even involve 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 sophisticated tools to generate code from models and maintain the link between models and the code. One advantage of MDD is that the same model can be used to create software for different platforms and different languages. MDD has a lot of promise, but it is still an emerging technology.

Some other models used in software development are given below.

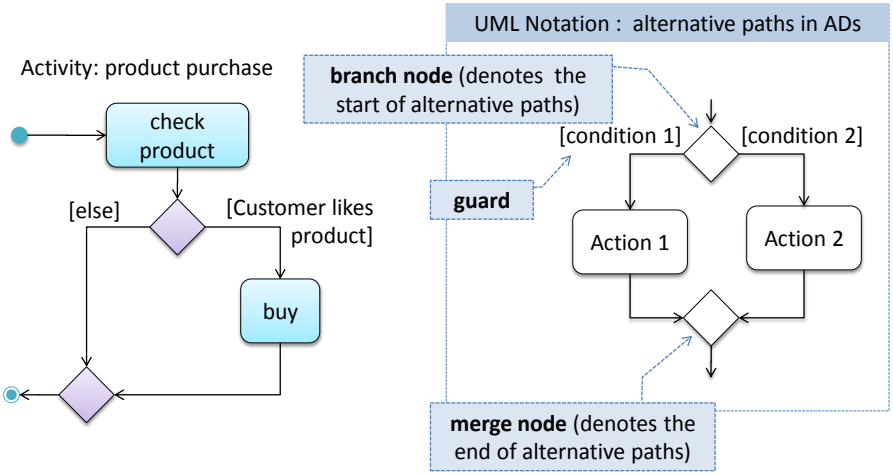
Modeling workflow

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 important if the problem that is to be solved is connected to the workflow.

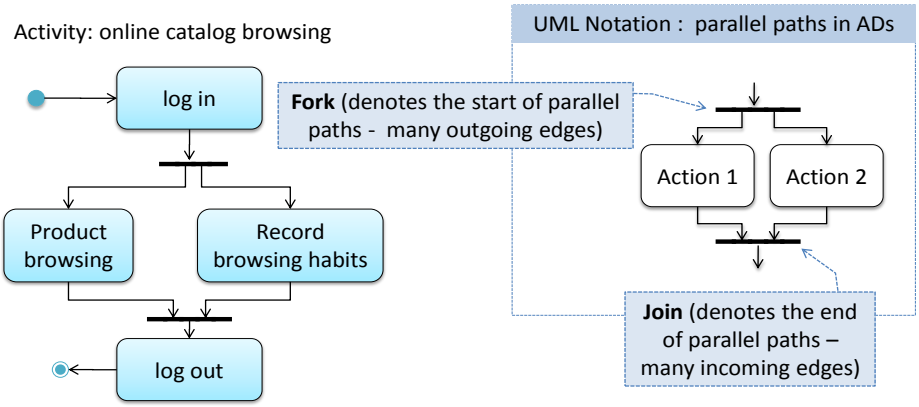
A UML *Activity diagram* (AD) can be used to describe a workflow. An activity diagram (AD) consists of a sequence of actions and control flows. An *action* is a single step in an activity. It is shown as a rectangle with rounded edges. A control flow shows the flow of control from one action to the next.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



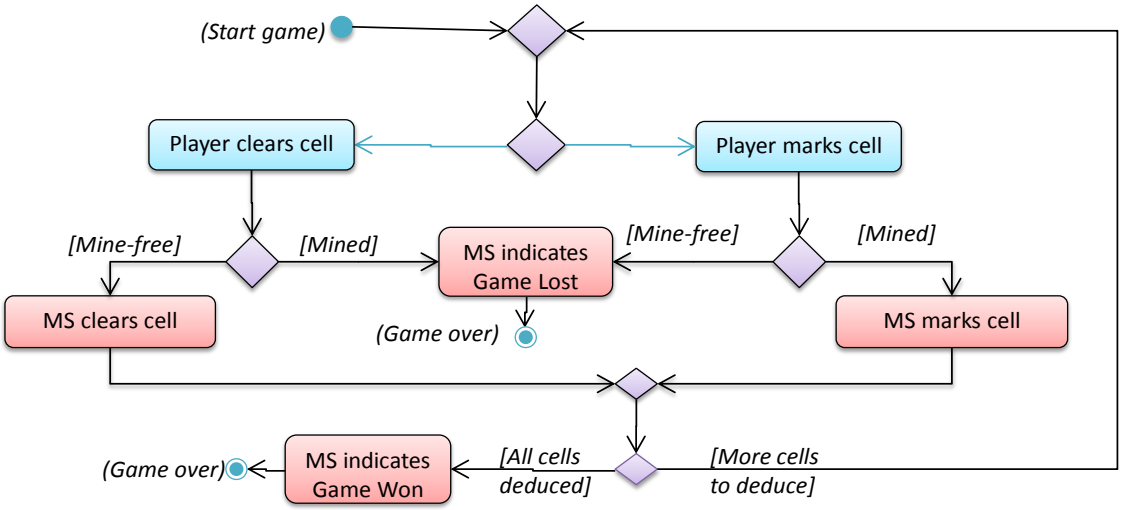
Branch nodes and *merge nodes* have the same notation: a diamond shape. They are used to show alternative (not parallel) paths through the AD. Each control flow exiting a *Branch node* has a guard condition which allows control to flow only if the guard condition is satisfied. Therefore, a *guard* is a boolean condition that should be true for execution to take a specific path.



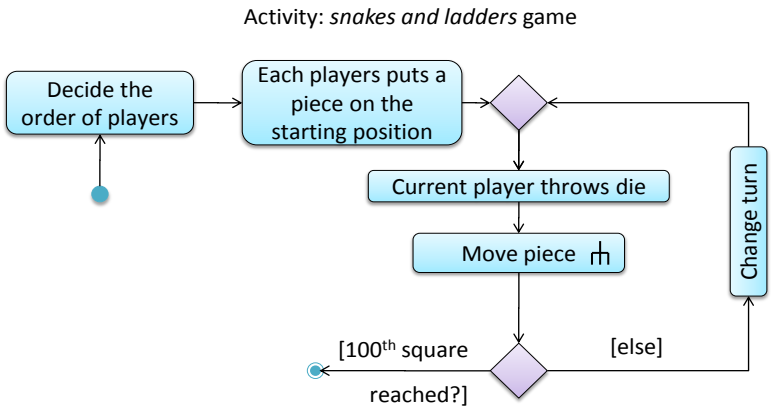
Forks and *joins* have the same notation: a bar. They indicate the start and end of concurrent flows of control. The following diagram shows an example of their use. For *join*, execution along all incoming control flows should be complete before the execution starts on the outgoing control flow.



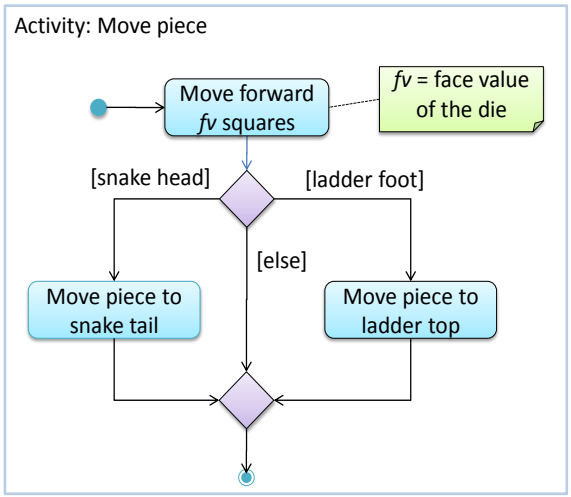
Here is the AD for the Minesweeper (MS) that shows actions done by the player and the game (MS).



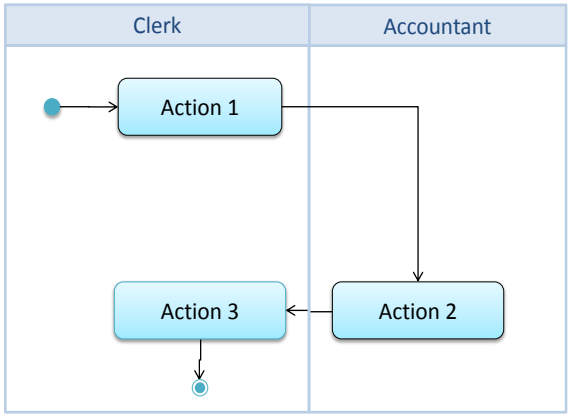
Here is the AD for a game of 'snakes and ladders'.



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



It is possible to *partition* an activity diagram to show who is doing which action. Such partitioned activity diagrams are sometime called *swimlane diagrams*.

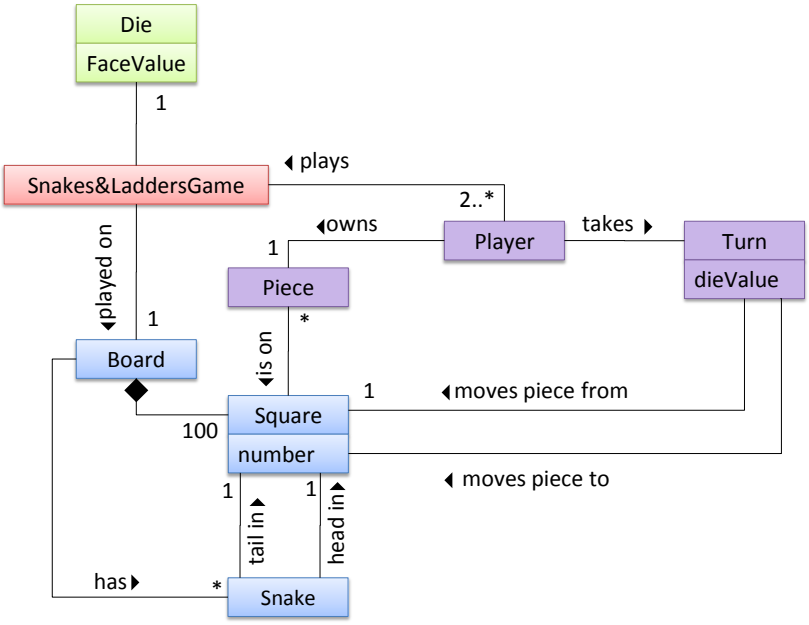


Note: Only essential elements of ADs are covered in this handout.

Modeling objects in the problem domains

Previously, UML class diagrams were used to model the structure of an OO solution. Class diagrams can also be used to model objects in the problem domain (i.e. to model how objects actually interact in the real world, *before* emulating them in the solution). When used to model the problem domain, such class diagrams are called *conceptual class diagrams* or *OO domain models*. Usually, operations or navigability are now depicted on OO domain models. As an example, an OO domain model of a *snakes and ladders* game is given below.

> 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.

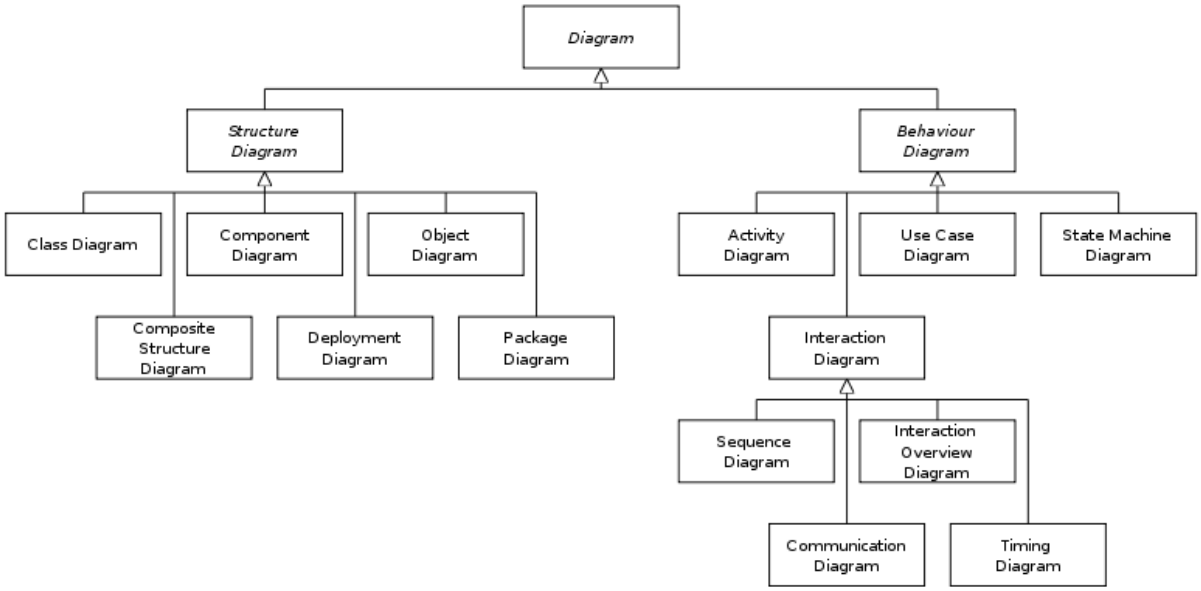
Note that OO domain models do not contain solution-specific classes (i.e. classes that are used in the solution domain but do not exist in the problem domain). For example, a class called DatabaseConnection could appear in a class diagram but not usually in an OO domain model because DatabaseConnection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

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

Domain model notation is similar to class diagram notation. However, classes in domain models do not have a compartment for methods. It is also common to omit navigability from domain models.

Other UML models

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



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

State Machine Diagrams model state-dependent behavior.

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

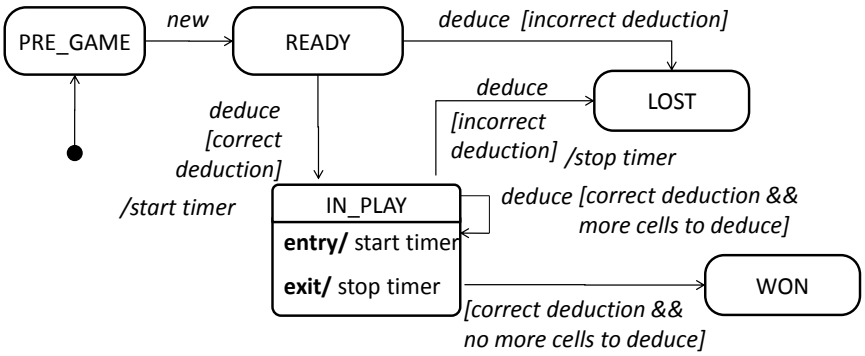
- If the CD tray is already open, it does nothing.
- If the CD tray is already in the process of opening (opened half-way), it continues to open the CD tray.
- If the CD tray is closed and the CD is being played, it stops playing and opens the CD tray.

- 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.

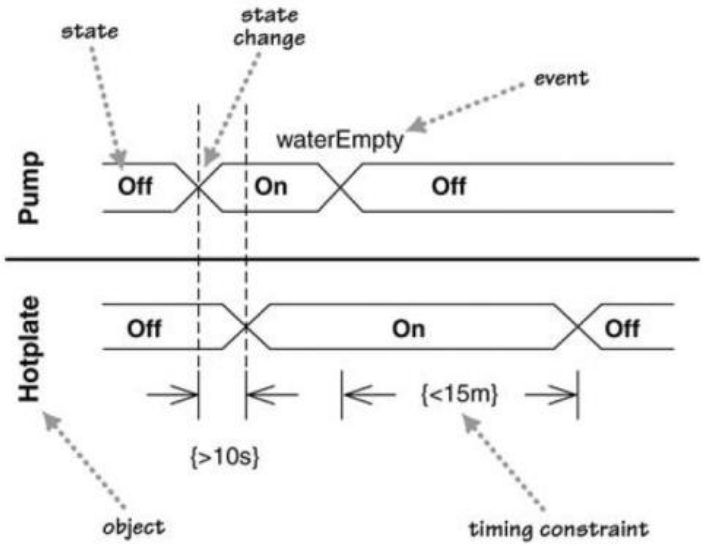
What this means is that the CD player's response to pushing the "eject CD" button depends on what it was doing at the time of the event. More generally, the CD player's response to the event received depends on its internal state. Such a behavior is called a *state-dependent behavior*.

Often, state-dependent behavior displayed by an object in a system is simple enough that it needs no extra attention; such a behavior can be as simple as a 'conditional' behavior like 'if x>y, then x=x-y'. Occasionally, objects may exhibit state-dependent behavior that is deemed too complex such that it needs to be captured into a separate model. Such state-dependent behavior can be modelled using UML *state machine diagrams* (SMD for short, sometimes also called 'state charts', 'state diagrams' or 'state machines').

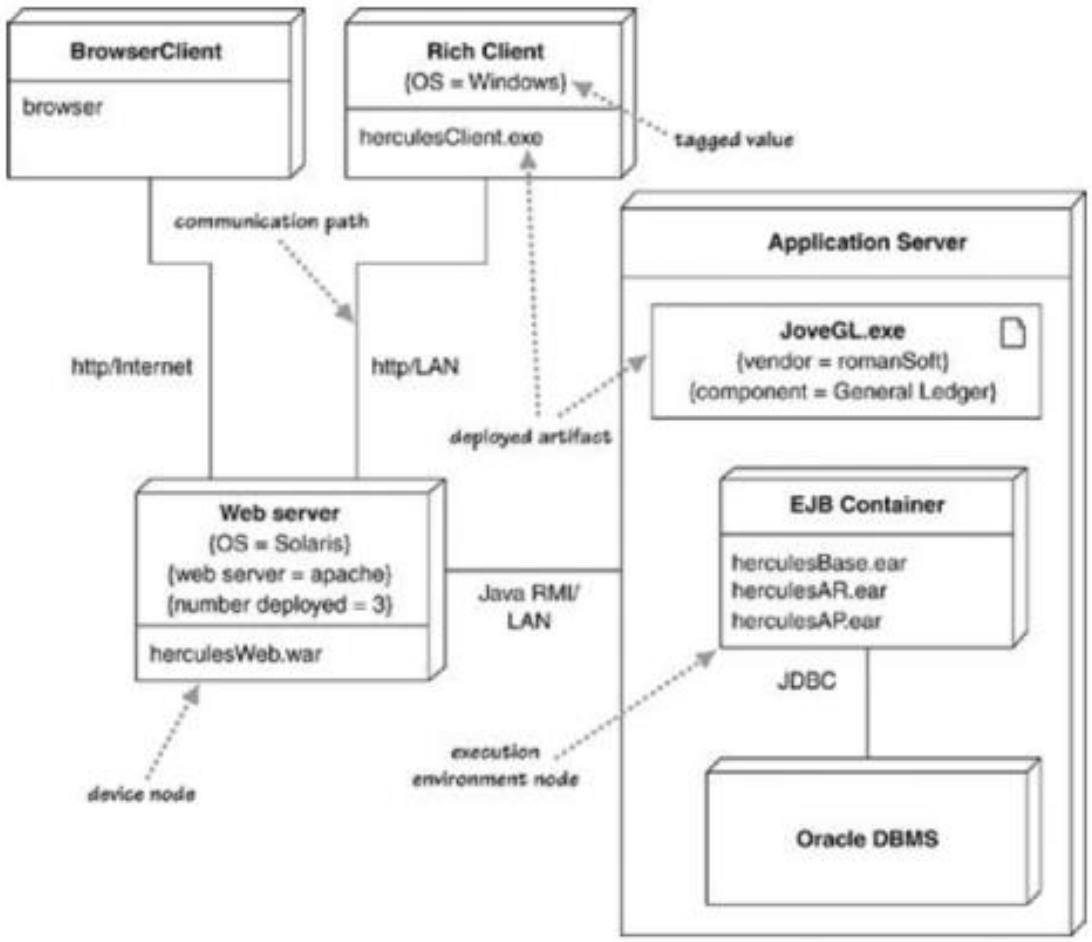
An SMD views the life-cycle of an object as consisting of a finite number of states where each state displays a unique behavior pattern. An SMD captures information such as the states an object can be in, during its lifetime, and how the object responds to various events while in each state and how the object transits from one state to another. In contrast to sequence diagrams that capture object behavior one scenario at a time, SMDs capture the object's behavior over its full life cycle. Given below is an example SMD for the Minesweeper game. More details on SMDs can be found in the appendix of this handout.



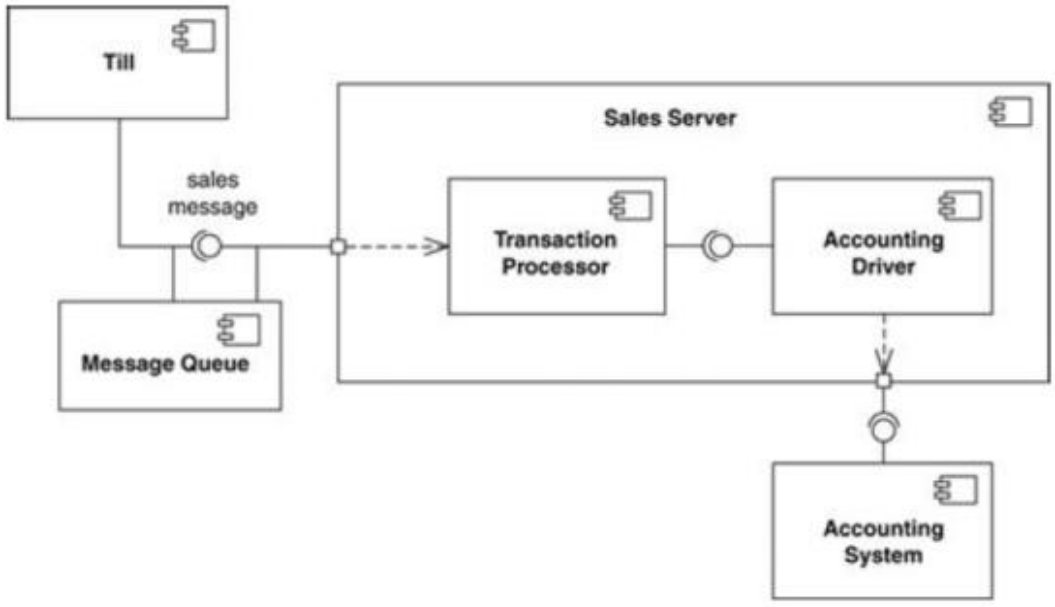
Timing diagrams focus on timing constraints.



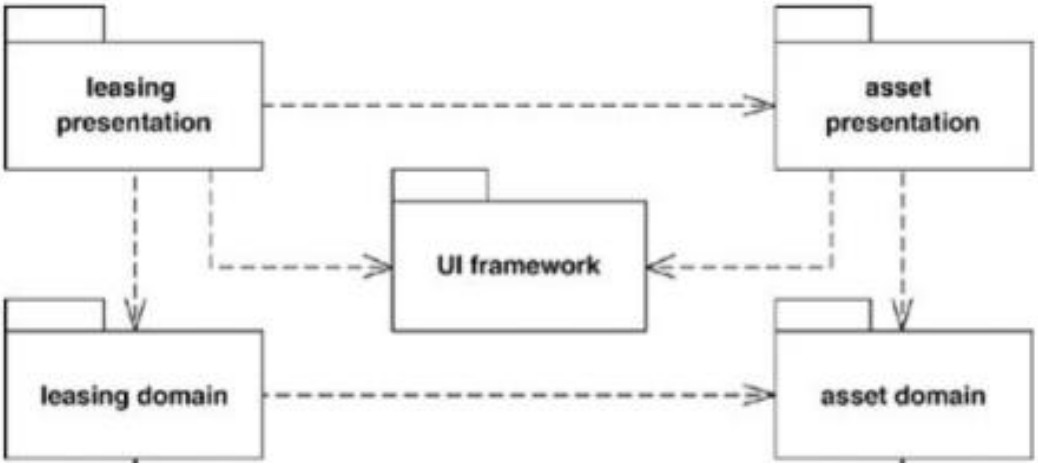
Deployment diagrams show a system's physical layout, revealing which pieces of software run on which pieces of hardware.



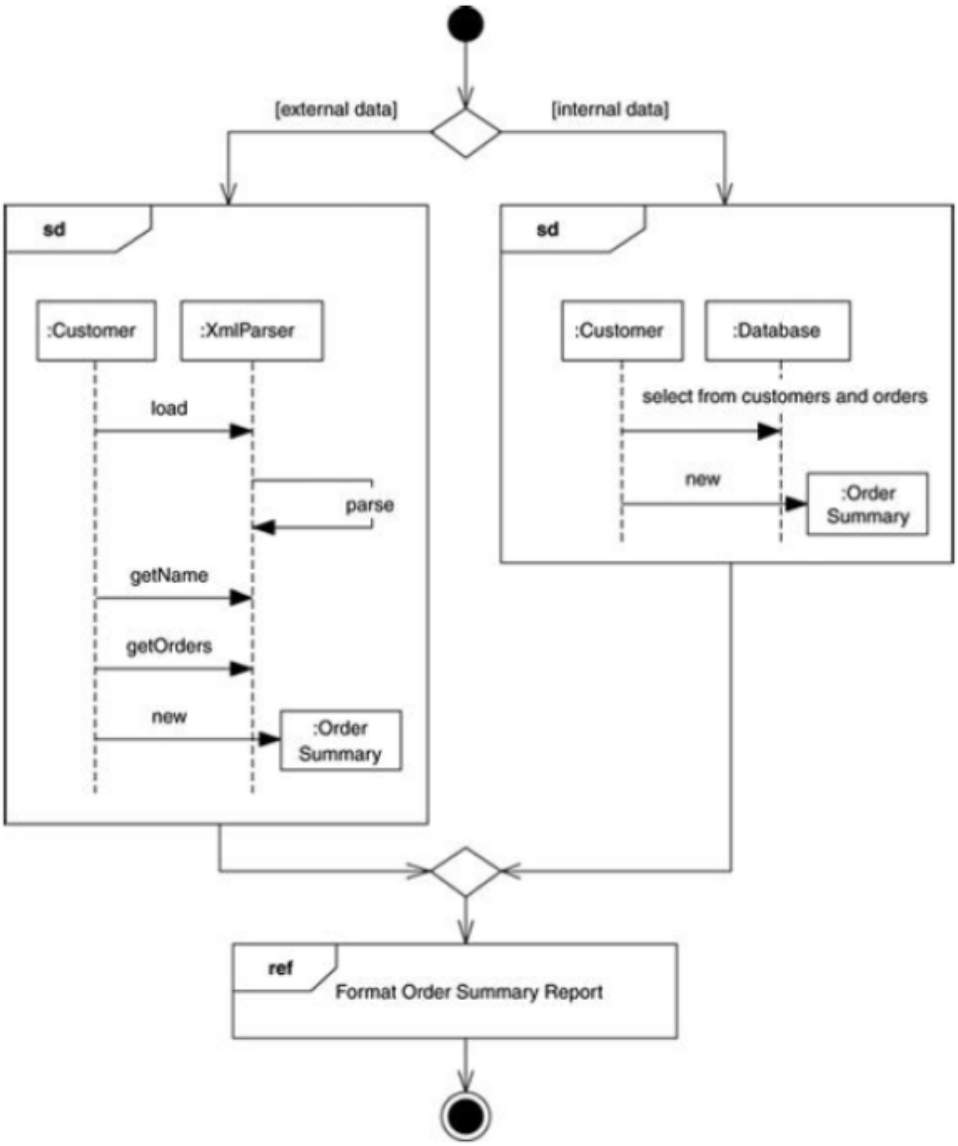
A **component diagram** is used to show how a system is divided into components and how they are connected to each other through interfaces.



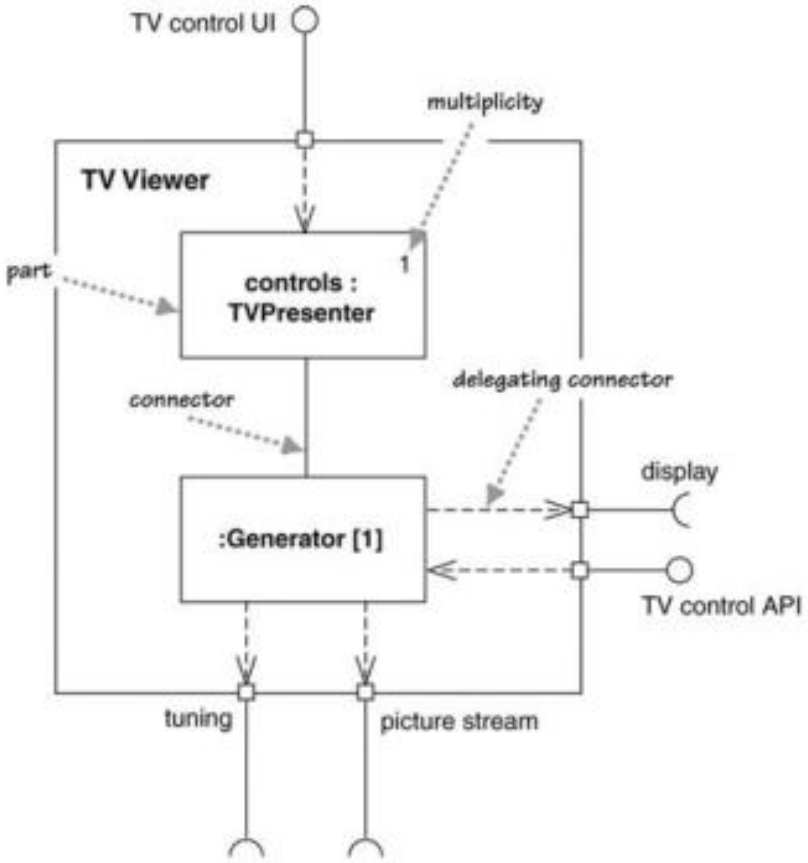
A package diagram shows packages and their dependencies. A package is a grouping construct for grouping UML elements (classes, use cases, etc.).



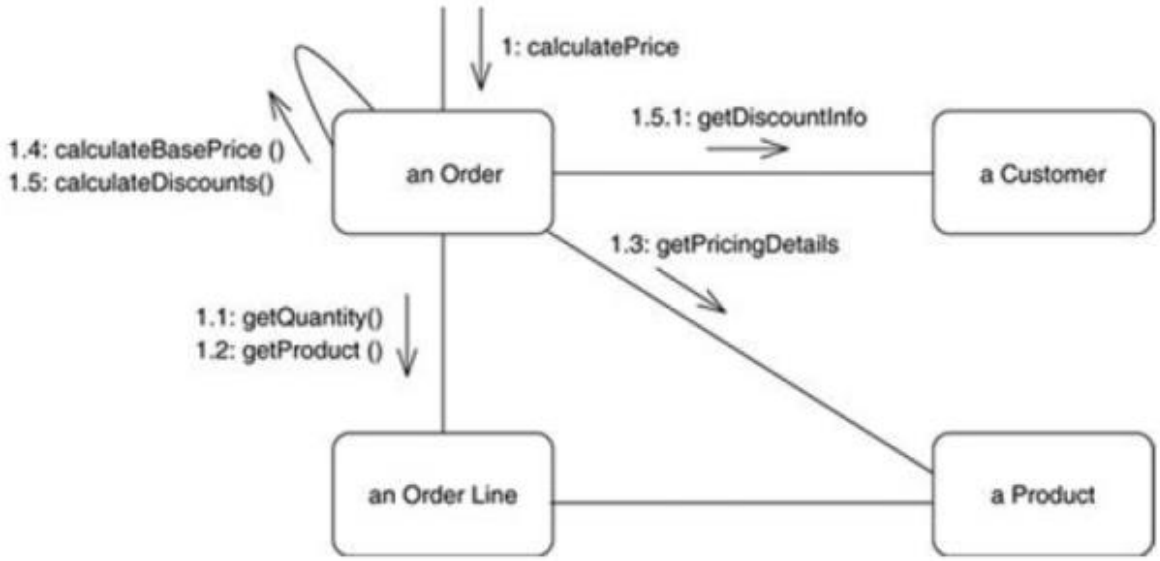
Interaction overview diagrams are a combination of activity diagrams and sequence diagrams.



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



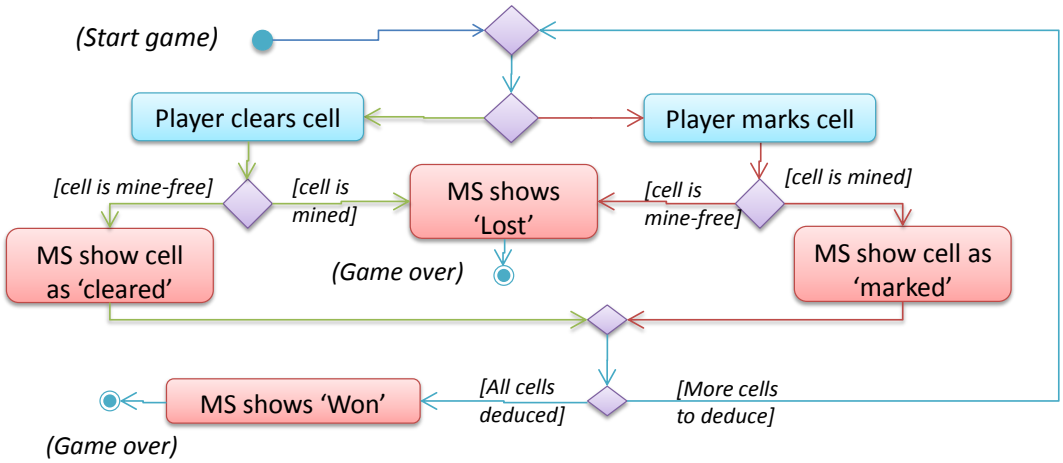
Communication diagrams are like sequence diagrams but emphasize the data links between the various participants in the interaction rather than the sequence of interactions.



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/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.

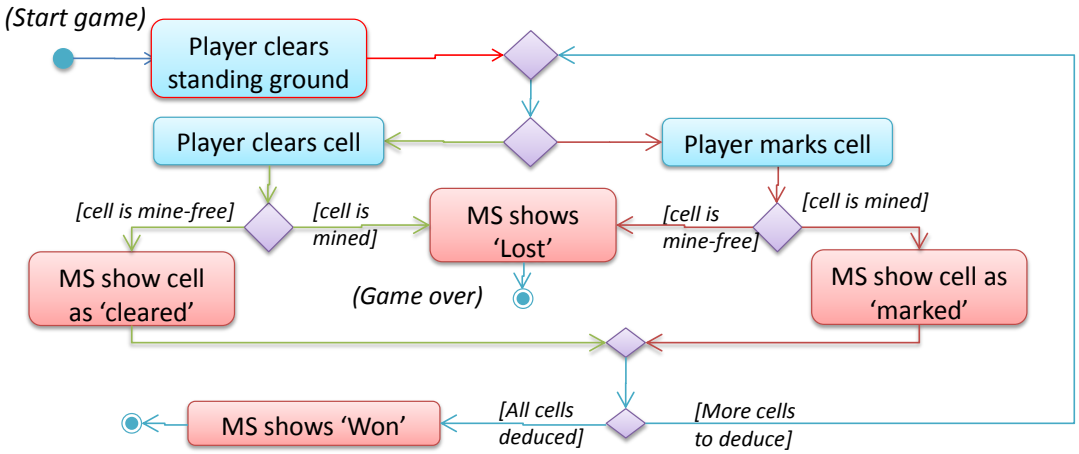
(c) tolerate

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]

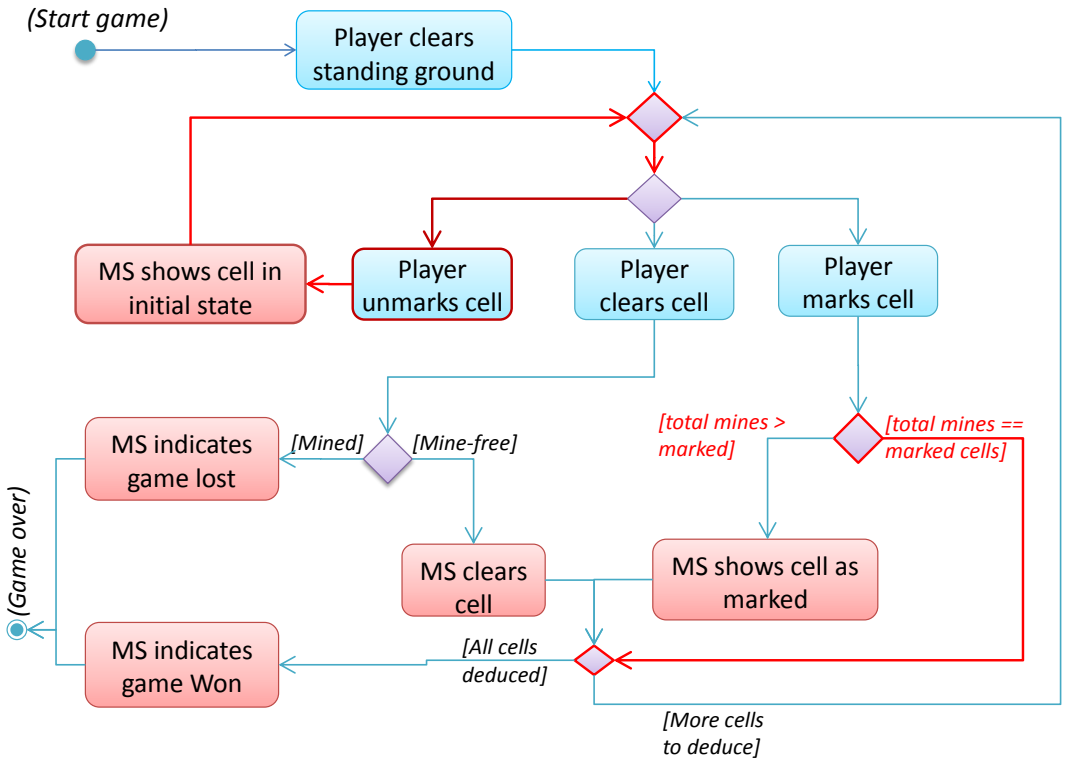
(a) No change to the AD

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)



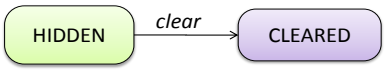
Note that some actions/paths have been deleted. The above diagram uses a diamond as either a branch or a merge (but not both). It is ok to use a diamond as both a merge and branch, as long as it does not lead to ambiguities.

L10P1.Appendix: State Machine Diagrams

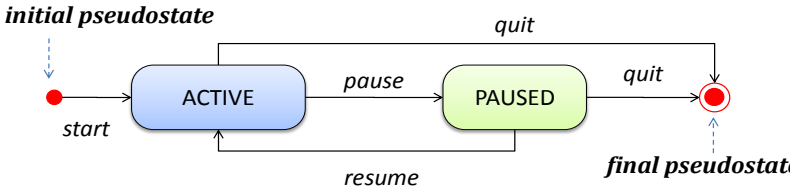
[This section is for your reference only. It is not examinable]

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

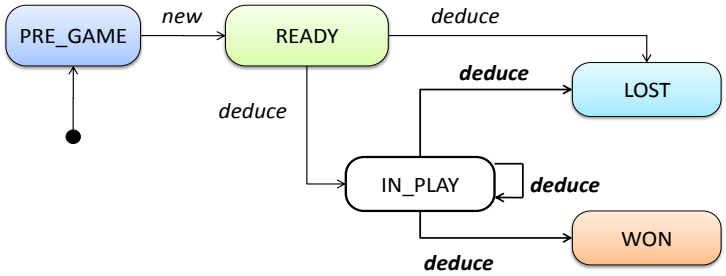
- *state*: A phase in the objects life-cycle that shows a unique behavior pattern (shown as rounded rectangles) e.g. HIDDEN, CLEARED
- *transition*: A movement from one state to another (shown as a directed arrow).
- *trigger*: A single event that causes a potential transition e.g. clear



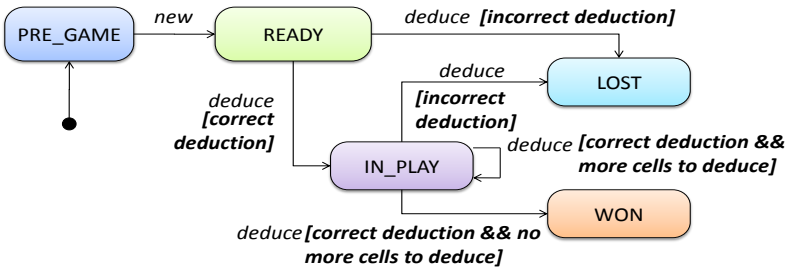
Given below is an SMD for an unidentified application. It illustrates the symbol for the *initial pseudostate* which indicates the starting state of the SMD and the *final pseudostate* which indicates the end of the lifecycle. Note that the final pseudostate can be omitted (for simplicity) if there is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



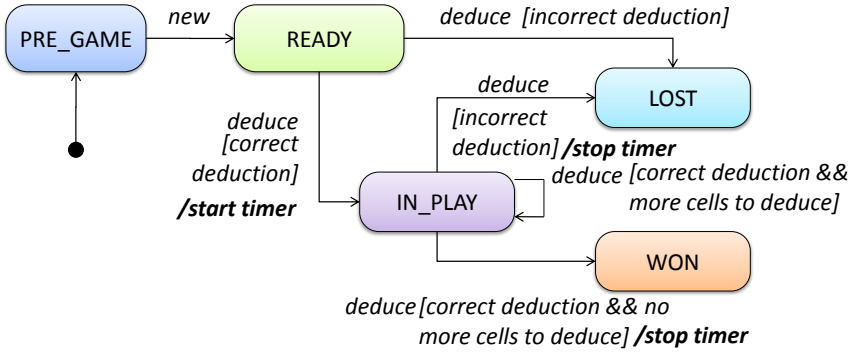
Given below is the (partial) SMD for the Minesweeper game with the assumption that the lifecycle of the whole game is managed by one object. Note that PRE_GAME means that the game has started but no Minefield has been created yet, while READY means the Minefield has been created and Minesweeper is ready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



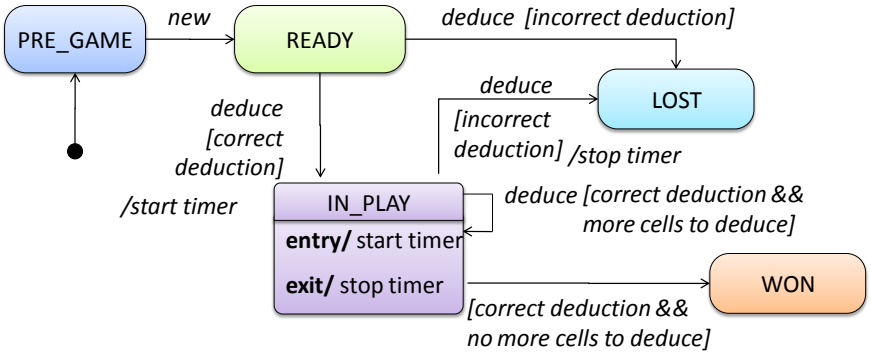
Note how the SMD is not precise on which transition to follow if the deduce event happened while in IN_PLAY state. i.e. a given event in a given state will not always produce the same transition. In simpler terms, it exhibits "random" behavior. Such SMDs are said to be *non-deterministic*. As non-deterministic SMDs are of not much use, it can be made *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.



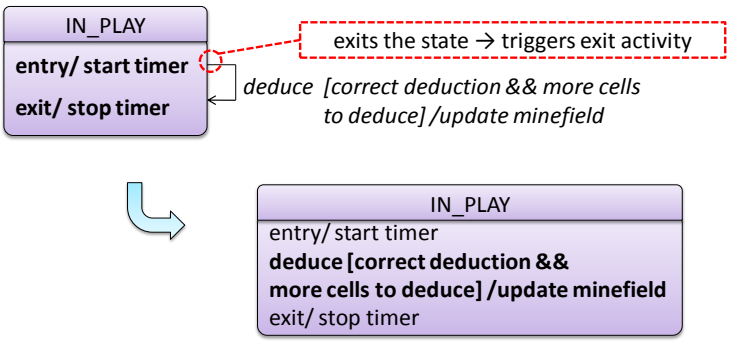
An *activity* is a behavior that takes place *during* a transition. The UML syntax is `event[guard]/activity` where all 3 components are optional. Note that in the context of SMDs, 'activity' 'action' and 'effect' are used interchangeably. In the following SMD, it is assumed that MS keeps track of the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



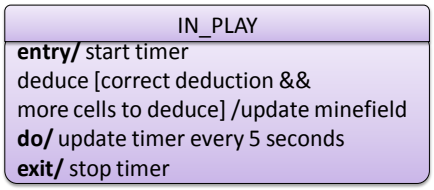
Alternatively, the start timer and stop timer can be included as *internal activities* in the IN_PLAY state. An *entry activity* is an internal activity that is executed whenever object enters the state. Similarly, the *exit activity* is an internal activity that is executed whenever object exits the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



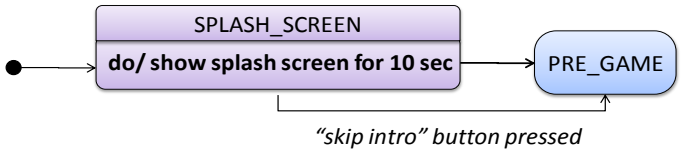
However, the above SMD does not give the desired behavior because the *deduce* event triggers both entry and exit activities every time a new cell is deduced! That means the timer will start from zero every time a cell is deduced. To rectify this, it can be made an internal activity, as shown below.



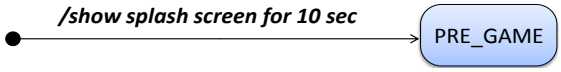
What if the "elapsed time" variable is only updated every 5 seconds during the IN_PLAY state? Note that this activity is not triggered by any external event. It can be modeled using an internal *do activity*, as shown below. Note that entry, do, and exit are UML keywords.



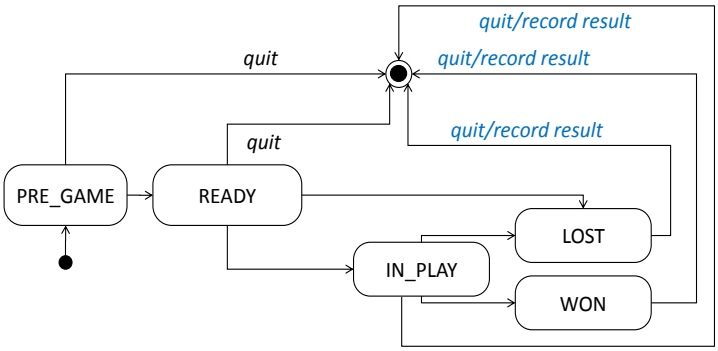
An *activity state* is a special type of a state that has a do activity and an outgoing transition that does not have any event associated with it. Once the do activity is over, the transition without any event occurs. The difference between an activity state and a regular activity is that an activity 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.



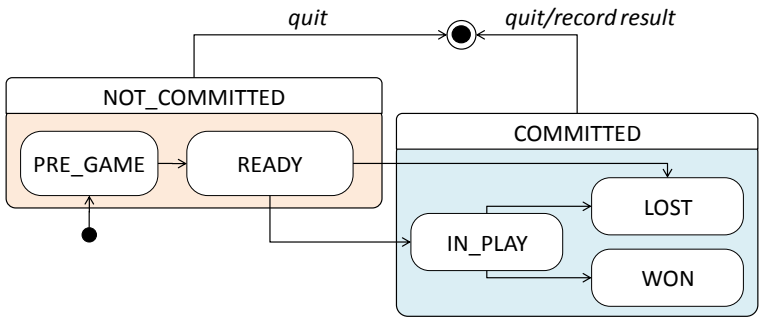
If the show splash screen for 10 sec activity cannot be interrupted, it can be depicted as a regular activity, as shown below.



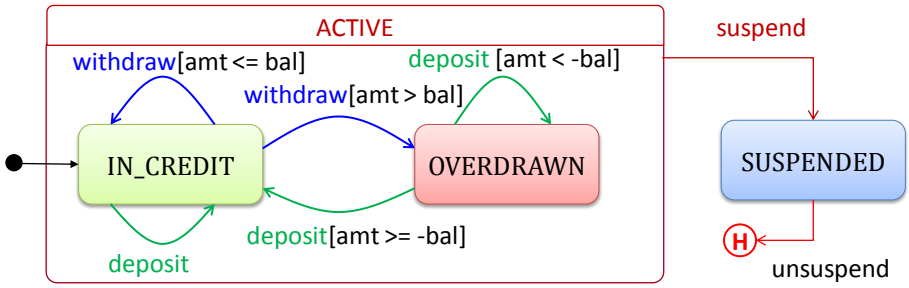
The following SMD assumes that a record is kept of the player's win/lose status. Quitting during IN_PLAY is recorded as a loss.



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



A history state (an 'H' inside a circle) is used to show that the 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 behavior.

- Identify classes that show complex state-dependent behavior. (Most classes do not fall into this category) e.g. Cell in Minesweeper
- Identify states e.g. for the Cell class: HIDDEN, MARKED, CLEARED
- Identify events that can be received by objects of this class (i.e. public operations) e.g. for the 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.

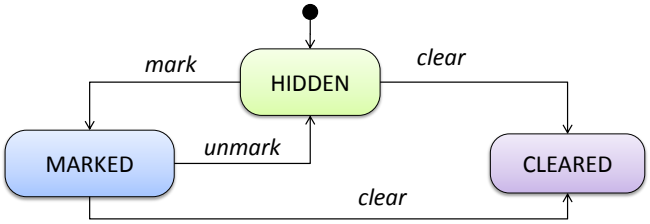
State-dependent behavior is modeled when it is complex enough to warrant a separate model. If the state dependent behavior becomes part of the solution, it makes sense to systematically translate those SMDs into code rather than implement the class in a way disconnected from the model being created. In the following, systematic ways of implementing state-dependent behavior are described.

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

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

The switch statement technique

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

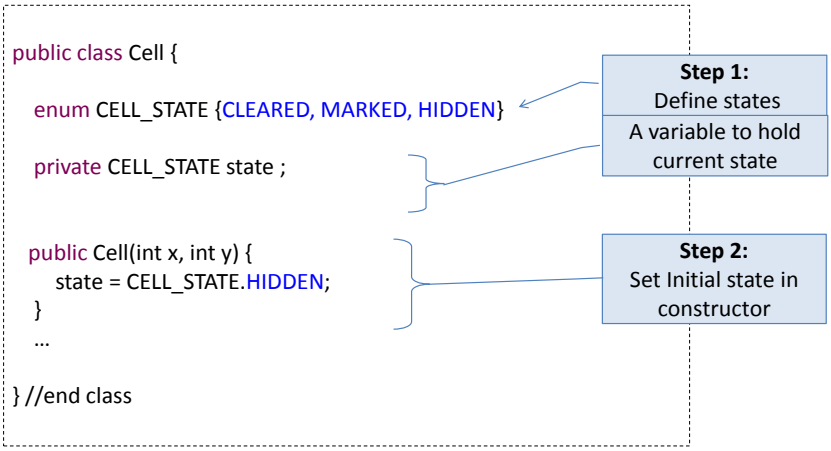


The steps to follow in this approach are:

Step 1: Define states

- Enumerate the states by using an enum construct.
- Define a variable to store the current state (for ease of reference, call this variable the 'state variable').

The code below illustrates how it is done using Java.



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

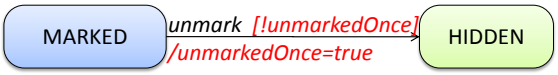
An event in the SMD corresponds to an operation in the class. By taking one event at a time, implement the corresponding operation to match the SMD. The logic of the operation is handled by a switch statement that is controlled by the state variable. The diagram below shows how this technique is applied 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 clear () {
    switch (state) {
      case HIDDEN:
        state = CELL_STATE.CLEARED;
        break;
      case MARKED:
        state = CELL_STATE.CLEARED;
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

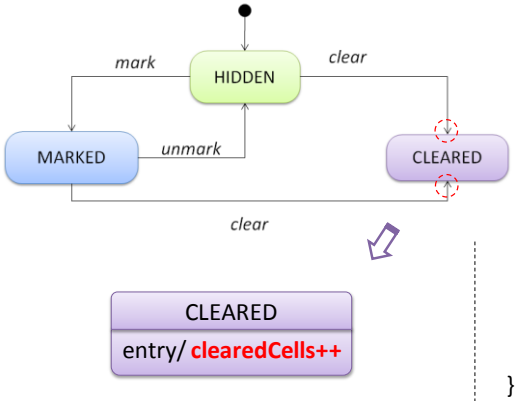
```
public void event (…) {
      switch (state) {
        case STATE_N:
          //behavior for event
           //       when in STATE_N
          break;
        …
      }
    }
```

Suppose the difficulty of the game is increased by not allowing more than one unmark per Cell. Given below is how the resulting guards and activities can be implemented.



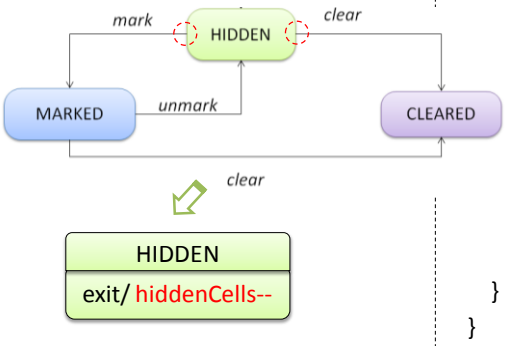

```
public void unmark () {
    switch (state) {
      case HIDDEN:
        break; //do nothing
      case MARKED:
        if (!unmarkedOnce) {   state = CELL_STATE.HIDDEN; unmarkedOnce=true;}
        break;
      case CLEARED:
        break; //do nothing
    }
  }
```

Now, assume the presence of the entry activity clearedCells++ in the CLEARED state. This entry activity is inserted in <u>every place</u> 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
    }
  }
```

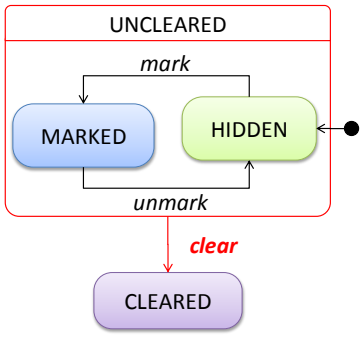
Similarly, exit activities are inserted where the state variable is changed from the state in concern (i.e. the one that has the exit activity) to another state. The example below shows how the HIDDEN state's exit activity is inserted into the clear() operation. Note that according to the SMD, there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



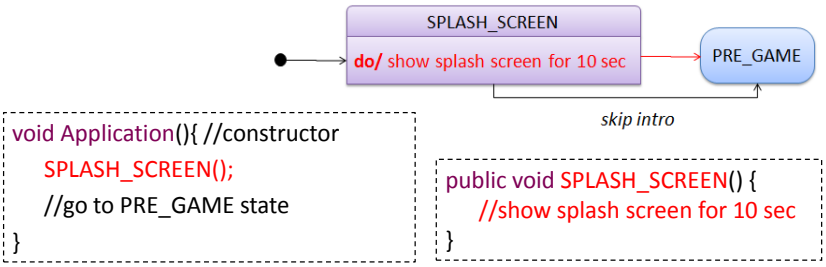

```
public void clear () {
    switch (state) {
      case HIDDEN:
        hiddenCells--;
        state=CELL_STATE.CLEARED;
        break;
      case MARKED:
        state=CELL_STATE.CLEARED;
        break;
      case CLEARED:  break; //do nothing
    }
  }
```

When superstates are involved, all the sub-states are simply grouped together in the switch statement. Note that the switch statement does not have a separate case for the superstate. Neither are superstates declared 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
    }
  }
```

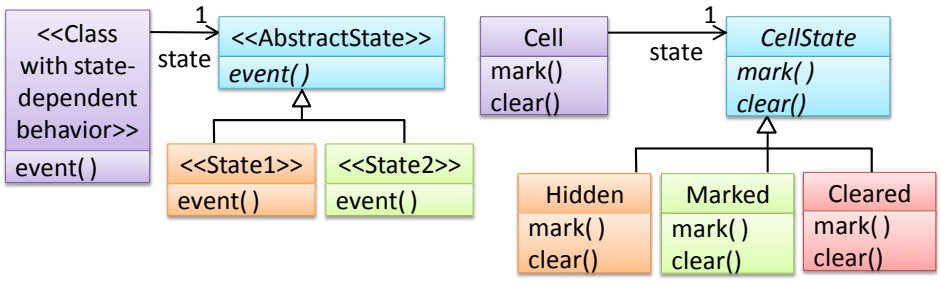


Activity states can be implemented as operations. To enter an activity state, simply call the operation that implements the activity state.



Using the State pattern

In this approach, inheritance and polymorphism are used to implement an SMD. Details about this pattern can be found from many other resources where state pattern (a GoF pattern) is documented. The diagram given below serves as a rough idea only.



Using state tables

In this approach, state transition data is specified 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, the object can be made to 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 that the object's behavior can be modified by simply altering the state table, even during runtime.

[Example 1]

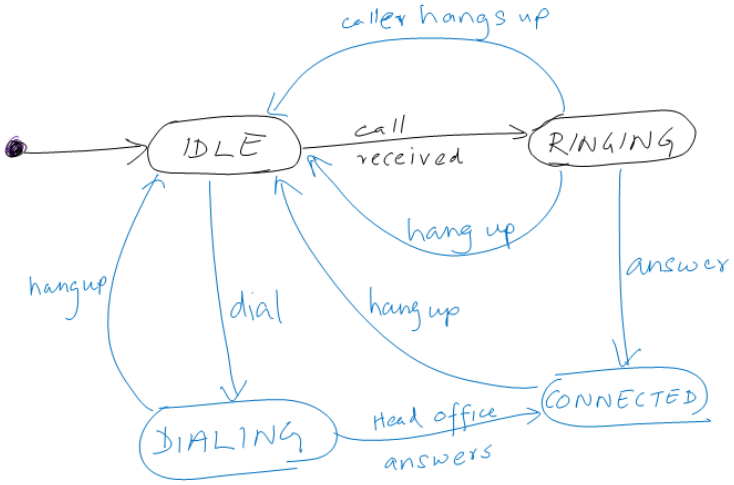
A software is designed for an emergency phone to be installed in security posts around a high security zone. It has a mouth piece, a speaker, and the following three buttons:

- *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 the starting point. It should capture information such as how the telephone will respond when the answer button in pressed while the phone is ringing.

[Answer]



Note that it is assumed that 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.

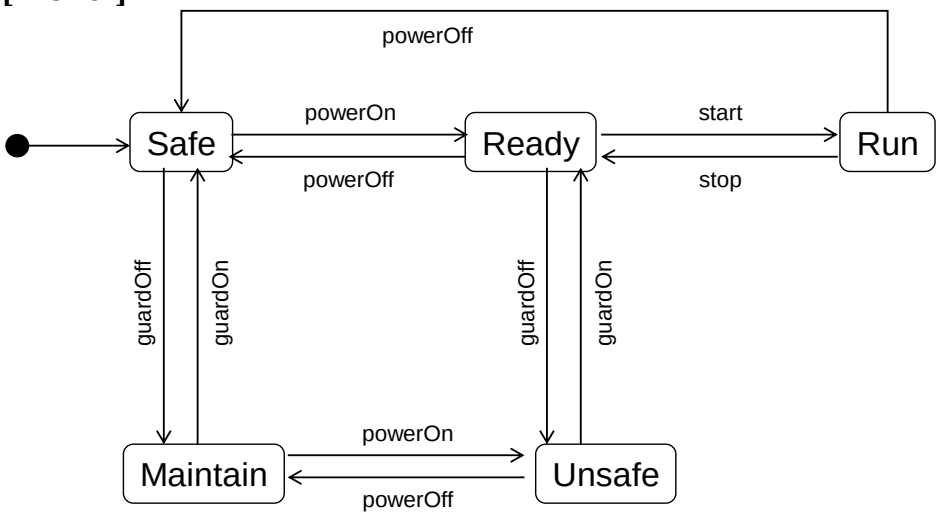
[Example 2]

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.

[Answer]



--- End of handout ---