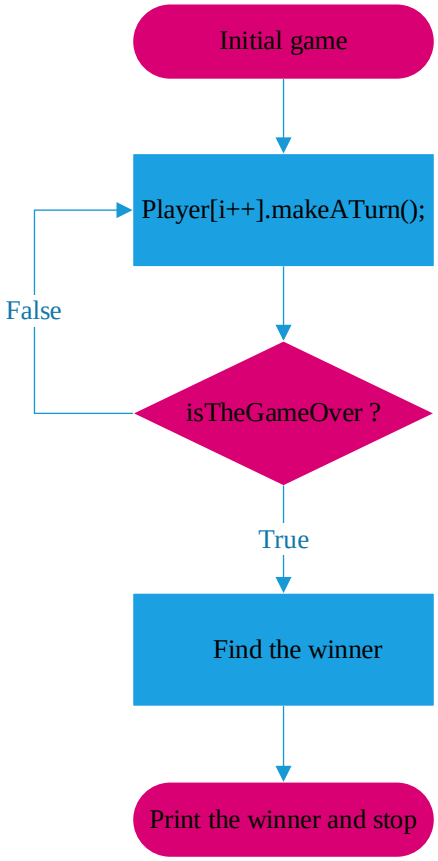


Control flow of the game

Class Player{

// The players will have an identification of (1, 2, 3, 4)

Int Id

// the amount of money that the player has

Int Money

// the number of rugs the player has remaining

Int Rugs

// Ture if the player is in the game, false if not

Boolean inthegame

//constructor method

//get method, get value of a certain field

//set method, set value of a certain field

//override toString method, helpful when using sout to debug

/* the player calling this method will make his turn

* in a turn, the player will first call **rollTheDice()** to roll the dice, then call **moveAssam()** to move

* Assam, then call **makeAPayment()** if Assam stands on a tile belongs to other player

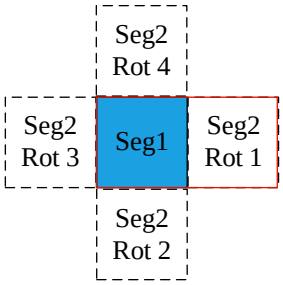
*/ @return **true** if the turn is over, **false** if not

Boolean makeATurn(){}

/* Each player needs to roll the dice to determine how many steps "Assam" should take. In

```
    * the "rollthedice()" method, we define the probability for each possible outcome of the dice roll.
    * For example: the probability of step1 is 1/6;
    * the probability of step2 is 2/6;
    * the probability of step3 is 2/6;
    * the probability of step4 is 1/6;
    */ @return random integer from1-6
    Int rollTheDice(){}

    /* The "makeapayment()" method will calculate and facilitate the exchange of money between players.
    *  assuming player who calls this method is the one who rolled the dice,
    *  "Assam" needs to move "n" steps based on the number rolled. If "Assam" lands on a carpet
    *  belonging to another player " currentTileOwner," current player needs to pay money to
    *  "currentTileOwner" No payment is required if "Assam" lands on an empty space or their own carpet.
    *  @param currentTileOwner the player who owns the tile that
    */ @return true if the payment is successful, false if not
    Boolean makeAPayment(Player currentTileOwner){}

    /* checks if the number of money that a player has is zero and if it is zero they are taken out of the
    *  game.
    */ @param player the player who will be kicked out of the game
    Void kickOut(Player player){}
}

//the cartesian coordinates which stores a pair of int (x, y)
Class Coordinates{
    //integer from 0 to 6
    Int x
    //integer from 0 to 6
    Int y
    //constructor method
    //get method, get value of a certain field
    //set method, set value of a certain field
    //override toString method, helpful when using sout to debug
}

// The piece of Assam that will be moving on the board.
Class Assam{
    // This tells us which tile the Assam is currently occupying. For example (3,3) is the initial coordinates
    Coordinates Position
    // This tells the direction that Assam is currently facing which can be top, right, left or bottom.
    // Top, right, left, bottom = 1,2,3,4
    Int Direction

    //constructor method
```

```
    //get method, get value of a certain field
    //set method, set value of a certain field
    //override toString method, helpful when using sout to debug

    /* This is a method used to determine whether "Assam" needs to rotate. The direction of
    *  Assam's movement is determined by the current player. When "Assam" reaches the edge of the
    *  market, we need to implement a method that allows "Assam" to turn based on the direction
    *  indicated by the " arrow ". The current method is designed to determine whether Assam's
    *  current position requires rotation. If Assam needs to rotate, it signifies that the rotation is valid.
    *  If no rotation is necessary, it signifies that the rotation is invariant.
    *  @ param direction: Top, right, left, bottom, keep same (only one)
    */ @return true if the rotation is valid, false if not
    Boolean isRotationAssamValid(int direction){}

    /*  check whether Assam will be moved out of the broad, always call this method before moveAssam()
    *  @param size the size of the dice
    *  @param direction the facing direction of Assam
    *  @param position the cartesian position of Assam, for example, the initial position of Assam is (3,3)
    */ @return true if Assam moves valid, false if not
    Boolean isMoveAssamValid(int size, int direction, Coordinates position){}

    /*   move Assam in a certain direction with certain steps
    *  @param size the size of the dice, also the steps Assam will move
    */ @return true if Assam is successfully moved, false if not
    Boolean moveAssam(int size, int direction){}
}

// The board consists of tiles and each tile on the board will have a number of a matrix of 7x7.It has
variables that tell us whether the tile is covered with a rug and the type of player rug it is covered with.
Class Tile{
    // the position of this tile on the board
    Coordinates Position
    // true if Assam stands on this tile
    Boolean isOccupiedByAssam
    // true if this tile is covered by a carpet
    Boolean isOccupiedByCarpet
    // if this tile is occupied by a carpet, then assign the id of the player who placed the carpet on this tile
    Int ownerId

    //constructor method
    //get method
    //set method
    //override toString method
}
```



Carpet with four rotations

/* There are carpets of four different suits that belong to four players respectively. We need
* to define which player each carpet belongs to. Additionally, we need to define the location of
* each carpet using Tiles.
*/ For example: Tile [0][1] represents the position of the rug.

Class Carpet{

//sepecify the owner of this carpet

int belongsToPlayer

//the cartesian position of segment1

Coordinates seg1Posistion

//the cartesian position of segment2

Coordinates seg2Posistion

//currentRotation, range from 1,2,3,4

Int Rotation

//constructor method

//get method, get value of a certain field

//set method, set value of a certain field

//override toString method, helpful when using sout to debug

/* the segment1 of the carpet will be placed by input player to the input position with a chosen
* direction
* @param **player** who will place the carpet
* @param **position** the position of segment1 that the carpet will be placed
* @param **direction** the direction of the carpet which will be used to place segment2
*/ @return **true** if the placement is successful

Boolean placeACarpet(Player player, Coordinates position, int direction){}

}

//the grid is the whole game board and this grid is made of 7 x 7 tiles

Class Grid{

/* "TheGameRuns" is a field within the class "Grid." We use this field to determine whether
* to continue the game. We utilize "Grid.isthegameover()" to check if the game has ended. If the
* game is over, we break the loop and use "Grid.Foundwinner()" to calculate the scores for each
*/player. If the game is not over, we proceed to the next round of the game using "makeaturn()."

Boolean theGameRuns

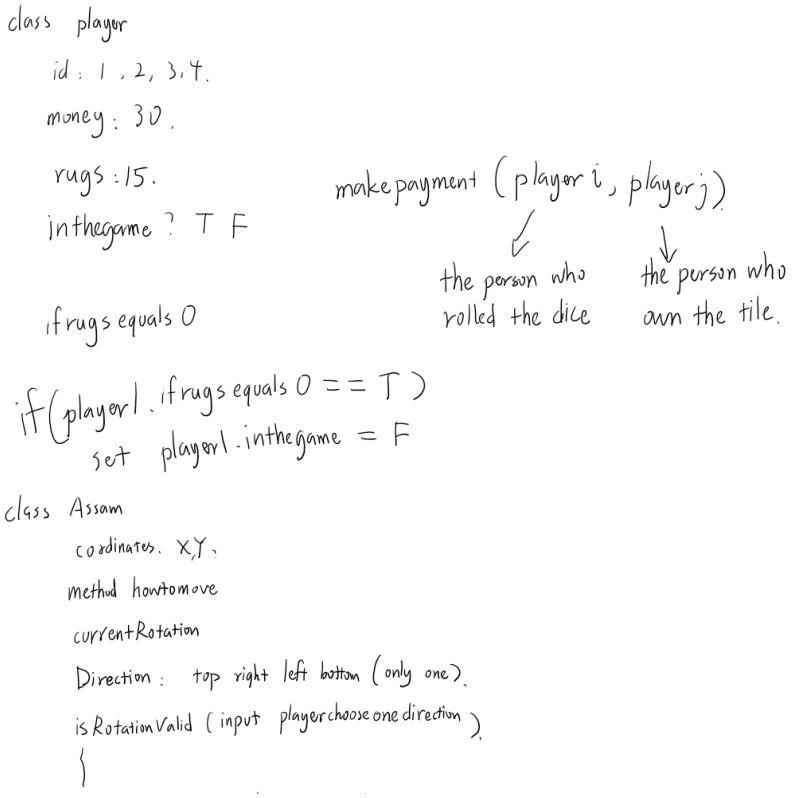
//constructor method
//get method, get value of a certain field
//set method, set value of a certain field
//override toString method, helpful when using sout to debug

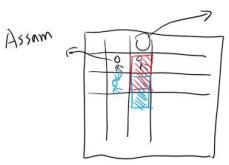
/* create user interface, game board, create players, place Assam at the center
* @param **numberOfPlayers** the number of players in the game minimum 2, maximum 4
*/ @return **true** if the game is created successfully, **false** if not
Boolean initGame(int numberOfPlayers){}

/* check whether there is two or more players in the game
*/ @return **true** if there are only one player in the game, **false** if not
Boolean isTheGameOver(){}

/* Find the winner of the game by calculating the money and number of carpets
*/ @return a **player** who wins the game
Player findWinner(){}
}

The following are discussion sketches:

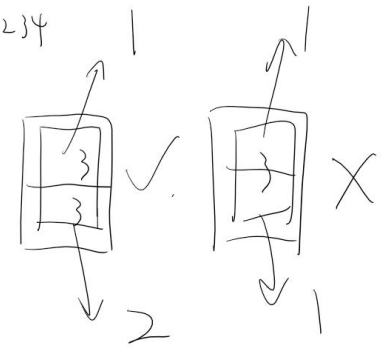




class Tile.

isoccupied by player : 1234

isAssamHere ? :



class Dice.

side 1 }
side 2 } 4
2 4

method rollTheDice (return random integer from 1-6).

if 1. steps 1

2 2

3 2

4 3

5 3

6 4

$\frac{1}{6}$ steps 1

$\frac{2}{6}$ steps 2

$\frac{2}{6}$ steps 3

$\frac{1}{6}$ steps 4

(0,0) (0,1)

class carpet (rug). {
belongs to player : 1234
position of seg 1 : X Y
position of seg 2 : x Y
}

carpet : Tile[0][0]
Tile[0][1]

class Grid.
Find winner ( )
isthegameover ( )
initgame ( )

main{

Tiles[7][7]

Grid.initgame( ) ← put Assam at the center

CorX = 4
CorY = 4. Tile[4][4]. isAssam = T.

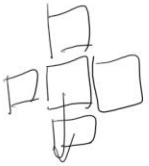
```
while (thegameruns). {
    makeaturn()
Grid.isthegameover()
  }.
Grid.Findwinner()
  makeaturn() {
    currentplayer : 1
    rollthedice()
    move Assam()
    makepayment()
    place a carpet()
    currentplayer ++; (1,2,3,4,1,2,3,4,1,2,3,...)
  }.
```

if game is over
break

if winner is found. ← calculate the money
the # of carpets.

```
}
Grid. moveAssam( currentplayer, side of Dice, position of Assam).
    return T if ismoveAssamValid
  else return F

  ismoveAssamValid ( )
    whether Assam is outside the grid.

  place a carpet (currentplayer, position of Assam).
    if (currentplayer. rug ==0)
      return false.

  else
    currentplayer. rug --;
    for( (x+1, y)  (x, y+1)
         (x-1, y)  (x, y-1) ) {
      // TODO
      if(Tile[x+1][y ] && Tile[  ][ ]
          == player1          == player1 )
        return false.
```

(x, y).



(x+1, y)

(x+2, y)
(x, y)
(x+1, y+1)
(x+1, y-1)