Neighborly - Readme

1.0 Description

Neighoborly simulates two groups of residents living in a city. The simulation is made up of runs, where at each run residents move to new houses and then all the residents' happiness values are calculated and printed.

At each run, which residents are chosen to move, the houses that are available to them, and whether they ultimately do move is determined by a Simulator. The Simulator can, however, take into account the resident's happiness with a house, how far the resident can move, and other Resident attributes.

Each resident's happiness is determined by the group they belong to because all residents in the same group have the same happiness function. A resident's happiness depends on the number of disparate neighbors and total number of neighbors.

Say a group1 resident's happiness is found using the following function, where happiness decreases as the number of disparate neighbors increases.

Happiness = 100 - 100 * (number of disparate neighbors / total number of neighbors)

Also say the group1 resident is at at coordinate (1, 1) in the city below. It has 3 disparate neighbors out of 8 total neighbors, then its happiness will be 62.5.

	00	01	02
00	G2	G1	G2
01	G2	G1	G1
02	G1	G1	G1

If a group1 resident is surrounded by Group 2 residents, then its happiness will be zero.

2.0 Copyrights

Neighborly was completed in September 2023 by Aurea F. Maldonado.

The idea of Neighborly is based on the simulation "Parable of the Polygons" at https://ncase.me/polygons/.

The code which allows rendering of text and blocks on a window is by Amine Ben Hassouna and can be found at https://github.com/aminosbh/sdl2-ttf-sample/blob/master/src/main.c.

I cloned Hassouna's project, changed the main.c file to main.cpp, and then in the CMakeLists.txt file I changed *file(GLOB SOURCES "src/*.c")* to *file(GLOB Sources "src/*.cpp")* on line 39.

I am using the same copyright as Amine Ben Hassouna, see comments in main.cpp, lines 3 - 43.

Also I received a lot of help from the mentors at Udacity. So, thank you Udacity mentors.

3.0 Instructions

3.0. A. Clone From Github

In a folder, clone the git repository. This will make a Neighborly folder.

```
git clone https://github.com/flocela/Neighborly.git
```

3.0 B. Install Libraries and Make Project

In order to install the SDL2_ttf library, in the command line in the Neighborly folder type the following.

```
sudo apt-get install libsdl2-ttf-dev
```

The installer will ask if you have additional space and if you would like to continue. Type ‘Y’.

Then to make the code, create a build folder inside the Neighborly folder. In the Neighborly/build folder type:

```
cmake .. && make
```

3.0. C. Run the Project

To run the code, in the build folder type:

```
./sdl2-ttf-sample
```

Alternatively you could use an input file when running the program.

```
./sdl2-ttf-sample txt-inputs/txt-ex0simA.txt
```

To start with, I suggest just typing `./sdl2-ttf-sample` without the text file.

If you do not include the text file, then you will be asked if you want to use a pre-made-example simulation or go through a series of questions to create the simulation. I suggest on first try, to use one of the pre-made-example simulations (type ‘1’).

3.0 D. Run the Tests

The tests in the Neighborly/tests folder are not extensive, but to run them type the following in the build folder.

```
./RunTests
```

4.0. Inputs

4.0. A. Pre-made-Simulation Examples

Intro

A simulation is made with a SimulationComponents object. CollectorByExamples creates a SimulationComponents object to return back to main. The object contains the number of runs in the simulation, the colors for the residents, the vector of residents, the city, and the simulator.

Case 0 as an example

Use CollectorByExamples.cpp as an easy way to examine a set of simulation inputs. There are 8 pre-made-example simulations represented as cases in the switch statement. Take case 0 on line 46.

The random seed for this input is 0, from line 52.

```
components.randomSeed = 0;
```

The city is 30 houses wide by 30 houses long, from line 54.

```
components.city = make_unique<City_Grid>(30);
```

There are 2 groups, their colors are found in Colors.h, from line 57.

As can be seen in lines 62 - 77: There are two-hundred Group 1 residents and they have a happiness function that decreases as the diversity of the resident's neighbors increases. When the resident has no neighbors, their happiness is 70. When the resident has no disparate neighbors, their happiness is 95. When the resident has all disparate neighbors, their happiness is 50. At each run, the resident is allowed to move 15 units. The resident will be considered happy if they reach a happiness value of 80.

```
for (int ii=0; ii<200; ++ii)
{
  components.residents.push_back(make_unique<Resident_Customizable>(
    ii,     // id
    1,      // group number
    15,     // allowed movement
    80,     // happiness goal
    make_unique<HappinessFunc_Falling> (
      70, // happiness value with zero neighbors
      95, // happiness value at zero diversity
      50  // happiness value at one diversity
    )
   ));
 }
```

There are six-hundred Group 2 residents, and they have the same attributes as Group 1 residents. See lines 77 - 89.

As can be seen in lines 96 through 102: A Simulator_Basic_A is used to determine which residents move. At each run, 30% of the residents are chosen to possibly move. The residents

choose which house would make them happiest from 20 random houses that are in their allowed movement distance. If the chosen house would make them happier than their current house, then the resident is moved.

```
components.simulator = make_unique<Simulator_Basic_A>(
  components.city.get(),
  getSetOfPointers(components.residents),
  30, // percent of residents that are chosen to move each run
  20, // number of houses the residents can choose from
  make_unique<CityState_Simple>(components.city.get())
);
```

There will be 20 runs in this simulation as can be seen on line 104.

```
components.numOfRuns = 20;
```

4.0. B. Summary of Simulations in CollectorByExamples and input text files:

The eight cases in CollectorByExamples.cpp can be found as examples the Neighborly/txt-inputs folder. Below is a summary of those pre-made-example simulations. All the residents use a happiness function whose happiness decreases as diversity increases. For regular residents, the happiness when they have only like neighbors (zero diversity) is 95 and their happiness when they have only disparate neighbors (one diversity) is 50. The drop for finicky residents is more pronounced: with all like neighbors finicky neighbors have a happiness of 100, and with all disparate neighbors they have a happiness of zero.

Name	Case Number	Input Text File in txt-inputs folder	City Width and Height	Group 1: Count, Allowed movement, Happiness goal, Happiness Function	Group 2: Count, Allowed movement, Happiness goal, Happiness Function
Simulator A with a small city and regular residents	0	txt-ex0simA.txt	30 x 30	count: 200, allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50	count: 600, allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50
Simulator A with a small city and finicky residents	1	txt-ex1simA.txt	30 x 30	count: 200 allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0	count: 600 allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0
Simulator A with a large city and regular residents	2	txt-ex2simA.txt	120 x 120	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50

Simulator A with a large city and finicky residents	3	txt-ex3SimA.txt	120 x 120	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0
Simulator B with a small city and regular residents	4	txt-ex4SimB.txt	30 x 30	count: 200, allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50	count: 600, allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50
Simulator B with a small city and finicky residents	5	txt-ex5SimB.txt	30 x 30	count: 200 allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0	count: 600 allowed mvmt: 15, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0
Simulator B with a large city and regular residents	6	txt-ex6SimB.txt	120 x 120	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 95, one diversity: 50
Simulator B with a large city and finicky residents	7	txt-ex7SimB.txt	120 x 120	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0	count: 5760 allowed mvmt: 30, happiness goal: 80, Happ Func: Falling zero neighbors: 70, zero diversity: 100, one diversity: 0

*Case number refers to Case numbers in CollectorByExamples.cpp file.

4.0 C. Creating input files

Remember when creating input files that there are limits to what the program can run. See main.cpp lines 76 - 80 and lines 195 - 207. The maximum number of houses in either the x or y directions are 120. The maximum number of runs is 200. The number of houses needs to be larger than the number of residents. The city must have at least one house. There must be at least one resident.

Residents of type Resident_Customizable use a HappinessFunction to determine their happiness. For examples of how to make Residents with different happiness functions see the following text files in the txt-inputs folder.

Filename	Happiness Functions Used in Creating Resident_Customizable
ex-res-fall-and-flat.txt	HappinessFunc_Falling and HappinessFunc_Flat

ex-res-fall-rise.txt	HappinessFunc_Falling and HappinessFunc_Rising
ex-res-stepup-stepdown.txt	HappinessFunc_StepUp and HappinessFunc_StepDown

5.0 Outputs

The graphical outputs are a window which shows 1) the run number, 2) a map of the city with the residents denoted by color and happiness, 3) a chart showing the number of disparate neighbors per group per run, 4) a chart showing the average happiness per group per run.

The command line outputs show 1) a summary of the inputs, 2) for the first and last run - a map of the city with the residents denoted by their group number and happiness 3) for each run the average number of disparate neighbors per resident per group, 4) for each run the average happiness per group per run.

6.0 File and Class Structure

All .h and .cpp files are in the “src” folder. Underscores indicate a subclass. So, Axis_Basic is a subclass of Axis.

The tests are in the “tests” folder. Each file corresponds to a class. Not all classes are tested.

The pre-made-example simulation files are in the “txt-inputs” folder.

The following UML diagrams (6.0 A - 6.0F) do not show every connection to a class nor every method in a class, but show the most fundamental connections and methods.

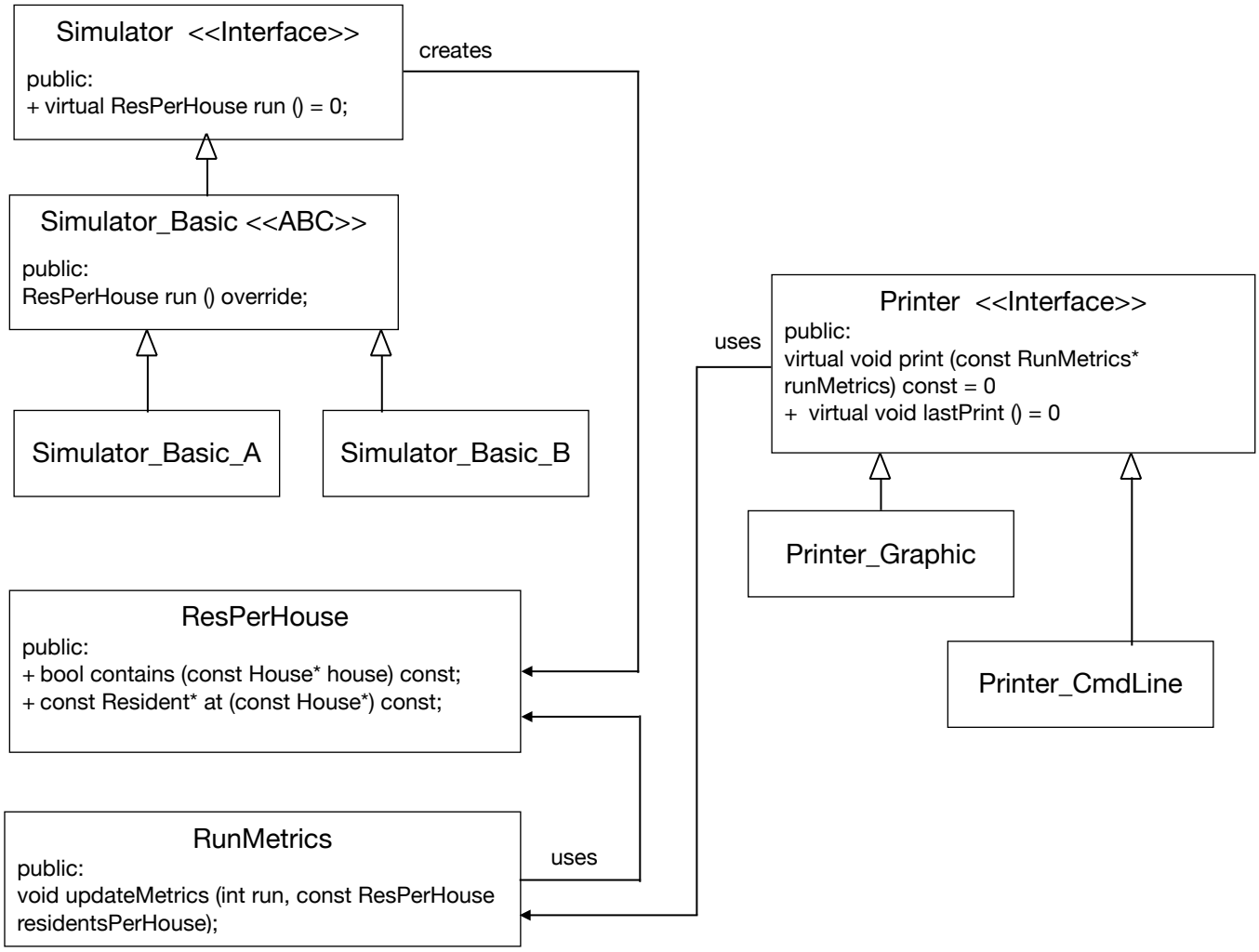
6.0 A. UML Diagram for Run Loop in main.cpp

The following is an abbreviation of the for-loop (in main.cpp) that runs and prints the simulation. Each loop is a run in the simulation. The simulator’s run() method returns the output of the run as a ResPerHouse object. That ResPerHouse object is used to update a RunMetrics object. The RunMetrics object is used to print the output. See lines 122 - 136.

```
for (int ii=0; ii<components.numOfRuns; ii++)
{
    ResPerHouse residentsPerHouse = components.simulator->run();
    runMetrics.updateMetrics(ii, move(residentsPerHouse));
    graphicPrinter.print(&runMetrics);
    cmdLinePrinter.print(&runMetrics);
}
```

Diagram on next page…

6.0 A. UML Diagram for Run Loop in main.cpp continued…



6.0 B. UML Diagram For Fundamental Classes

House contains an address as identification.

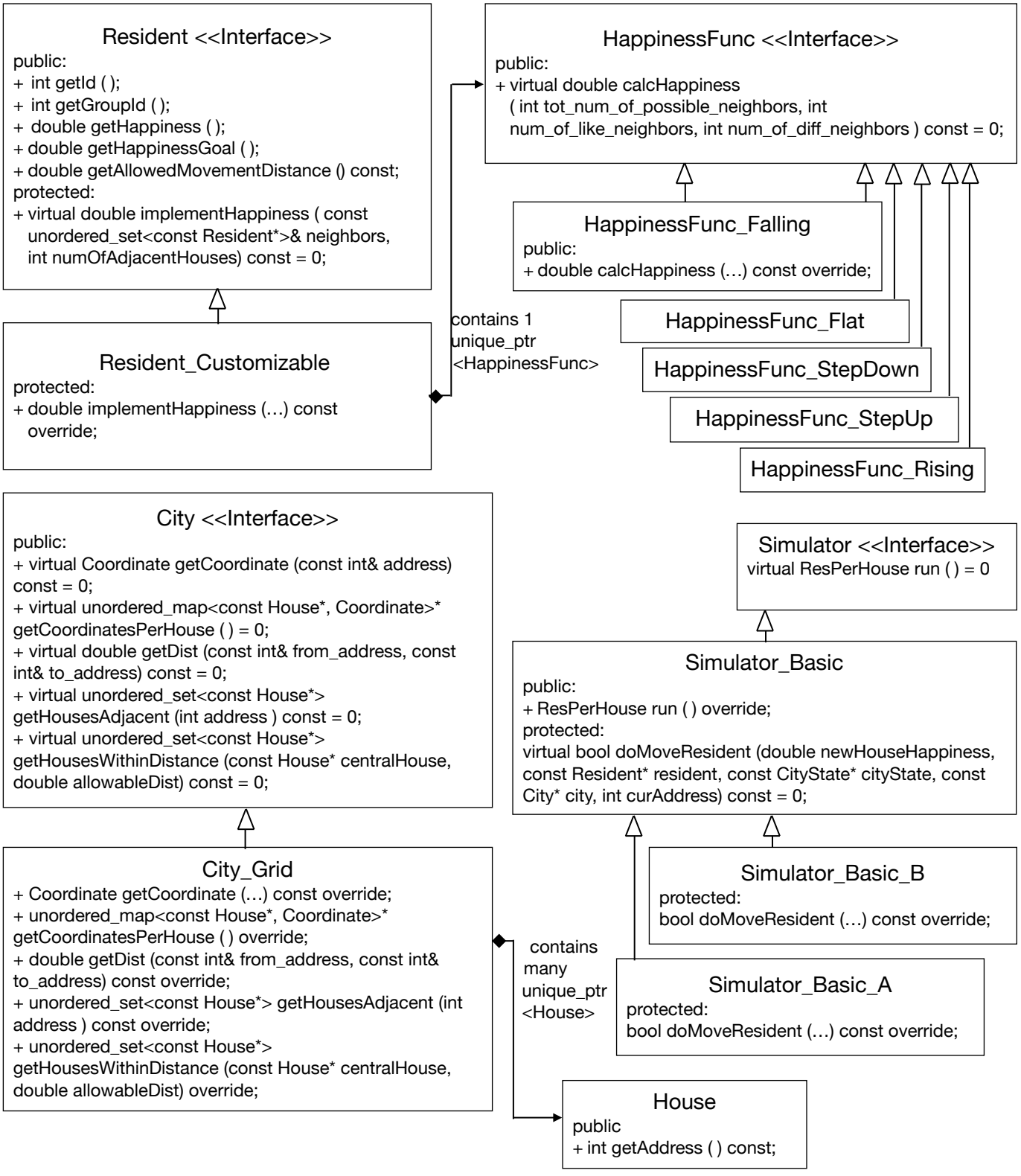
Resident has methods to get the Resident's happiness value, happiness goal, and the allowed distance a Resident can travel per run. A Simulator may use the allowed movement distance to limit how far a resident can move per run. Resident_Customizable, a subclass of Resident, takes a function to determine its happiness value.

A City contains many houses. It has methods to retrieve the houses and the houses' coordinates. City also determines which houses are adjacent to a certain house.

A Simulator completes each run, which means it moves residents to different houses and keeps track of where the residents live in the City. It also updates the residents' happiness values after a run before it returns the resulting ResPerHouse.

Diagram on next page…

6.0 B. UML Diagram For Fundamental Classes Continued….

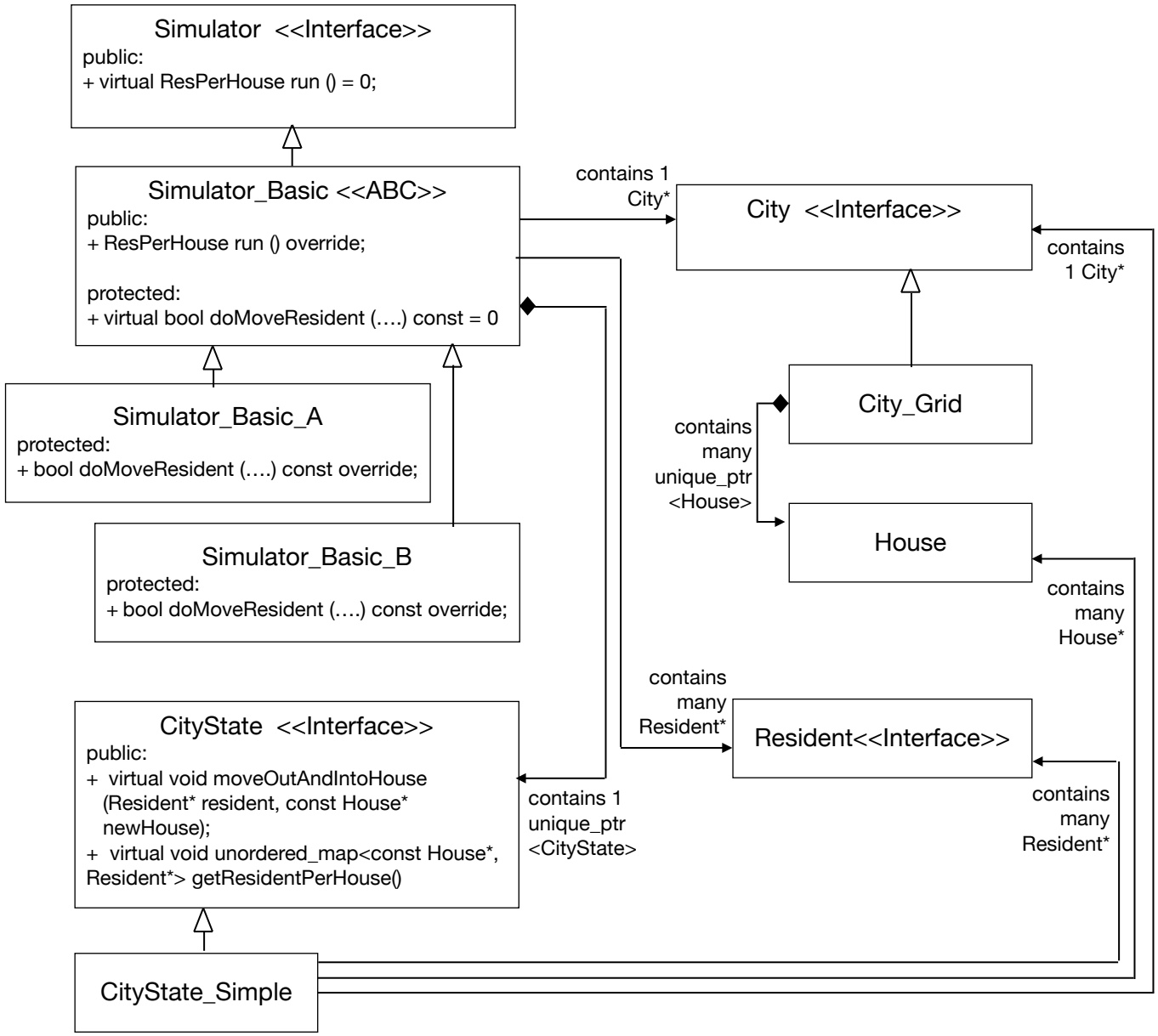


6.0 C. UML Diagram for Simulator_Basic

At each run, the Simulator_Basic object moves residents to new houses, updates all the residents' happiness values, keeps track of which house belongs which resident, and returns a ResPerHouse object.

At each run Simulator_Basic does the following: 1) chooses the residents to try to move (It does that using its vector<Resident*> _sorted_residents.) 2) For each of those moving-residents, asks CityState what houses are available (CityState keeps track of which house corresponds to each resident), 3) It chooses the new house that will make the resident happiest, and 4) tries to move the resident into that house. (Based Simulator_Basic's subclass the resident is moved or not moved. For Simulator_A, the new house must make the resident happier than the current house. For Simulator_B, the house must make the resident happy).

Once all the moving-residents have been moved, all the residents' happiness values are updated and the ResPerHouse object is constructed and returned. (ResPerHouse is basically an unordered_map of the Residents per House.)

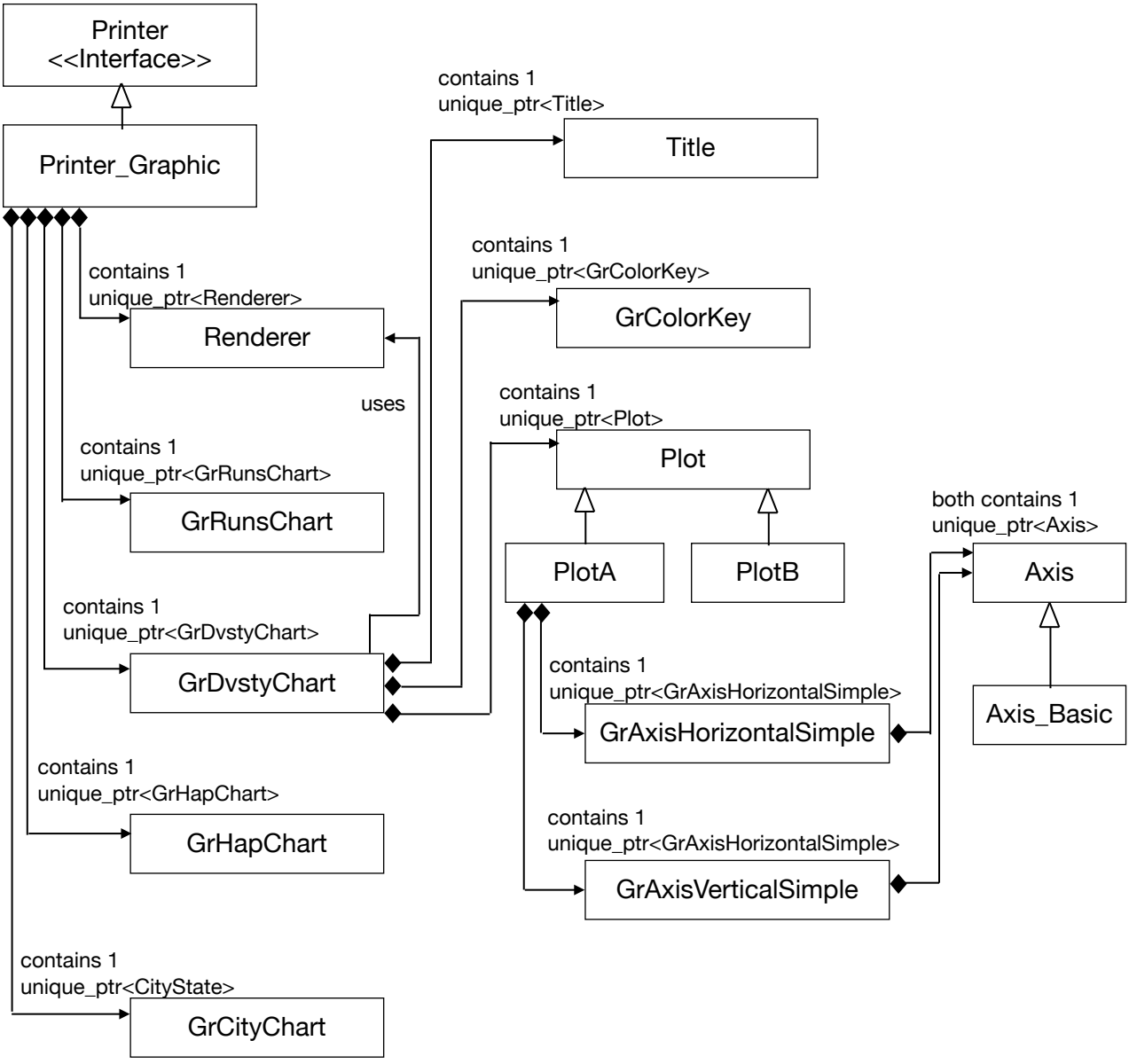


6.0 D. UML Diagram Printer_Graphic

At the end of each run the Printer_Graphic's print method is called. The Printer_Graphic's print method then calls the print methods contained in its GrRunChart, GrDvstyChart, GrHapChart, and GrCityChart.

- GrRunChart prints the current run number.
- GrDvstyChart prints the average diversity for each group per run.
- GrHapChart prints the average Happiness value for each group per run.
- GrCityChart prints the city map, where the residents are colored by their group number and whether they reached their happiness goal or not.

Only the exiting arrows of GrDvstyChart are shown, but the other charts have similar exiting arrows. See tests/test_Axis_Basic.cpp for a better understanding of the outputs of Axis_Basic.



6.0 E. CollectorByQuestions and CityMaker_CmdLine UML Diagram

main uses a CollectorByQuestions to create a SimulationComponents object, which it uses to create a simulation.

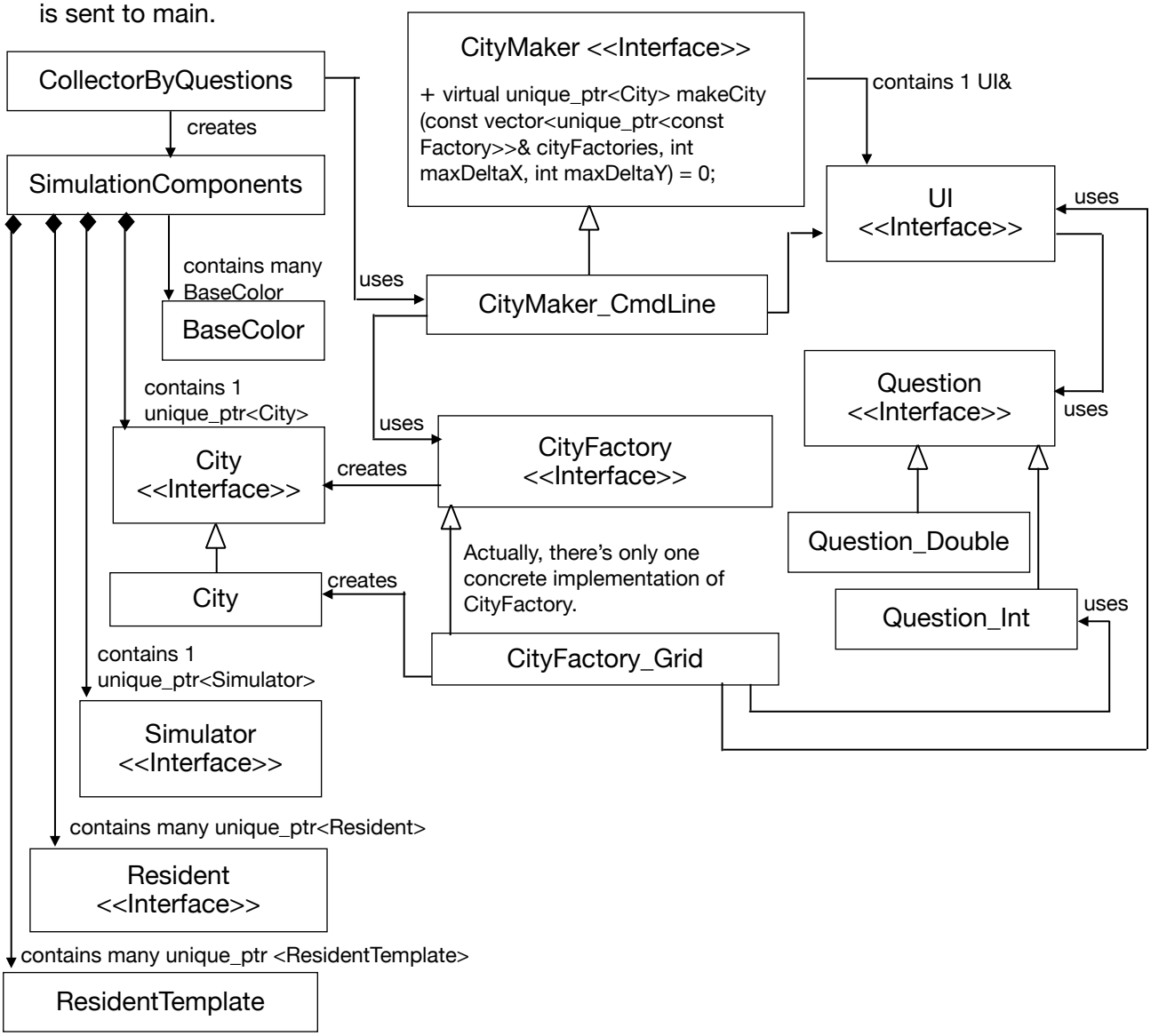
This diagram shows how CollectorByQuestions uses CityMaker_CmdLine to create a City that is placed inside the SimulationComponents object.

main.cpp passes CollectorByQuestions a vector of unique_ptrs of CityFactories. CollectorByQuestions creates a CityMaker_CmdLine and passes it the CityFactories.

CityMaker_CmdLine uses a UI to ask the user what specific type of City to create. It then calls on the corresponding concrete CityFactory to create the correct City.

CityMaker_CmdLine passes back a unique_ptr<City> to CollectorByQuestions.

CollectorByQuestions moves this unique_ptr<City> into the SimulationComponents object that is sent to main.



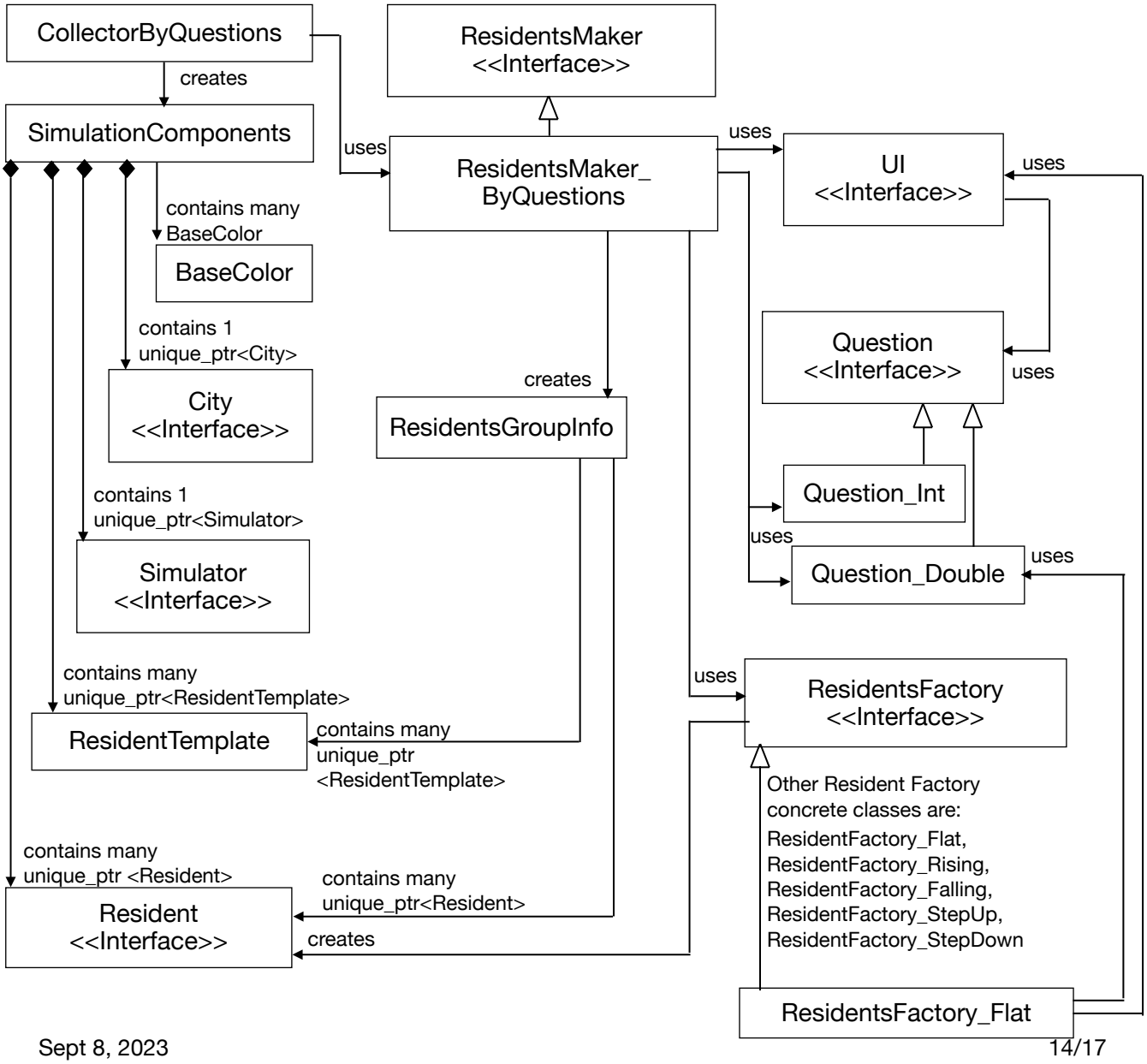
6.0 F. ResidentsMaker_ByQuestions UML Diagram

This diagram focuses on how ResidentsMaker_ByQuestions creates Residents that are eventually placed inside the SimulationComponents object.

main.cpp passes CollectorByQuestions a vector of unique_ptrs of ResidentFactories. CollectorByQuestions passes the ResidentsFactories into ResidentsMaker_ByQuestions.

ResidentsMaker_ByQuestions uses a UI, Question_Ints, and Question_Doubles to get basic information about the to-be-created Residents (how many Residents and which subtype). Then ResidentsMaker_ByQuestions calls on a specific concrete ResidentFactory to create the Residents.

The ResidentFactory uses a UI and Question_Doubles to get information (specific to this type of ResidentFactory) and creates a vector of unique_ptrs of Residents. This vector is moved into a ResidentsGroupInfo object by ResidentsMaker_ByQuestions. CollectorByQuestions then moves the vector from ResidentsGroupInfo into the SimulationComponents. Finally that SimulationComponents object is passed back to main.



7.0 Rubric Points

7.0 A. Requirement 1: A variety of control structures are used in the project. The project code is clearly organized into functions.

"if else" control structure in main.cpp's createSimulationComponents() method, at line 161:

If the command line execution statement includes an argument, then an input file is being used and a CollectorByInputFile object is used to make the SimulationComponents object. Example of an execution statement with a filename argument is "./sdl2-ttf-sample txt-inputs/txt-ex0simA.txt".

Otherwise the user is asked if they will be using the pre-made-simulation examples or be answering questions to create a simulation. If the variable usingExamples is true, then a CollectorByExamples object is used to create the SimulationComponents. Otherwise, a CollectorByQuestions object is used to ask the user questions and make the SimulationComponents.

"for loop" control structure in main.cpp at line 121:

The for loop at line 121 processes each run in a for loop. (A simulation is made up of many runs.) For each run the loop gets the results in a ResPerHouse object from the "simulator->run()" function. Then updates the RunMetrics object with these results. Finally, it prints the output of the the run using the Printer_Graphic and Printer_CmdLine objects.

"switch statement" in Color.cpp at line 81:

Color.cpp operator<< (ostream& os, const Mood& obj) method uses a switch statement to return the correct corresponding string for the given Mood. It is used in GrColorKey_Basic.cpp line 148.

```
labelStream << " " << mood;
```

organized functions in CityState_Simple.h

CityState_Simple.h is clearly organized into functions. It keeps track of where the residents are living as they move out and into houses. It does this using the moveOutAndIntoHouse(), moveIn(), and moveOut() methods. Please see tests/test_CityState_Simple.cpp line 243 as an example of how CityState_Simple is used.

7.0 B. Requirement 2: The project reads data from a file and process the data, or the program writes data to a file.

See CollectorByInputFile.cpp createSimulationComponents(…) method on line 25:

To run the program with a text file as input, type the filename after the executable name.

```
./sdl2-ttf-sample txt-inputs/txt-ex0simA.txt
```

This will trigger a call to CollectorByInputFile's createSimulationComponents() method (see line 165 in main.cpp). The method uses an ifstream to get each line and then uses string's find() method to find a specific descriptive word. Each descriptive word corresponds to or contributes to the making of an attribute of SimulationComponents.

For instance in line 1 of txt-ex0simA.txt, the “<random_number>” specifies the random number used in the simulation.

```
<random_number>
    1
</random_number>
```

When the createSimulationComponents() method is done, the SimulationComponents object is returned to main.

7.0 C. Requirement 3. All class members that are set to argument values are initialized through member initialization lists.

Most of the classes use member initialization in their constructors. See Axis_Basic.cpp lines 6 - 24.

```
Axis_Basic::Axis_Basic(
    bool forward,
    int crossPixel,
    int lowVal,
    int highVal,
    int pxPerUnit,
    int startOffset,
    int endOffset
):
    _forward{forward},
    _cross_pixel__px{crossPixel},
    _low_val{lowVal},
    _high_val{highVal},
    _start_val{forward ? lowVal : highVal},
    _end_val{forward ? highVal : lowVal},
    _px_per_unit{forward ? pxPerUnit : -pxPerUnit},
    _start_offset_m{startOffsetMultiplier},
    _end_offset_m{endOffsetMultiplier}
{}
```

See HappinessFunc_Falling.cpp lines 6-13.

```
HappinessFunc_Falling::HappinessFunc_Falling (
    double happinessWithNoNeighbors,
    double happinessAtZeroDiversity,
    double happinessAtOneDiversity
):  _happ_with_no_neighbors{happinessWithNoNeighbors},
    _happ_at_zero_diversity{happinessAtZeroDiversity},
    _happ_at_one_diversity{happinessAtOneDiversity}
{…}
```

7.0 D. Requirement 4. One member function in an inherited class overrides a virtual base class member function.

HappinessFunc is a pure virtual class. Its concrete classes are:

1) HappinessFunc_Flat 2) HappinessFunc_Falling 3) HappinessFunc_Rising

2) HappinessFunc_StepDown 5) HappinessFunc_StepUp

They all override the base class's virtual methods. See the following lines in HappinessFunc.h, HappinessFunc_Flat.h, and HappinessFunc_Flat.cpp.

Method Name	HappinessFunc.h line number	HappinessFunc_Flat.h line number	HappinessFunc_Flat.cpp line number
1) calcHappiness()	23	27	23
2) toStrBasic()	28	32	38
3) getLargestValue()	30	34	50
4) getSmallestValue()	31	36	55

7.0 E. Requirement 5. At least two variables are defined as references, or two functions use pass-by-reference in the project code.

The askForGridWidth() method uses pass by reference. See CityFactory_Grid.h line 34 and CityFactory_Grid.cpp line 19.

```
int CityFactory_Grid::askForGridWidth(const UI& ui, int maxWidth)const
{…
```

The implementAddTicksAndLabels() method uses pass by reference. See GrAxisHorizontalSimple.h line 97 and GrAxisHorizontalSimple.cpp line 119.

```
void implementAddTicksAndLabels(
      vector<Rect>& ticks,
      vector<TextRect>& texts) const;
```