Project

For your project, do one of the following: write a paper relating to object-orientation; or develop a programming project in Ruby of your choice; or do the *elevator project* described below; or do the *graph project* described below. If you choose one of the first two, you must let me know by email within one week of this handout what you plan to do for your project, and I will respond with feedback. Alternatively, if you do the elevator project or the graph project, you don't need to tell me which of these you're doing in advance.

Paper

Your paper can be either an investigation or a survey. If the former, your paper should present an interesting problem or technique relating to object orientation. It should state the problem, provide background and motivation, and describe novel approaches (proposed by others or by you). Your paper might or might not require programming, depending on its focus. Alternatively, if you write a survey, it should be based on key papers related to the area you choose. Regardless of whether your paper is an investigation or a survey, it should include key references and should run about 12–16 double-spaced pages in length.

In addition to the resources cited in the bibliography section of our course syllabus, use NSU's online databases, particularly the ACM Digital Library, IEEE Computer Society Digital Library, and ScienceDirect.

Programming project

Develop an interesting program in Ruby. Possible projects:

- Significantly enhance one of the programming assignments from this course.
- Develop a game, such as a board, adventure, word, or computer game.
- Develop a domain-specific language (DSL) within Ruby, such as for manipulating files in a directory system, editing files, generating quizzes, describing graphical or geographical scenes, and adding and viewing entries in a calendar.
- Develop useful metaprogramming tools, similar to the attr_* class methods, or the methods defined in the Forwardable or Singleton modules.

Elevator project

Develop a simulation of an elevator that travels between floors 1 and *N* where *N* is an input. Time is sequenced in discrete steps starting at 1; at each time step, the elevator may ascend one floor, descend one floor, or remain at its current floor, as determined by its strategy. The first line of the input file indicates the number *N* of floors. This is followed by one line per person using the elevator: her id, call time (when she calls for the elevator), origin floor (where she boards the elevator), and destination floor (where she debarks the elevator). For example:

```
5
100   2   1   4
101   3   4   5
102   3   3   2
103   5   5   1
104   5   1   4
```

Here, individual 100 boards at time 2 from floor 1 with a destination of floor 4. Assume sensible inputs (e.g., times are positive integers, floors are integers in the range 1 through *N*).

Whenever the elevator stops at a floor from which people have called it (i.e., their origin floor but not before their call time), they all board the elevator. And whenever the elevator stops at any passengers' destination floor, they all get off the elevator. (Everyone behaves so as to minimize their travel time.) The simulation stops when every passenger has been brought to his or her destination floor (subject of course to their call time and origin floor constraints).

A *strategy* determines the floor the elevator begins on at time 1, and a policy for moving to the next floor (up one floor, down one floor, or remaining at the current floor) from one time step to the next. These are implemented by its *intial_floor* and *next_floor* policy methods. A strategy is notified whenever a person calls it (*person_calls method*), when a passenger boards (*person_gets_on*), and when a passenger debarks (*person_gets_off*). Information provided through these last three *notification methods* can be used to tailor an intelligent strategy for moving from floor to floor.

```
class Strategy
    def initial_floor
    end

    def next_floor
    end

    def person_calls(time, from)
    end

    def person_gets_on(time, from, to)
    end

    def person_gets_off(time, to)
    end
end
```

Here is a naïve strategy that does not make use of the information supplied by the notification methods:

> *Strategy 1:* Start at floor 1. In each time step, successively go up one floor until reaching the top floor *N,* then successively go down one floor until reaching the bottom floor 1, and continue this 'up to the top floor then down to the bottom floor' policy.

Here we run the simulation using the input file *data1.in* (given above) and *Strategy 1:*

```
>> sim = Simulation.new
=> #<Simulation:0x27d3fc8>
>> sim.run('data1.in', :strategy1)
Time 1: Floor 1
Time 2: Floor 2
        100 calls from floor 1
Time 3: Floor 3
        101 calls from floor 4
        102 calls from floor 3
        102 boards from floor 3
Time 4: Floor 4
        101 boards from floor 4
Time 5: Floor 5
        103 calls from floor 5
        104 calls from floor 1
        103 boards from floor 5
        101 debarks onto floor 5
Time 6: Floor 4
Time 7: Floor 3
Time 8: Floor 2
        102 debarks onto floor 2
Time 9: Floor 1
        100 boards from floor 1
        104 boards from floor 1
        103 debarks onto floor 1
Time 10: Floor 2
Time 11: Floor 3
Time 12: Floor 4
        100 debarks onto floor 4
        104 debarks onto floor 4
=> nil
```

Some definitions: A passenger's *wait time* is the number of time steps she must wait for the elevator to arrive from when she places the call. Her *travel time* is the number of time steps she is actually traveling in the elevator, from she boards until she gets off at her floor. Her *trip time* is the sum of her wait time and travel time.

We measure the efficiency of a strategy in two ways: by average wait time and by average trip time. In our example, the trip times for customers 100 through 104 are 10, 2, 5, 4, and 7, and their wait times are 7, 1, 0, 0, and 4, respectively. Then we define the *wait-time efficiency* (*trip-time efficiency*) of a strategy with respect to input to be the average wait time (average trip time) over all passengers. Our simulation reports these value for the simulation run most recently:

```
>> puts sim.average_trip_time
5.6
>> puts sim.average_wait_time
2.4
```

Here is what you need to do and what you should submit:

1. We will discuss design and go over some sequence diagrams for this project in class. You will not need to submit diagrams as part of this project.

2. Implement *Strategy 1* as part of your project. Where `sim` is a Simulation object, we run a simulation like this:

   ```
   sim.run(filename_string, strategy_symbol)
   ```

   The symbol `:strategy1` identifies this strategy.

3. Design and implement your own strategy, *Strategy 2,* that takes advantage of the information supplied by the notification methods. The symbol `:strategy2` identifies this strategy.

4. Write a method

   ```
   sim.multirun(nbr_runs, filename_prefix, strategy)
   ```

   that runs your program *nbr_runs* many times on the input files named by *filename_prefix* with suffixes *000.in, 001.in, …*, and returns the average efficiency. For example:

   ```
   sim.multirun(15, 'myelev', :strategy2)
   ```

   runs your *Strategy 2* on the files *myelev000.in, ..., myelev014.in* and returns the average of the 15 efficiency values. I will give you a set of input files to test your program on, and you should report the average efficiency you obtain on these files for both strategies. You can also use these input files to compare your strategy to other strategies, with respect to efficiency. I'll also give you a short Ruby program to generate your own test files.

5. Compare efficiency for Strategy 1 and Strategy 2 (and any other strategies you might choose to devise) with respect to average wait and trip time. Use a set of input files with these characteristics:
   number of floors: 8
   number of time steps: 120
   average arrival rate: 0.5 passengers per time step (i.e., 1 passenger every 2 time steps)
   percentage of trips that start or end at floor 1: 90%

   For example, you might use this to generate 10 test files:

   ```
   GenPeopleFiles.gen_input_files('myInput', 10, 8, 0.5, 120, 0.9)
   ```

6. Submit your Ruby code for this project in a zip archive by the due date.

Graph project

Part 1

A *graph* consists of nodes and edges. An edge is an unordered pair of two distinct nodes in the graph. Every pair of nodes are joined by at most a single edge (these are *simple* graphs). We create a new empty graph from the class `Graph`. We use the `add_node` method to add a single node and the `add_nodes` method to add multiple nodes. Nodes are identified by unique symbols. We call `add_edge` with two nodes to add an edge between a pair of nodes belonging to the graph. We can ask a graph for the number of nodes and of edges it contains, and for a list of its nodes and of its edges. The `to_s` method returns a string representing the graph's adjacency lists. Methods should not change the graph if called with invalid arguments; e.g., adding an edge that is already in the graph or that references a node that does not already belong to the graph. Your code does not have to generate the exact transcript that follows but it should provide this basic functionality.

```
>> g = Graph.new
=> <Graph: 0, 0>
>> g.add_node :a
=> a
>> g.add_nodes([:b, :c])
=> [b, c]
>> g
=> <Graph: 3, 0>
>> g.get_nodes
=> [a, b, c]
>> g.nbr_nodes
=> 3
>> g.add_edge(:a, :b)
=> [a, b]
>> g.add_edge(:b, :c)
=> [b, c]
>> g
=> <Graph: 3, 2>
>> g.get_edges
=> [[a, b], [b, c]]
>> g.nbr_edges
=> 2
>> puts g
a -> b
b -> a,c
c -> b
=> nil
```

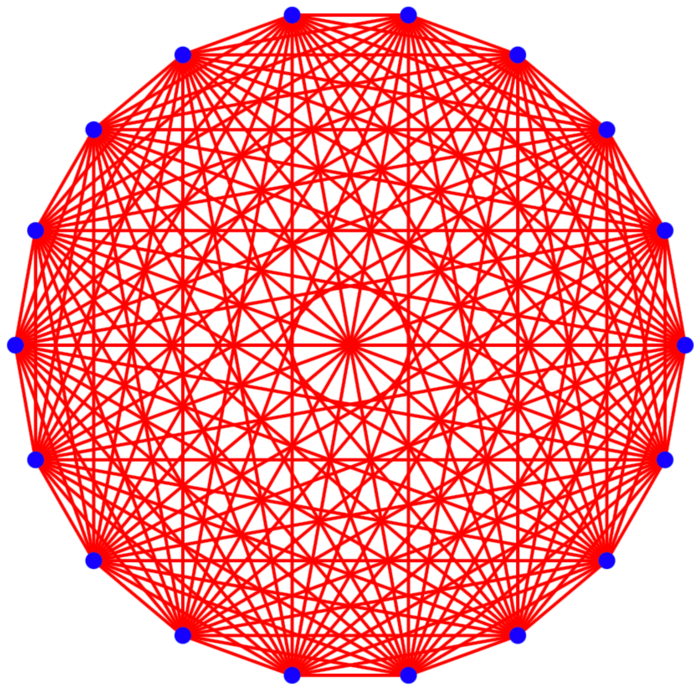
Part 2
A *complete graph* on *n* nodes is an *n*-node graph containing all possible edges. Such a graph has $\binom{n}{2} = \frac{n(n-1)}{2}$ edges. Write a function `genCompleteGraph(n, p)` that takes a positive integer *n* and, if argument *p* is not supplied, returns a complete graph on *n* nodes. The value of *p,* a floating-point between 0.0 and 1.0, is the probability that an edge joins any given pair of nodes. The default value of *p* is 1.0 indicating a complete graph with exactly $\binom{n}{2}$ edges. As the value of *p* decreases, the graph becomes increasingly sparse, and for *p* equal to zero the graph consists of *n* isolated nodes having no edges.

```
>> c3 = GraphUtils.genCompleteGraph(3)
=> <Graph: 3, 3>
>> c3.get_nodes
=> [v0, v1, v2]
>> c3.get_edges
=> [[v0, v1], [v0, v2], [v1, v2]]
>> c5 = GraphUtils.genCompleteGraph(5)
```

```
=> <Graph: 5, 10>
>> c5.get_nodes
=> [v0, v1, v2, v3, v4]
>> c5.get_edges
=> [[v0, v1], [v0, v2], [v0, v3], [v0, v4], [v1, v2], [v1, v3], [v1, v4],
[v2, v3], [v2, v4], [v3, v4]]
```

Part 3
Add the method `render(filename, center, radius)` to your *Graph* class. This method saves an image of this graph to *filename* in SVG format. The graph layout is circular, where the layout circle's *center* and *radius* are given by the second and third arguments. Look into using the *Victor* SVG image builder (https://github.com/DannyBen/victor), though you're welcome to use a different graphics package.

```
c18 = GraphUtils.genCompleteGraph 18
=> <Graph: 18, 153>
>> c18.render 'c18.svg', [400, 400], 200
=> true
```



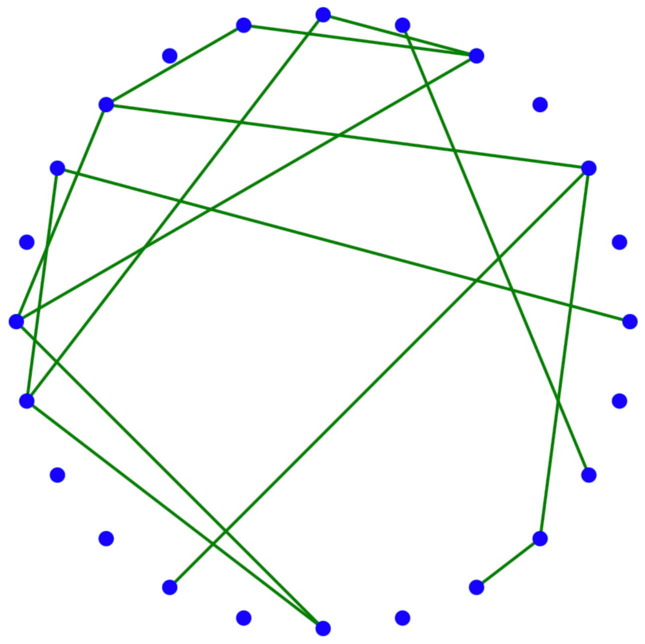
Part 4
Define the utility method `dfs(graph, node)`, a class method of *GraphUtils*, which performs depth-first search in *graph* starting from *node.* The method returns an array of the nodes reachable from *node,* that is, the set of nodes belonging to *node*'s connected component. Here is pseudo-code:

```
dfs(graph, n)
    visited = empty set
    dfs_rec(n, visited)
    return visited

dfs_rec(n, visited)
    visited.add(n)
    for each node v adjacent to n
        if v is not in visited
            dfs_rec(v, visited)
```

Part 5
Define the method `render_graphs(filename, graphs_strokes_fills)`, a class method of *GraphUtils,* for rendering and saving a set of graphs in SVG. The second argument is an array of triples *[g, scolor, fcolor]* where *g* is a graph, *scolor* is the (stroke) color of its edges, and *fcolor* is the (fill) color of its nodes. Note in the second transcript below, we use the utility method `genSubgraph(graph, nodes)` which constructs a subgraph of *graph* on a subset of its *nodes*. The subgraph's nodes inherits the properties – e.g., layout – of its parent graph.

```
>> c24 = GraphUtils.genCompleteGraph 24, 0.05
=> <Graph: 24, 15>
>> c24.layout_circular [400, 400], 200
=> [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15,
v16, v17, v18, v19, v20, v21, v22, v23]
>> GraphUtils.render_graphs 'c24.svg', [[c24, 'green', 'blue']]
=> true
```



```
>> ns = GraphUtils.dfs c24, :v0
=> [v0, v14, v11, v6, v12, v15, v17, v20, v18, v22, v3, v4, v8]
>> c24v0 = GraphUtils.genSubgraph c24, ns
=> <Graph: 13, 14>
>> GraphUtils.render_graphs 'c24v0.svg', [[c24, 'green', 'blue'], [c24v0,
'red', 'red']]
=> true
>>
```