Math City
Final Document

Alyse Haynes, Heidi Lewis, Nathan Sapun, Zach Cabell-Kluch

I. Abstract

Math City, a game designed to help fifth graders develop math skills, is a simulation in which students can build cities complete with power lines, factories, coal plants, wind-turbines, police stations, and hospitals. The money required to expand the city can be earned through correctly answering math questions. We have updated this game to include varying weather based on season, as well as solar panels. The weather affects energy outputs for the wind-turbines and solar panels. These upgrades allow the game to be more interactive by providing a challenge between managing power and pollution. This also provides a basis for users to learn the importance of sustainable development. With these additions, the user will have to use critical thinking in order to design and implement the best city. In order to accomplish this task, we implemented icons for the solar panels, calculated energy generated by different types of weather, and ensured that the energy output for both the solar panels and the wind-turbines varies appropriately with the weather.

The code for the game is written in Java. We approached this project by first analyzing the original source and then refactoring. It was then possible to incorporate the solar panels into the code. We also developed a way to connect the weather to the solar panels and wind-turbines, so that their power output reacts accordingly.

We tested our program with students ranging from 4th to 6th grade. They enjoyed the weather functionality, and seemed to grasp the concept that the power from renewable energy corresponded to the changing weather. However, our testers found the game a bit lacking for a fully interactive, interesting game.

II. Introduction to the Project

The client, Dr. Polycarpou, is a professor at the Colorado School of Mines (CSM). She has been developing Math City along with student researchers for the past few years. Math City is a game designed to heighten fifth-grade students' interest and knowledge in math. Students answer math questions to earn money they can use to build a city. This is accomplished by building fire departments, police stations, hospitals, and power plants. Since the basic architecture of the program was already developed, we updated some of the options and functionality of the game to make game play more challenging. Our primary task was to add solar panels to the game to increase awareness of the importance of renewable resources. We also added season appropriate weather conditions, which affect the power generation of the solar panels and wind turbines. These additions should encourage students to think critically about the balance of power generation and pollution. In order to create these additions, we first refactored the code, decoupling classes and adding more efficient code where possible.

After our primary tasks were completed, we added load and save functionality to the game, to enable students to save their city and come back to it later. The game can also now pause/unpause, to enable short breaks during the game. We also added a City History line graph, which allows the students to see how much of each energy they are using in the game throughout the lifetime of the city. There is also a graph that displays the current energy statistics.

III. Requirements

A. Functional requirements

1. Added varying weather conditions which are season appropriate, e.g., no snow in the summer months.
2. Added solar panels, which vary their output of energy based on current weather conditions.
3. Updated wind-turbines with energy output that varies based on the weather, similar to the solar panels.

Our team also added these requirements:

1. Save and load functionality.
2. Pause and unpause functionality.

B. Non-Functional requirements

1. The code was written in Java.
2. The code was refactored to enhance readability and extensibility.
3. Added J-Unit tests to any part of the code that we altered.
4. Once the game was updated with our additions, it was tested with the intended demographic.

IV. Design

Refactoring was a major portion of our alterations to the original design. We separated different classes, where appropriate, to make the code more robust; this included separating classes for multi-player functionality within Economy, creating enumerated types, deleting duplicate code, and adding comments.

The original system for Math City was designed for single and multi-player use. The multi-player functionality of the game was commonly threaded throughout the code as a Boolean and the Economy class was no exception. The first thing we did when refactoring the Economy class was separate multi-player from single player functions. Other refactoring methods performed on the Economy class included moving methods from the Economy class to more appropriate classes. For example, the economy of the game used a timer and also kept track of the day, year, status of the world, and cash on hand. Now that it has been refactored, Economy handles money, the status of the world is contained within the world section, and the time ticking is maintained in the Day Monitor class. The UML of the refactored Economy class is shown in Figure 1.

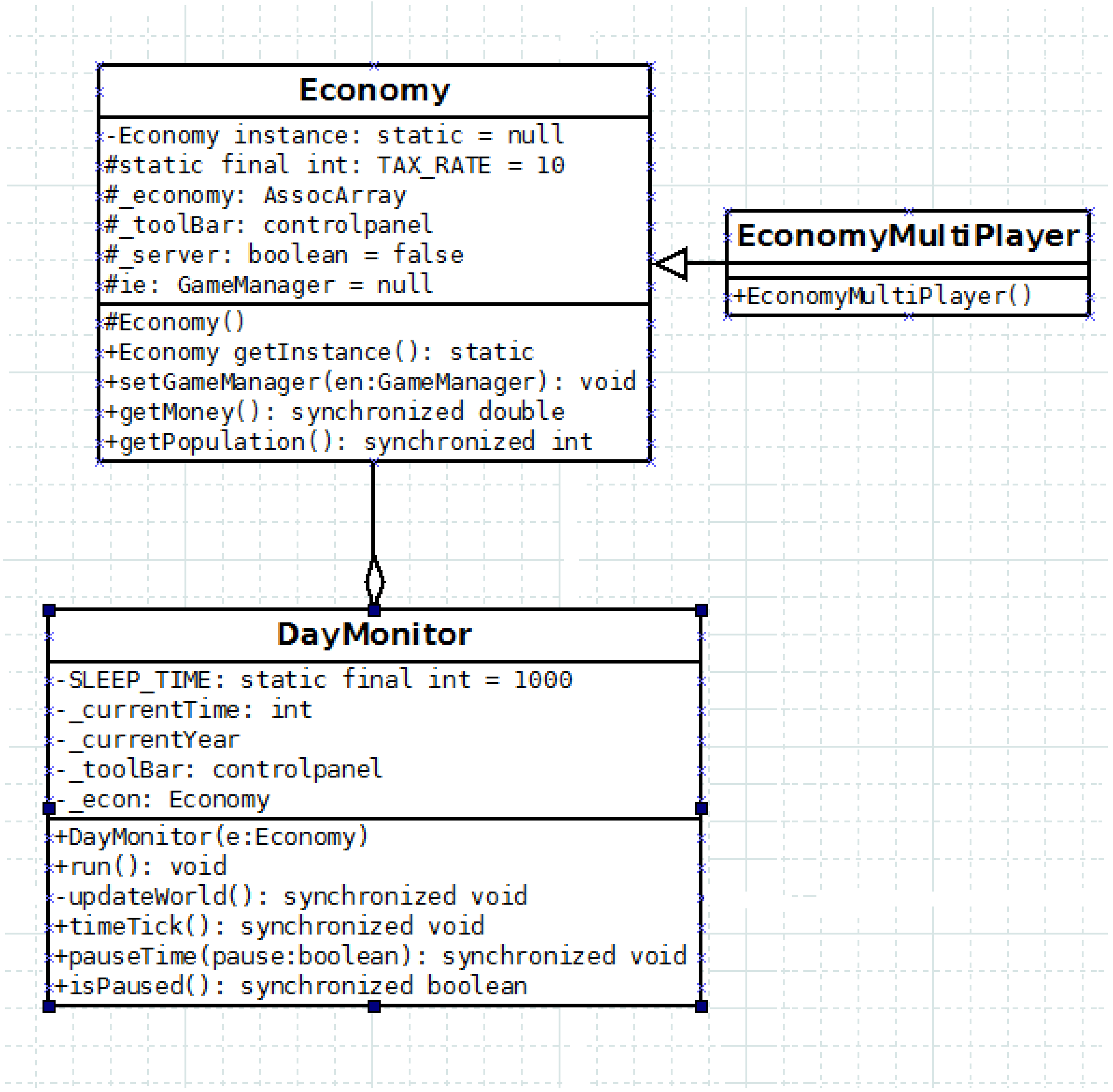
Economy
-Economy instance: static = null
#static final int: TAX_RATE = 10
#_economy: AssocArray
#_toolBar: controlpanel
#_server: boolean = false
#ie: GameManager = null
#Economy()
+Economy getInstance(): static
+setGameManager(en:GameManager): void
+getMoney(): synchronized double
+getPopulation(): synchronized int
EconomyMultiPlayer
+EconomyMultiPlayer()
DayMonitor
-SLEEP_TIME: static final int = 1000
-_currentTime: int
-_currentYear
-_toolBar: controlpanel
-_econ: Economy
+DayMonitor(e:Economy)
+run(): void
-updateWorld(): synchronized void
+timeTick(): synchronized void
+pauseTime(pause:boolean): synchronized void
+isPaused(): synchronized boolean

Figure 1 - Refactored UML of Economy Class

The Economy class also works separately from the DayMonitor. Every second DayMonitor tells WorldGrid to update, which calculates population, number of jobs, and happiness data. Every 5 seconds of real time equals a day in the game, and after 7 days DayMonitor tells the weather to change. Once the weather changes, this information is passed into the hash table where it can then be accessed by other classes. Every 365 days, DayMonitor tells Economy that it’s time to collect taxes. This is the only time DayMonitor interacts with Economy. After 365 days the ToolBar updates to reflect a year had passed. The communication diagram for Economy with the other classes is shown in Figure 2.

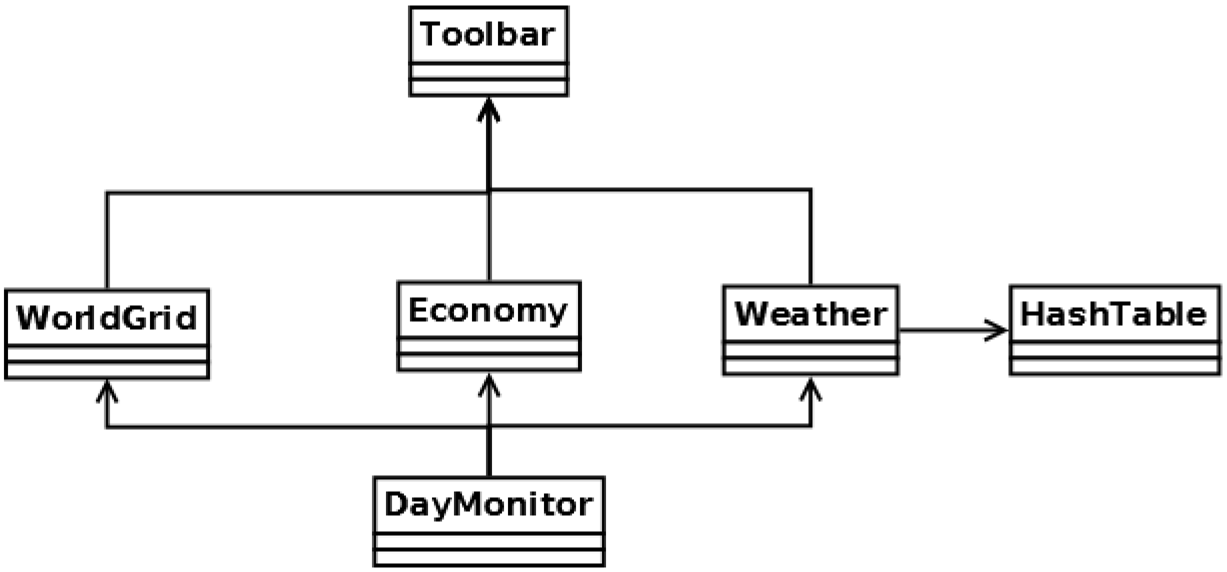

Figure 2 - Communication Diagram for Economy with other classes

With this new refactored design, we implemented a weather class, which changes based on the seasons. The seasons are determined based on a date stored in the Day Monitor class. We created solar panels, which have functionality similar to windmills, but are affected by sun intensity rather than wind intensity. The design detailing the weather interaction with the other classes is described in the weather module and in Figure 3.

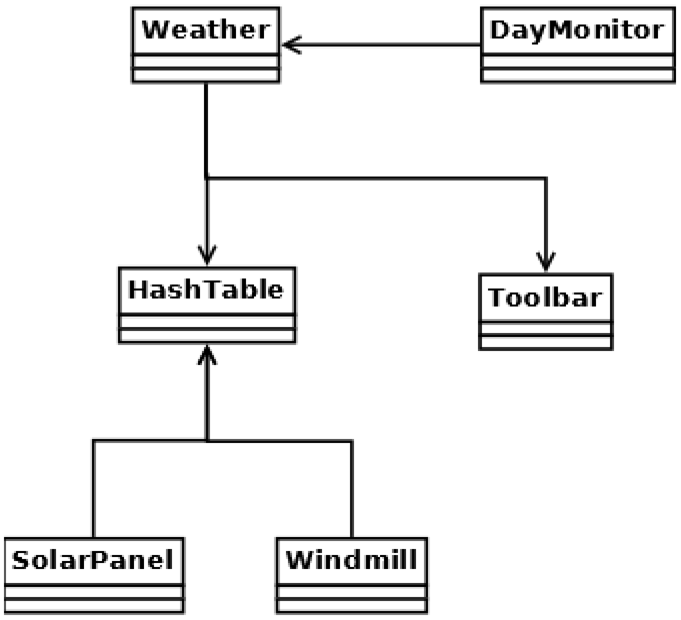

Figure 3 - Communication Diagram for the weather class

The original design for Math City included coal plants and windmills for sources of power which had large, constant values for their power output. Once we began to change the functionality of windmill it became apparent that PowerDistribution, the class that calculates and distributes power, needed refactoring as well. The constant amount of energy that each energy source created also needed to be changed, and is more realistic now. For example, a coal plant may use 9 squares on the grid, but can power up to 20 houses; while a solar panel or a windmill in their highest energy outputs can only power 3 or 4 houses.

The PowerDistribution has been modified so that it will add up any connected power sources. Previously, each power source was distributed individually, and left over power was discarded. This wasn't as noticeable when power plants provided a large amount of power, but the newer power plants are smaller and their power would be discarded without powering anything. Now power distribution adds connected power sources and distributes the power to any connected buildings until it runs out of power to distribute. In Figure 4, with the old design, the fire department would never be powered, but with the new design it is.

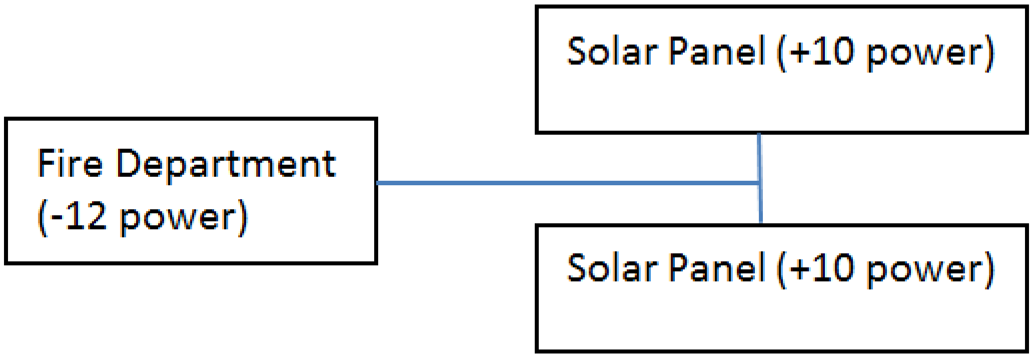

Figure 4 - Power distribution example

Since the power output from solar panels and windmills change with weather, it is also important to display how much energy the city is currently using, and of what type (coal, solar, wind). This information is valuable to students so they know how the power is affecting their city. In order to properly display this information a button has been added to the tool bar as shown in Figure 5. This button has a lightning bolt on it and a tool tip that says “City’s power information”.

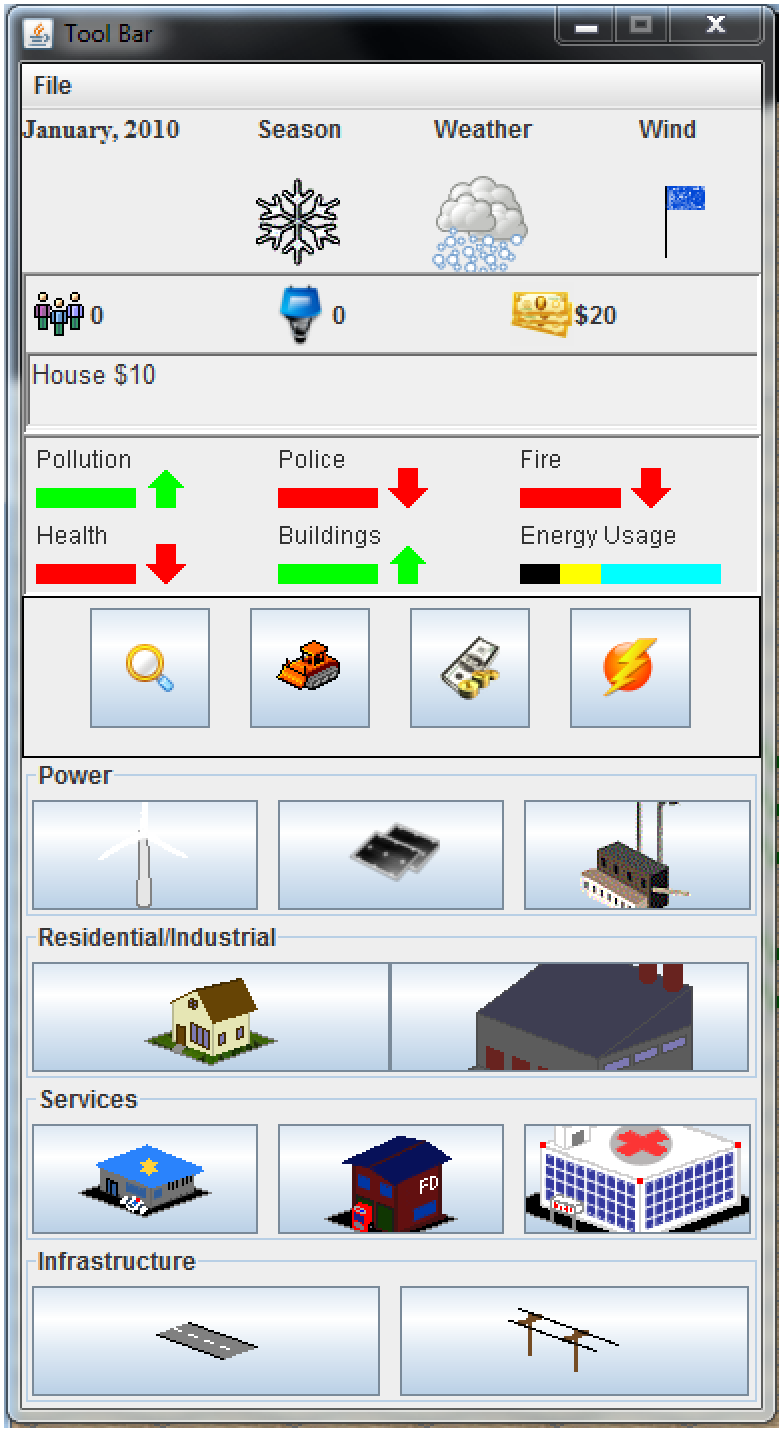

Figure 5 - Updated, complete tool bar with energy button

Once pressed, this button opens another window with a bar graph and a line graph. The bar graph has a vertical bar for each power source displaying the fraction of the city that each particular power source is powering; it also has a red bar displaying how much of the city is without power, shown in Figure 6.

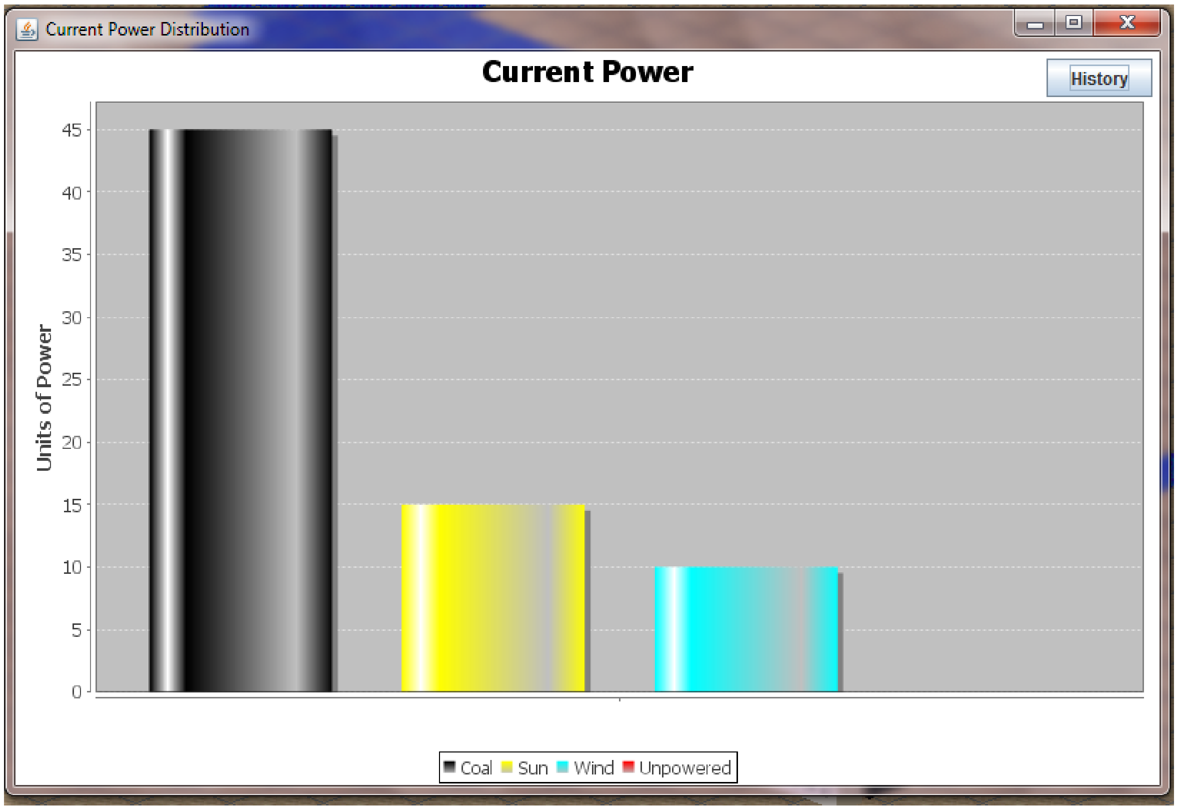

Figure 6 - Bar graph displaying current energy usage

The line graph displays all power sources as well and shows the relative amount of power that each source is providing to the city over time. Like the bar graph, it also has a red line representing when the city was without power, shown in Figure 7.

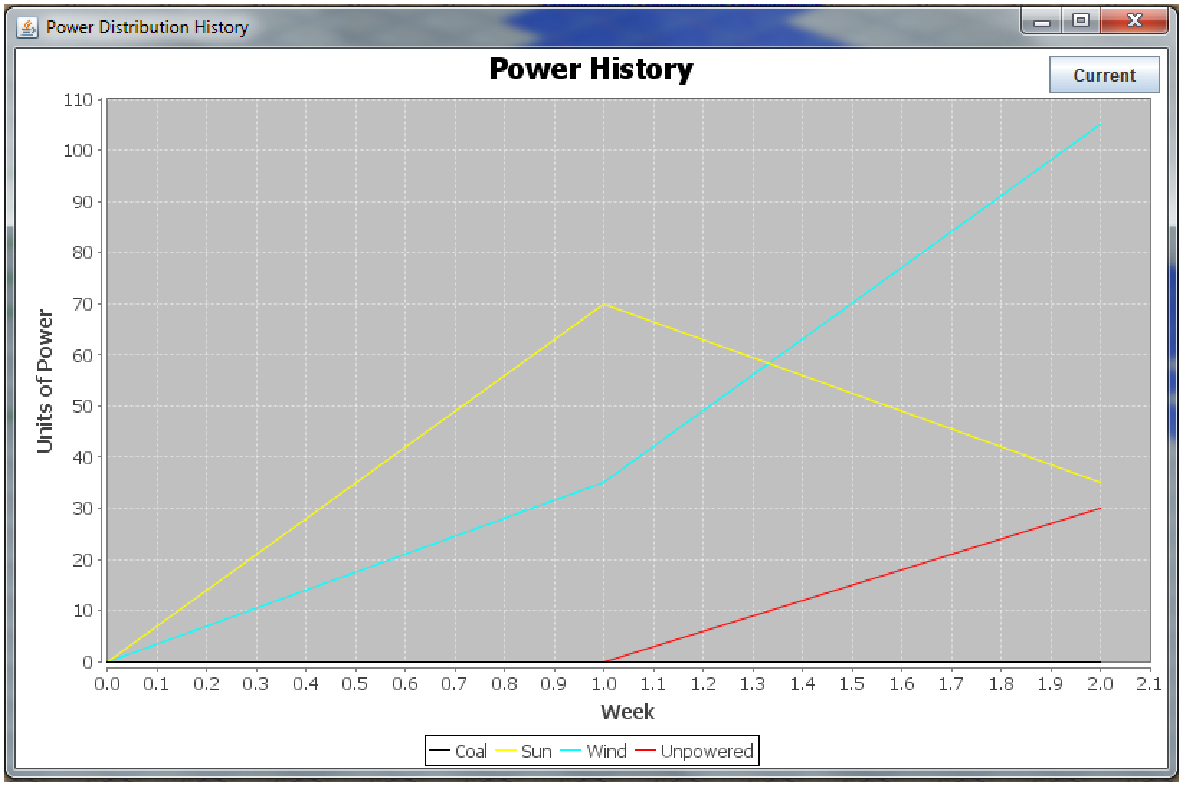

Figure 7 - Line graph of power history

In addition to the button and graph window, the tool bar will constantly display an "energy bar" that displays how much of an energy source is being consumed by the city at any given time, shown in Figure 8.

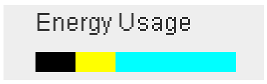

Figure 8 - Energy Bar displayed on tool bar

<u>List and description of modules:</u>

Weather module: Weather gets the month from the DayMonitor class. It determines what season the user is currently in based on the month. It then randomly selects from season appropriate weather conditions with a bias towards more common weather. Once the weather has been determined, weather updates the icons in Toolbar, and puts the wind and sun intensities in the hash table. The SolarPanel and Windmill classes then query the hash table to calculate how much power is generated.

Solar Panel Module: The SolarPanel class takes the sun intensity level from the hash table, and calculates the power drawn from the sun. We added a solar panel icon to the toolbar, so the user can click and place them onto the grid. The SolarPanel class updates the energy toolbar to show the user how much of their total power is being generated by solar energy.

Graphics Refactored module: Enumerated types have been created where possible in order to access information efficiently across the classes and improve clarity of code. We removed many functions that were never called.

Economy refactored module: Refactoring the Economy module moved the functions that keep track of population, jobs, unhappiness, and various other status-related variables into the Graphics module, where all of these variables were derived from. In addition to this, the section of the economy module that kept track of the passage of time was moved to its own, dedicated class. The Economy class now only handles how much money the user has on hand by adding or subtracting from the total as the user purchases buildings or answers math questions.

Power Distribution module: The PowerDistribution was tweaked so that less leftover power in each distinct power grid in the world was wasted. We accomplished this by having each new power generator seek out other generators connected to it. After finding each connected generator, the connected generator's power would be added to the first generator. After each connected generator is found, their total power is applied to the grid. While determining how much power is generated, the module calculates which type of power is generated, and in what amounts. This data is used in the City History module.

Current History and City History module: The history module tracks energy usage on a week-to-week basis. At the end of each week, information about the city's power production and use is stored. The user can click on an icon in the control panel to view this data at any time, and the city's power history is displayed as a line graph. The city's current power production is also displayed on a bar graph accessible in the same way as the history graph.

V. Implementation Details and Results

We developed the code in Eclipse and made use of JFreeChart, a Java library, which requires a copyright notice inside the program. We wrote JUnit tests for all new code. To accomplish our goals, we implemented/altered the following classes:

Power Distribution:
We altered PowerDistribution's algorithm to check the main corner of buildings when it calculates power consumption/generation. The original implementation relied on a building's size, and had a tendency to miss setting the additional sections of a building to powered, leaving several instances where a building's contribution was counted multiple times or not counted at all. PowerDistribution now adds up all the connected energy available and uses the renewable resources first. If renewable resources is not enough to power the city, PowerDistribution will then take energy from any nonrenewable resources.

Energy type:
While it would be possible to simply display all of the power that is generated in the city at any given time, we decided to only display the power that is *consumed*. The reasoning behind this is as follows: If a city can be powered entirely by renewable resources, it will avoid using its non-renewable resources, as they are typically more expensive and would be wasted anyways. Making this work included finding the proper way to total up the amount of energy needed to power the city. With the potential for several disconnected power grids it was slightly tricky to find how much energy was needed and fill this need prioritizing renewable.

Day Manager:
Originally, the code that monitored the in-game time was split over several classes. We decided to compile these sections into a single class, as this would make implementing the weather much simpler.

Hash Table:
While writing our code, we decided that relevant information that needed to be passed between classes should be written to the hash table. This allows classes to simply query the hash table instead of a separate class. This also allows greater freedom for JUnit testing, as we don't need to initialize many classes that would otherwise be dependent on each other.

Weather:
When we designed the weather class, we originally made the wind speed more dependent on what weather type was active. This was altered for game play purposes, as it could be potentially crippling to roll both low wind and low sunlight. Wind speed is

now more dependent on sun intensity, so low sunlight effectively forces high wind speed and vice-versa. Also, while designing the weather class, we decided that a month would be passed into the weather class, and the season calculated inside of the weather class instead of the DayMonitor class. This is because DayMonitor doesn’t really care about what season, just day, month and year. Also, it will allow future groups to fine tune the weather so it can be month dependent, instead of season dependent.

Save/Load:
Originally, the game did not allow for a current grid (game) to be saved or loaded. We added the save and load functionality to the game to allow students to save their progress and return to it later. The save function will save the current date, current amount of money, places of buildings, roads/power lines, and the history graph that displays the amount of power that each power source is providing to the city over time. When a student loads a game, it will replace everything that was saved into the current grid.

Pause:
The original game design did not have any functions to change or alter time. With the addition of renewable energy that changes power output with respect to changing weather, we thought it would be appropriate to add in a pause functionality to the game. If a student pauses the game, they can think about what to add to their city, what type of power sources it needs or even simply take a break without the fear of changing weather creating detrimental effects to their city. While the game is paused students will not be allowed to build on the grid. This is to help hinder any possible advantage one student may have over another if attempting to build the best city in a class room setting.

VI. Scope and/or Project Progression

As defined by the client, the scope of our project was to create weather and solar panels that interacted appropriately with the game. In order to complete this task, we found that refactoring the code was necessary. The first week, we planned the design, after familiarizing ourselves with the code. The next two weeks were spent refactoring the code. Once the code was refactored, we were able to implement the weather and solar panel functionality during week four. Since our initial requirements were done, we also added the history line graph during week four. Throughout week five we were able to finish the current history bar graph, add pause/unpause and save/load functionality to our project, leaving week six for presentations and documents.

VII. Conclusions & Future Directions

A. Conclusion

To create a functional addition to the game, we tested our code in many different ways. We used JUnit tests for all new code, to ensure that the code was performing correctly. We also tested the game by setting up certain scenarios inside the game to ensure it was functioning as expected. After the bulk of our coding was done, we were able to test our game with students. One student said, “I like that [the game] is realistic with changing weather.” Every student that tested it tried to use renewable energy in some way. There were confusing points in the game for most of the students, and many times they were not able to connect power and or buildings appropriately, as the power lines and roads were hidden behind the buildings in some cases.

We learned a lot from this project. The first thing we discovered was that poorly commented or uncommented code greatly increases the amount of time it takes to understand exactly what is going on in a project that you haven’t worked with before. Commenting code you develop is extremely important, even if it doesn’t seem like it at the time. Another lesson learned is not to take code at its face value. There were many times that a piece of code seemed to do one thing but did something completely different, creating errors. For example, when a building is found, the code iterated over the size of the building, with the intent of affecting all tiles that belonged to that building. In reality, the found portion of the building may not have been the main corner of the building, which is required for the iterator to hit every other tile belonging to that building. This caused several hard to track down bugs. We also learned that heavily coupled classes can be unpredictable. In some cases, it prevents J-Unit tests from being able to function properly, and if there is an error it can make tracking down exactly what the issue is extremely difficult. Along with coupled classes, program functionality should never be coupled with a GUI. It makes testing program functionality nearly impossible without creating an instance of the GUI, which should not be necessary.

B. Future work for this project

Since MathCity is an ongoing project, future teams will be further developing the code. As ascertained from testing with students, this game is not yet interesting enough to be interacted with in a classroom situation. Future development needs to improve game design to keep student’s interest. Also noted by the students, the math questions need to be refactored as the wording is vague and confusing. The question difficulty scaling needs reworking, to allow for a more reliable indication of student progress.

VIII. Glossary

J-Unit testing: An executable jar included in java that allows for test driven development. This allows for automated testing of the code so that the writer knows that the functions are working properly.

JFreeChart: A java library for creating charts and graphs.

IX. References

We used JFreeChart for our line and bar graphs. Copyright information is located under File->About in the program and is covered by the LGPL.