Appendix B — Testable API

`info.gridworld.grid.Location` class (`implements Comparable`)

`public Location(int r, int c)`
constructs a location with given row and column coordinates

`public int getRow()`
returns the row of this location

`public int getCol()`
returns the column of this location

`public Location getAdjacentLocation(int direction)`
returns the adjacent location in the direction that is closest to `direction`

`public int getDirectionToward(Location target)`
returns the closest compass direction from this location toward `target`

`public boolean equals(Object other)`
returns `true` if `other` is a `Location` with the same row and column as this location; `false` otherwise

`public int hashCode()`
returns a hash code for this location

`public int compareTo(Object other)`
returns a negative integer if this location is less than `other`, zero if the two locations are equal, or a positive integer if this location is greater than `other`. Locations are ordered in row-major order.
Precondition: `other` is a `Location` object.

`public String toString()`
returns a string with the row and column of this location, in the format (row, col)

Compass directions:

```
public static final int NORTH = 0;
public static final int EAST = 90;
public static final int SOUTH = 180;
public static final int WEST = 270;
public static final int NORTHEAST = 45;
public static final int SOUTHEAST = 135;
public static final int SOUTHWEST = 225;
public static final int NORTHWEST = 315;
```

Turn angles:

```
public static final int LEFT = -90;
public static final int RIGHT = 90;
public static final int HALF_LEFT = -45;
public static final int HALF_RIGHT = 45;
public static final int FULL_CIRCLE = 360;
public static final int HALF_CIRCLE = 180;
public static final int AHEAD = 0;
```

`info.gridworld.grid.Grid<E>` interface

`int getNumRows()`
returns the number of rows, or -1 if this grid is unbounded

`int getNumCols()`
returns the number of columns, or -1 if this grid is unbounded

`boolean isValid(Location loc)`
returns `true` if `loc` is valid in this grid, `false` otherwise
Precondition: `loc` is not `null`

`E put(Location loc, E obj)`
puts `obj` at location `loc` in this grid and returns the object previously at that location (or `null` if the location was previously unoccupied).
Precondition: (1) `loc` is valid in this grid (2) `obj` is not `null`

`E remove(Location loc)`
removes the object at location `loc` from this grid and returns the object that was removed (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`E get(Location loc)`
returns the object at location `loc` (or `null` if the location is unoccupied)
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedLocations()`
returns an array list of all occupied locations in this grid

`ArrayList<Location> getValidAdjacentLocations(Location loc)`
returns an array list of the valid locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getEmptyAdjacentLocations(Location loc)`
returns an array list of the valid empty locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<Location> getOccupiedAdjacentLocations(Location loc)`
returns an array list of the valid occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`ArrayList<E> getNeighbors(Location loc)`
returns an array list of the objects in the occupied locations adjacent to `loc` in this grid
Precondition: `loc` is valid in this grid

`info.gridworld.actor.Actor` class

`public Actor()`
constructs a blue actor that is facing north

`public Color getColor()`
returns the color of this actor

`public void setColor(Color newColor)`
sets the color of this actor to `newColor`

`public int getDirection()`
returns the direction of this actor, an angle between 0 and 359 degrees

`public void setDirection(int newDirection)`
sets the direction of this actor to the angle between 0 and 359 degrees that is equivalent to `newDirection`

`public Grid<Actor> getGrid()`
returns the grid of this actor, or `null` if this actor is not contained in a grid

`public Location getLocation()`
returns the location of this actor, or `null` if this actor is not contained in a grid

`public void putSelfInGrid(Grid<Actor> gr, Location loc)`
puts this actor into location `loc` of grid `gr`. If there is another actor at `loc`, it is removed.
Precondition: (1) This actor is not contained in a grid (2) `loc` is valid in `gr`

`public void removeSelfFromGrid()`
removes this actor from its grid.
Precondition: this actor is contained in a grid

`public void moveTo(Location newLocation)`
moves this actor to `newLocation`. If there is another actor at `newLocation`, it is removed.
Precondition: (1) This actor is contained in a grid (2) `newLocation` is valid in the grid of this actor

`public void act()`
reverses the direction of this actor. Override this method in subclasses of `Actor` to define types of actors with different behavior

`public String toString()`
returns a string with the location, direction, and color of this actor

`info.gridworld.actor.Rock` class (`extends Actor`)

`public Rock()`
constructs a black rock

`public Rock(Color rockColor)`
constructs a rock with color `rockColor`

`public void act()`
overrides the `act` method in the `Actor` class to do nothing

`info.gridworld.actor.Flower` class (`extends Actor`)

`public Flower()`
constructs a pink flower

`public Flower(Color initialColor)`
constructs a flower with color `initialColor`

`public void act()`
causes the color of this flower to darken

Appendix C — Testable Code for APCS A/AB

Bug.java

```
package info.gridworld.actor;

import info.gridworld.grid.Grid;
import info.gridworld.grid.Location;

import java.awt.Color;

/**
 * A Bug is an actor that can move and turn. It drops flowers as it moves.
 * The implementation of this class is testable on the AP CS A and AB Exams.
 */
public class Bug extends Actor
{
  /**
   * Constructs a red bug.
   */
  public Bug()
  {
    setColor(Color.RED);
  }


  /**
   * Constructs a bug of a given color.
   * @param bugColor the color for this bug
   */
  public Bug(Color bugColor)
  {
    setColor(bugColor);
  }


  /**
   * Moves if it can move, turns otherwise.
   */
  public void act()
  {
    if (canMove())
      move();
    else
      turn();
  }


  /**
   * Turns the bug 45 degrees to the right without changing its location.
   */
  public void turn()
  {
    setDirection(getDirection() + Location.HALF_RIGHT);
  }
```

```
    /**
     * Moves the bug forward, putting a flower into the location it previously occupied.
     */
    public void move()
    {
      Grid<Actor> gr = getGrid();
      if (gr == null)
        return;
      Location loc = getLocation();
      Location next = loc.getAdjacentLocation(getDirection());
      if (gr.isValid(next))
        moveTo(next);
      else
        removeSelfFromGrid();
      Flower flower = new Flower(getColor());
      flower.putSelfInGrid(gr, loc);
    }


    /**
     * Tests whether this bug can move forward into a location that is empty or contains a flower.
     * @return true if this bug can move.
     */
    public boolean canMove()
    {
      Grid<Actor> gr = getGrid();
      if (gr == null)
        return false;
      Location loc = getLocation();
      Location next = loc.getAdjacentLocation(getDirection());
      if (!gr.isValid(next))
        return false;
      Actor neighbor = gr.get(next);
      return (neighbor == null) || (neighbor instanceof Flower);
      // ok to move into empty location or onto flower
      // not ok to move onto any other actor
    }
}
```