Mock Exam 2

ETH Zurich

December 2, 2015

Name: ______________________________

Group: ______________________________

Question 1	/ 11
Question 2	/ 16
Question 3	/ 14
Total	/ 41

1 Contracts (11 points)

We are interested in a software system simulating a cellular automaton. The universe is represented by a finite square grid composed of square cells (there is at least 1). Each cell can be in two states: alive or dead. Every cell, depending on its position in the grid, can have from a minimum of 3 neighbors (a cell in a corner) to a maximum of 8 neighbors (a cell in the middle).

The evolution of the automaton from one generation to the next is fully determined by the following set of rules:

- Any living cell with less than 2 living neighbors dies in the next generation.
- Any living cell with 2 or 3 living neighbors lives in the next generation.
- Any living cell with more than 3 living neighbors dies in the next generation.
- Any dead cell with exactly 3 living neighbors becomes alive in the next generation.
- Any dead cell with a number of living neighbors different from 3 stays dead in the next generation.

The evolution from one generation into the next happens by applying the above rules simultaneously to every cell in the grid (see Figures 1 and 2).

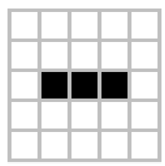

Figure 1: Sample first generation. A black square is a living cell.

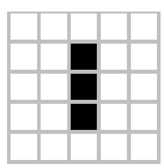

Figure 2: Second generation, computed from the first according to the given set of rules.

Your task is to add appropriate contracts (preconditions, postconditions and class invariants) to the excerpt of class *CELL_GRID* below, so that the informal specification above and the feature comments are reflected in each class interface.
Please note that the number of dotted lines does not indicate the number of missing contracts. It might also be useful to have a look at the excerpt of class *ARRAY_2* shown below.

1.1 Solution

```
class
    CELL_GRID
create
    make

feature {NONE} -- Initialization
```

```
    make (a_dimension: INTEGER)
            -- Initialize grid's dimension to 'a_dimension' and its cells to dead.
        require
            dim_positive: a_dimension >= 1
        do
            -- Implementation omitted.
        ensure
            dim_set: dim = a_dimension
            current_grid_initialized_to_default : current_grid. all_default
        end

feature -- Access

    dim: INTEGER
            -- Grid dimension.

    cell_at (i, j: INTEGER): BOOLEAN
            -- Value of cell at (i,j).
        require
            i_within_bounds: i >= 1 and i <= dim
            j_within_bounds: j >= 1 and j <= dim
        do
            -- Implementation omitted.
        ensure
            right_cell : Result = current_grid.item (i,j)
        end

feature -- Status Setting

    set_cell_status (b: BOOLEAN; i, j: INTEGER)
            -- Set status of cell at (i,j).
        require
            i_within_bounds: i >= 1 and i <= dim
            j_within_bounds: j >= 1 and j <= dim
        do
            -- Implementation omitted.
        ensure
            cell_status_set : cell_at (i, j) = b
        end

feature -- Basic operations

    compute_next_generation
            -- Compute next_grid, copy it to current_grid and re-initialize next_grid.
        do
            -- Implementation omitted
        end

feature {NONE} -- Implementation

    current_grid: ARRAY2 [BOOLEAN]
```

```
            -- Grid representation as a matrix of boolean cells ("True" means alive for a
               cell).

    new_state_of_cell (i, j, living_neighbors: INTEGER): BOOLEAN
            -- Apply Conway's Game of Life rules to compute new state for cell at (i,j)
               given a number of 'living_neighbors'.
        require
            i_within_bounds: i >= 1 and i <= dim
            j_within_bounds: j >= 1 and j <= dim
            living_neighbors_within_bounds: living_neighbors >= 0 and living_neighbors <=
               8
        do
            -- Implementation omitted.
        ensure
            death_rule_1: current_grid.item (i, j) and (living_neighbors < 2 or
               living_neighbors > 3) implies not Result
            life_rule: current_grid.item (i, j) and (living_neighbors = 2 or
               living_neighbors = 3) implies Result
            birth_rule: not current_grid.item (i, j) and (living_neighbors = 3) implies
               Result
            death_rule_2: not current_grid.item (i, j) and (living_neighbors /= 3) implies
               not Result
        end

invariant
    current_grid_exists: current_grid /= Void
    grid_dimension_positive: dim > 0
    current_grid_dimension_is_dim: current_grid.width = dim and current_grid.height = dim
end
```

2 Data Structures (16 points)

In this task you are going to implement several operations for a generic class *SET* [*G*].

A set is a collection of distinct objects. Every element of a set must be unique; no two members may be identical. All set operations preserve this property. The order in which the elements of a set are listed is irrelevant (unlike for a sequence or tuple). Therefore the two sets $\{5, 10, 12\}$ and $\{10, 12, 5\}$ are identical.

There are several fundamental operations for constructing new sets from given sets.

- Union: The union of A and B, denoted by $A \cup B$, is the set of all elements that are members of either A or B.
- Intersection: The intersection of A and B, denoted by $A \cap B$, is the set of all elements that are members of both A and B.
- Relative complement of B in A (also called the set-theoretic difference of A and B), denoted by $A \backslash B$ (or $A - B$), is the set of all elements that are members of A but not members of B.

The Jaccard index (or coefficient) measures similarity between sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets (see Figure 3). If both sets are empty the Jaccard coefficient is defined as 1.0.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Figure 3: Jaccard index definition for non-empty sets A and B.

Your task is to fill in the gaps of class *SET* [*G*] below. Please note:

- Your code should satisfy the contracts and provide new contracts where necessary.
- The set should never contain **Void** elements.
- The number of dotted lines does not indicate the number of missing contract clauses or code instructions.
- The implementation of class *SET* [*G*] is based on an arrayed list. The arrayed list is set up to use object comparison, so features like *has* and *prune* use object equality instead of reference equality when comparing elements from the set. The following features of class *ARRAYED_LIST* may be useful:

```
class ARRAYED_LIST [G]

feature
  has (v: G): BOOLEAN
    -- Does current include 'v'?

  start
    -- Move cursor to first position if any.

  extend (v: G)
    -- Add 'v' to the end.
```

```
  prune (v: G)
      -- Remove first occurrence of 'v', if any, after cursor position.
      -- Move cursor to right neighbor.

-- Other features are omitted.
end
```

2.1 Solution

```
class
  SET [G]

create
  make_empty

feature {NONE} -- Initialization

  make_empty
      -- Create empty Current.
    do
      create content.make (0)
      content.compare_objects
    ensure
      empty_content: content.is_empty
    end

feature -- Access

  count: INTEGER
      -- Cardinality of the current set.
    do
      Result := content.count
    end

  is_empty: BOOLEAN
      -- Is current set empty?
    do
      Result := count = 0
    end

  has (v: G): BOOLEAN
      -- Does current set contain 'v'?
    require
      v /= Void
    do
      Result := content.has (v)
    end

  add (v: G)
      -- Add 'v' to the current set.
    require
```

```
      v /= Void
    do
      if not has (v) then
        content.extend (v)
      end
    ensure
      in_set_already: old has (v) implies (count = old count)
      added_to_set: not old has (v) implies (count = old count + 1)
    end

  remove (v: G)
      -- Remove 'v' from the current set.
    require
      v /= Void
    do
      if has (v) then
        content.start
        content.prune (v)
      end
    ensure
      removed_count_change: old has (v) implies (count = old count - 1)
      not_removed_no_count_change: not old has (v) implies (count = old count)
      item_deleted: not has (v)
    end

  duplicate: like Current
      -- Deep copy of Current.
    do
      create Result.make_empty
      across content as c
      loop
        Result.add (c.item)
      end
    ensure
      same_size: Result.count = count
      same_content: across content as c all Result.has (c.item) end
    end

feature -- Set operations.

  union (another: like Current): like Current
      -- Union product of the current set and 'another' set.
    require
      another /= Void
    do
      Result := another.duplicate
      across content as c
      loop
        Result.add (c.item)
      end
    ensure
      not_smaller: Result.count >= count and Result.count >= another.count
```

```
  end

 intersection (another: like Current): like Current
   -- Intersection product of the current set and 'another' set.
  require
   another /= Void
  do
   create Result.make_empty
   across content as c
   loop
    if another.has (c.item) then
     Result.add (c.item)
    end
   end
  ensure
   not_bigger: Result.count <= count and Result.count <= another.count
  end

 difference (another: like Current): like Current
   -- Set-theoretic difference of the current set and 'another' set.
  require
   another /= Void
  do
   create Result.make_empty
   across content as c
   loop
    if not another.has (c.item) then
     Result.add (c.item)
    end
   end
  ensure
   not_bigger_than: Result.count <= count
   not_smaller_than: Result.count >= count - another.count
  end

feature -- Set metrics.

 jaccard_index (another: like Current): REAL_64
   -- Jaccard similarity coefficient between current set and 'another' set.
  require
   another /= Void
  do
   if not (is_empty and another.is_empty) then
    Result := intersection (another).count / union (another).count
   else
    Result := 1.0
   end
  ensure
   bounds: Result >= 0.0 and Result <= 1.0
   empty_case: (is_empty and another.is_empty) implies Result = 1.0
  end
```

```
feature {NONE} -- Implementation

  content: ARRAYED_LIST[G]
      -- Items of the set.

invariant

  content_exists: content /= Void
  content_object_comparison: content.object_comparison
  non_negative_cardinality: count >= 0

end
```

3 Recursion (14 points)

The N-queens problem is the problem of positioning N queens on an $N \times N$ board such that no queen can attack another (i.e., share the same row, column, or diagonal). The N-queens problem can be solved recursively: having a solution for the first 4 rows of the board can be used to build a solution for the 5^{th} row, as is being done in Figure 4.

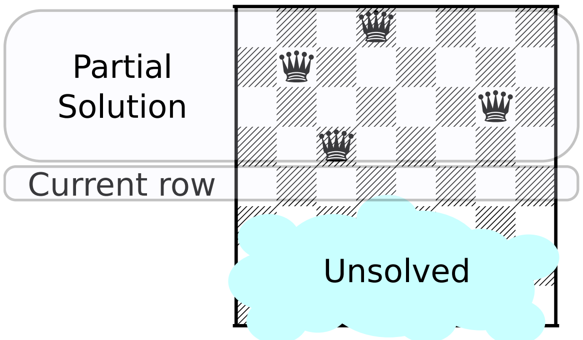

Figure 4: An example of a partial solution

A safe location is one which cannot be attacked by any of the currently placed queens.

A routine to solve the N-queens problem, *complete* (*partial*: *SOLUTION*), does as follows: if the partial solution is not yet complete, then for each safe location in the current row, add the safe location to the solution and use this new solution to solve the problem for the next row. The *current row* is *partial*.*row_count* + 1; for example in Figure 4 the partial solution has *row_count* equal to 4, thus the current row is 5. If the solution is already complete then it is added to the list of solutions.

You must complete the implementation of *PUZZLE* (which has an attribute *solutions* to store all solutions) below by filling in the body of *complete* and *attack_each_other*. Note that a solution can be added to the list of solutions using the *extend* feature from *LIST*.

3.1 Solution

```
note
  description: "N-queens puzzle."

class
  PUZZLE

feature -- Access

  size: INTEGER
      -- Size of the board.

  solutions: LIST [SOLUTION]
      -- All solutions found by the last call to 'solve'.

feature -- Basic operations

  solve (n: INTEGER)
      -- Solve the puzzle for 'n' queens.
```

```
    require
        solvable: n > 3 -- All puzzles with size > 3 are solvable
    do
        size := n
        create {LINKED_LIST [SOLUTION]} solutions.make
        complete (create {SOLUTION}.make_empty)
    ensure
        solutions_exists : not solutions.is_empty
        complete_solutions: across solutions as s all s.item.row_count = n end
    end

feature {NONE} -- Implementation

  complete (partial: SOLUTION)
        -- Find all complete solutions that extend the partial solution 'partial'
        -- and add them to 'solutions'.
    require
        partial_exists : partial /= Void
    local
        c: INTEGER
    do
        if partial.row_count = size then
            solutions.extend (partial)
        else
            from
                c := 1
            until
                c > size
            loop
                if not under_attack (partial, c) then
                    complete (partial.extended_with (c))
                end
                c := c + 1
            end
        end
    end

  under_attack (partial: SOLUTION; c: INTEGER): BOOLEAN
        -- Is column 'c' of the current row under attack
        -- by any queen already placed in partial solution 'partial'?
    require
        partial_exists : partial /= Void
    local
        current_row, row: INTEGER
    do
        current_row := partial.row_count + 1
        from
            row := 1
        until
            Result or row > partial.row_count
        loop
            Result := attack_each_other (row, partial.column_at (row), current_row, c)
```

```
            row := row + 1
        end
    end

    attack_each_other (row1, col1, row2, col2: INTEGER): BOOLEAN
            -- Do queens in positions (`row1', `col1') and (`row2', `col2') attack each other?
        do
            Result := row1 = row2 or
                col1 = col2 or
                (row1 - row2).abs = (col1 - col2).abs
        end

end
```