## Free-response Questions

1. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

   Consider using the `BoundedGrid` class from the GridWorld case study to model a game board.

   **DropGame** is a two-player game that is played on a rectangular board. The players — designated as BLACK and WHITE — alternate, taking turns dropping a colored piece in a column. A dropped piece will fall down the chosen column until it comes to rest in the empty location with the largest row index. If the location for the **newly dropped** piece has **at least four** neighbors that match its color, the player that dropped this piece wins the game.

   The diagram below shows a sample game board on which several moves have been made.

**North**

West

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ● |   |   |   |   |   |
| 1 | ● |   |   |   |   | ○ |
| 2 | ● | ○ | ▨ | ○ |   | ● |
| 3 | ○ | ○ | ○ | ● |   | ● |

East

**South**

The following chart shows where a piece dropped in each column would land on this board.

| Column | Location for Piece Dropped in the Column |
|---|---|
| 0 | No piece can be placed, since the column is full |
| 1 | (1, 1) |
| 2 | (2, 2) |
| 3 | (1, 3) |
| 4 | (3, 4) |
| 5 | (0, 5) |

Note that a WHITE piece dropped in column 2 would land in the shaded cell at location (2, 2) and result in a win for WHITE because the four neighboring locations — (2, 1), (3, 1), (3, 2), and (2, 3) — contain WHITE pieces. This move is the only available winning move on the above game board.

The `Piece` class is defined as follows.

```
public class Piece
{
  /** @return the color of this Piece
   */
  public Color getColor()
  {  /* implementation not shown */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

An incomplete definition of the `DropGame` class is shown below. The class contains a private instance variable `theGrid` to refer to the `Grid` that represents the game board. Players will add `Piece` objects to this grid as they take turns. You will implement two methods for the `DropGame` class.

```
public class DropGame
{
  private Grid<Piece> theGrid;


  /** @param column  a column index in the grid
   *         Precondition: 0 ≤ column < theGrid.getNumCols()
   *  @return null  if no empty locations in column;
   *          otherwise, the empty location with the largest row index within column
   */
  public Location dropLocationForColumn(int column)
  {  /* to be implemented in part (a) */  }


  /** @param column  a column index in the grid
   *         Precondition: 0 ≤ column < theGrid.getNumCols()
   *  @param pieceColor  the color of the piece to be dropped
   *  @return true  if dropping a piece of the given color into the specified column matches color
   *                with at least four neighbors;
   *          false  otherwise
   */
  public boolean dropMatchesNeighbors(int column, Color pieceColor)
  {  /* to be implemented in part (b) */  }


  // There may be instance variables, constructors, and methods that are not shown.
}
```

(a) Write the DropGame method dropLocationForColumn, which returns the resulting Location for a piece dropped into the specified column. If there are no empty locations in the column, the method should return null. Otherwise, of the empty locations in the column, the location with the largest row index should be returned.

Complete method dropLocationForColumn below.

```
/** @param column  a column index in the grid
 *         Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @return null  if no empty locations in column;
 *           otherwise, the empty location with the largest row index within column
 */
public Location dropLocationForColumn(int column)
```

(b) Write the DropGame method dropMatchesNeighbors, which returns true if dropping a piece of a given color into a specific column will match the color of at least four of its neighbors. The location to be checked for matches with its neighbors is the location identified by method dropLocationForColumn. If there are no empty locations in the column, dropMatchesNeighbors returns false.

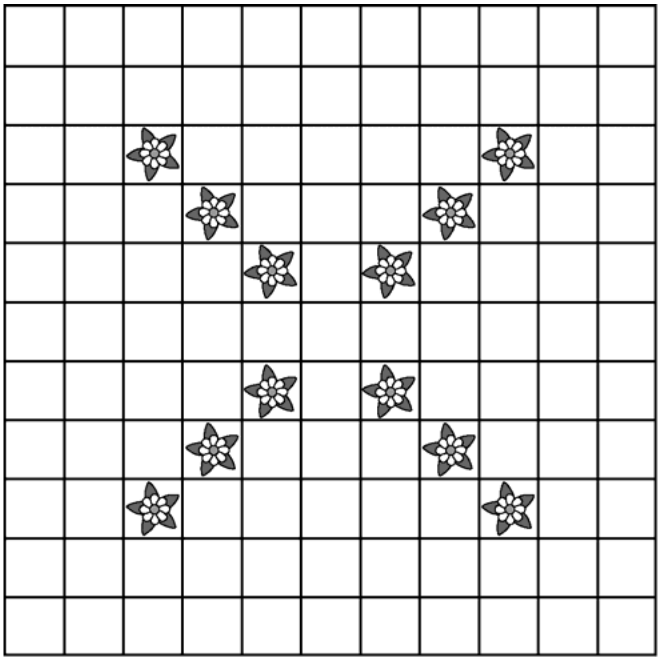
In writing dropMatchesNeighbors, you may assume that dropLocationForColumn works as specified regardless of what you wrote in part (a).

Complete method dropMatchesNeighbors below.

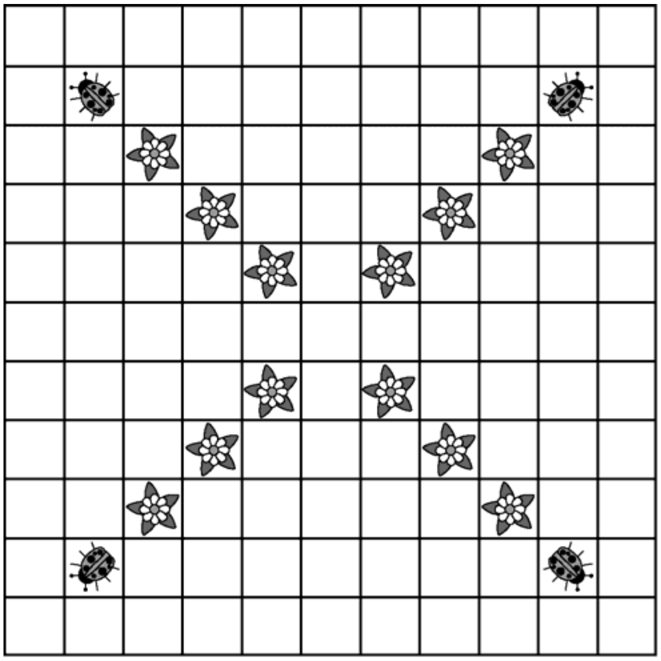
```
/** @param column  a column index in the grid
 *         Precondition: 0 ≤ column < theGrid.getNumCols()
 *  @param pieceColor  the color of the piece to be dropped
 *  @return true  if dropping a piece of the given color into the specified column matches color
 *                 with at least four neighbors;
 *          false  otherwise
 */
public boolean dropMatchesNeighbors(int column, Color pieceColor)
```

2. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

   In this question, you will consider two approaches for implementing the design of a bug that produces an X-shaped pattern of flowers. You may assume that there are no other actors in the grid and that there is enough room for the X to be placed in the grid with a row of empty locations surrounding the area filled by the X. Here is a pattern in which each arm of the X has length 3. Note that the center of the X is not marked with a flower.



(a) In the first approach, the bug releases four helper bugs that each drop the appropriate number of flowers along one arm of the X.



This approach is implemented by a class `XBug1`. The declaration of the `XBug1` class is as follows. The `act` method puts four instances of a class `LineBug` (which you will need to implement) into the grid and then removes itself.

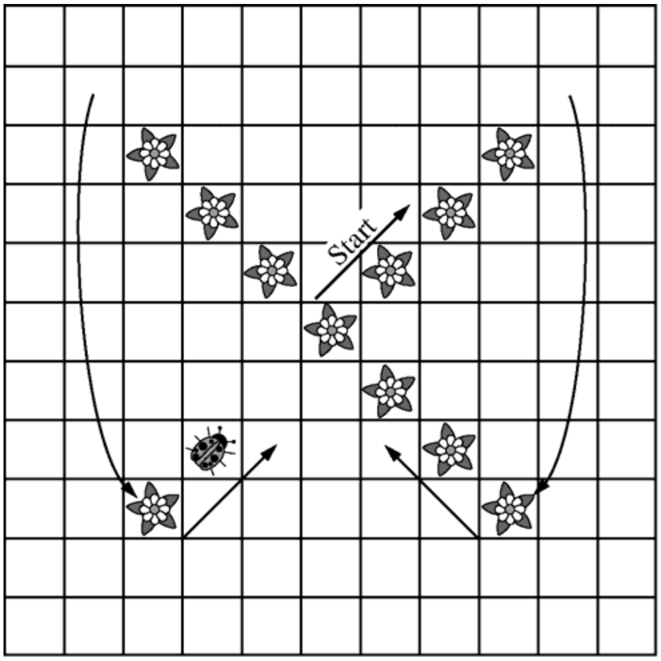
```
public class XBug1 extends Bug
{
  private int length; // the length of each of the arms of the X

  public XBug1(int aLength)
  {  length = aLength;  }

  public void act()
  {
    Grid<Actor> gr = getGrid();
    Location loc = getLocation();
    int dir = Location.NORTHEAST;
    for (int k = 0; k < 4; k++)
    {
      LineBug lbug = new LineBug(length);
      lbug.setDirection(dir);
      lbug.putSelfInGrid(gr, loc.getAdjacentLocation(dir));
      dir += Location.RIGHT;
    }
    removeSelfFromGrid();
  }
}
```

Write the declaration for a class `LineBug` with the following features:

- A `LineBug` is constructed with an integer parameter, denoting the number of flowers that the bug drops during its lifetime.
- When the `act` method is called, if the appropriate number of flowers has already been dropped, the `LineBug` removes itself from the grid; otherwise, the `LineBug` moves once, thereby dropping a flower.

Write the complete `LineBug` class, including all instance variables, a constructor, and any required methods.

(b) In the second approach, the bug drops flowers along the path in successive calls to `act`. When the bug has reached the end of an arm, it jumps to the end of another arm, as shown below.



The declaration of the `XBug2` class is as follows.

```
public class XBug2 extends Bug
{
  private int length;           // the length of each of the arms of the X
  private int steps;            // the number of times the act method has been called
  private Location bottomLeft;  // the location of the bottom left end of the X
  private Location bottomRight; // the location of the bottom right end of the X

  public XBug2(int aLength)
  {
    length = aLength;
    steps = 0;
  }

  public void putSelfInGrid(Grid<Actor> gr, Location loc)
  {
    /* puts the bug in the grid and initializes the bottomLeft and bottomRight locations */
  }

  public void act()
  {   /* to be implemented in part (b) */   }
}
```

Write the `XBug2 act` method. You may assume that the instance variables have been initialized prior to the first call of the `act` method.

In each call to the `act` method, the `XBug2` makes one call to `move`. It starts in the center point of the X and moves northeast. When it reaches the top right end of the X, it calls `moveTo` to move to the bottom right end of the X. It then moves northwest. When it reaches the top left end of the X, it calls `moveTo` to move to the bottom left end of the X. When the X pattern is completed, the `XBug2` removes itself from the grid in the next call to the `act` method.

Complete method `act` below.

```
  public void act()
```