# High Level Comments

There are many different ways to answer the following questions. Just be sure that student has answer somewhere in their analysis document

1. Determine if square at (r,c) is unused: This could be done by declaring squares[r][c] = null; it could be done by having an isUnused:boolean attribute in Square; it could be done by subclass

2. Determine if a request to extend color is valid: You need a method to determine if a move is valid without actually making that move and returning true (or false).

3. Create a Model class. These are mandated by EBC and are just so useful for long-term change and growth. The Model keeps track of all configurations as well as which one is the current configuration (could be stored as an integer index into configurations; could also just be an object reference).

4. Determine if puzzle has been won. This capability needs computation, so it is a function in Puzzle I expect, and there is benefit to storing the result as “victory” to avoid recomputing.

5. Reset puzzle. If you add this method to PlanarPuzzle, then no points off, but it has more responsibility than it should (it could be done, by removing color from all squares other than base squares) but it just makes more sense to put this method in Model.

6. No getXXX or setXXX methods. These are always assumed by the existing of attributes.

7. Define Direction Class. Without this, you will constantly be fumbling with deltaR/deltaC arguments.

# Total Points = 52 + 5 + 34 + 14 + 5 = 110

This will be pro-rated out of 116 to result in a score that is out of 100.

Key methods that must be identified:

* hasWon() – can be with either Model or Puzzle and computes whether Puzzle has been won
* canExtend() – arguments could be absolute (i.e., row, col) but it could also be relative as with a direction or MoveType
* neighbors() – could be in Square class, but this has limits because Square doesn’t know about the borders and can’t know if a square is “too much to the right”
* reset() – could be with either Model or Puzzle and knows how to reset puzzle
* extendColor() – same comment as canExtend

# Entities (52 + 5)

+2 **Model (10)**  
 --------  
+3 configurations: PlanarPuzzle[\*] +1 for att; +1 for type; +1 for multiplicity  
+1 currentPuzzle : PlanarPuzzle +1 for att; (could be an integer to select from set)  
+2 victory : Boolean +1 for att; +1 for type  
 ----------------------------  
+2 resetPuzzle() : void +2 ability to reset puzzle (either in Model or PlanarPuzzle)

+2 **PlanarPuzzle (28)**  
 -------------  
+3 squares : Square [\*] +1 for att: + 2 for type and multiplicity  
+2 numRows : int +1 for att and +1 for int type  
+2 numColumns : int +1 for att and +1 for int type  
+3 selected : Square[0..1] +1 for att and +1 for type and multiplicity  
 maxRow: int = 10  
 maxColumn : int = 10  
 maxColors : int = 5  
 ---------------------  
+2 PlanarPuzzle(info) +1 for constructor; +1 for incoming info  
+4 neighbors (sq:Square) : Square[\*] +1 for method; +1 for arg; + 2 for response/multiplicity  
+4 isValidExtend(row:int, col:int, c:Color) : bool +1 for method; +1 for coordinate position; +1 for color  
 +1 for result  
+4 extendColor(row:int, col:int, c:Color) : bool +1 for method; +1 for coordinate position; +1 for color  
 +1 for result  
+2 hasWon() : Boolean +1 for method; +1 for result

+2 **Square (14)**  
 -------  
+2 row : int +1 for att; +1 for type  
+2 column : int +1 for att; +1 for type  
+2 label : int +1 for att; +1 for type  
+2 color : Color[0..1] +1 for att; +1 for optional [0..1] type  
+2 isUnused : Boolean +1 for att.; +1 for type  
 ----------------------  
+2 Square(row:int, column:int) +1 for constructor; +1 for position information

-2 The ability to get neighbors could be in the Square class, but that would ultimately be challenging because   
some neighbors it computes would be “off the board.” Take off two points if this is done.

+5 Student has a way to deal with “direction”. This could be done either:  
 • A Direction class, with a deltaR and deltaC  
 • For every method that needs to refer to a neighboring square

# Controllers (34 points)

+2 **ExtendColorController (9)**  
 ------------------------  
+2 model: Model +1 for att; +1 for type   
+2 app: PlanarPuzzleApp +1 for att; +1 for type  
 -----------------------  
+3 extendColor(dir: Direction): boolean +1 for method; +1 for arg; +1 for return

+2 **SelectController (9)**  
 ----------------   
+2 model : Model +1 for att; +1 for type   
+2 app : PlanarPuzzleApp +1 for att; +1 for type  
 -------------------------  
+3 selectSquare(e:MouseEvent): boolean +1 for method; +1 for arg; +1 for return

+2 **ResetController (7)**  
 ------------------  
+2 model:Model +1 for att; +1 for type  
+2 app:PlanarPuzzleApp +1 for att; +1 for type  
 -------------------------  
+1 resetPuzzle(): void +1 for method

+2 **SelectConfigurationController (9)** --------------------------------  
+2 model : Model +1 for att; +1 for type  
+2 app: PlanarPuzzleApp +1 for att; +1 for type  
 -------------------------  
+3 chooseConfiguration(n:int):boolean +1 for method; +1 for arg; +1 for return

# Boundary (14 + 5)

+5 Need to explain how user will request to fill a square with color. Some choices:  
 • Standard set of four buttons (left, up, right, and down).  
 • Key handler (arrow keys) based on selected square  
 • Collection of buttons with colors that are allowed for selected square

+2 **PlanarPuzzleApp**  
 ------------------  
+2 drawingCanvas : Canvas +1 for att; +1 for type  
+2 resetButton : Button +1 for att; +1 for type  
+2 chooseConfigurations : Select +1 for att; +1 for type  
+2 Congratulations : Label +1 for att; +1 for type  
+2 model : Model +1 for att; +1 for type  
 ---------------------  
+2 PlanarPuzzleApp(model:Model) +1 for constructor; +1 for argument

This will be rendered via HTML using different HTML elements and CSS for placement.