MS Project

Spreadsheet Model for Programming

Advisor

Dr.Timothy. A Budd

Submitted By: Mahesh Ambravaneswaran

Oregon State University
Corvallis, Oregon
June, 2000

CONTENTS

Acknowledgements

My work presented here deals with a spreadsheet programming interface design and it would not have been possible without perspective comments and advice from my major professor Dr.Timothy Budd. I got the inspiration to start this project after reading the abstract from Dr.Budd's web page. I'm extremely thankful to him for carefully wading through drafts of my project and offering invaluable critical insights. I'm very grateful to him for always being there when I had questions, for being very understanding and supportive when I had difficulties with the project and for offering extensive feedback. His views, comments and his extensive knowledge in the object oriented programming methodology was a great asset to my project.

This project was greatly influenced by Dr. Margaret Burnett's invaluable suggestions. My heartfelt thanks to her for patiently explaining the features of Forms/3 and offering helpful insights into the area of spreadsheet programming.

I would also like to specially thank Dr. Bella Bose for having agreed to be part of my committee. I would also like to acknowledge Saumya Venkataraman and Rudrakumar Venkatesan for all their help and continuous support throughout the project. Special thanks to Saumya for offering valuable feedback and for helping me format my report, in spite of her rigorous graduate student schedule.

Mahesh Ambravaneswaran

1. Introduction

This is an attempt to increase the power of a spreadsheet and try to use the spreadsheet as a powerful programming tool. The basic idea is to treat each cell of the spreadsheet as an object. The cell (Object) could be programmed, that is, the attributes and the functionality of the cell could be described by the programmer. This type of visual programming is extremely useful in cases such as simulation systems and when analyzing "what if" conditions.

The following are some of the advantages of taking a spreadsheet like programming approach.

- The task of programming is greatly simplified. This is later shown using some example problems which demonstrate the simplicity of spreadsheets.
- We are able to express relationships over many different data types, including numbers, strings, boolean, fonts, colors etc.
- The programmer of such an interactive application gets immediate feedback on the updates made to the system
- The complicated flow of control in the present day programming techniques could be avoided. This is makes testing and maintenance of such an software easier
- The programmer is saved the burden of expressing the computed result in an appropriate visual form for analysis. The two different tasks of computation of data and expression of results (which might actually be more difficult than computation) are bundled into one. The programmer is saved from the need of separate graphic programming

The spreadsheet model will make the line of distinction between a programmer and an end-user very thin. In fact, if the end-user has some basic programming abilities, he or she can change the application to fit their needs more appropriately.

2. Why take a spreadsheet approach to programming?

If we have to name a single software invention that revolutionized the way people perform numerical calculations with computers, it is undoubtedly the spreadsheets. Before the invention of spreadsheets, the only way to perform calculations would be to use a complex programming language such as FORTRAN. For the first time, spreadsheets allowed people not experienced with any programming language to do complex calculations easily.

The reason why spreadsheets became so popular is due the fact that they are conceptually very simple. The power of the traditional spreadsheets could be summarized more or less with the following statements

1. A spreadsheet contains a grid of cells whose value or formulae could be accessed
2. A formulae can refer to other cells (thus building the relationship between cells) or the results of computation of formulae of these other cells.
3. There is a standard library with some already defined functions available to the user.

2.1 Supremacy of spreadsheet over conventional procedural techniques

It is not difficult to see that this conceptually simple spreadsheet can actually be a powerful programming tool. In fact, this is already proved with the success of traditional spreadsheets. Many programmers found that using spreadsheets to perform their numerical calculations was much easier than trying to do the same in languages such as FORTRAN. Thus, we can safely say that the spreadsheets have long back established their supremacy over procedural languages in the area of numeric data representation and calculation.

One of the primary motivations of using the spreadsheet would be the possibility for **visualization of data (1)**. An important technique for understanding data is to apply an appropriate visualization model to it. Each cell of the spreadsheet can contain a data set. The physical appearance of the cell could be programmed to reflect the structure of data in the cell. Lets take a simple example of analyzing sets of data. Let each set of data be represented in a cell. Let the cells color be made to reflect the standard deviation of this set. Thus a cell's red color component could be made to reflect its standard deviation value. We can also make the cell color depend on the relative standard deviation among sets of data. This type of analysis is made extremely easy using the spreadsheet model. The relationship between the cells could be used to describe both the data and the view attributes. The duplication of sets of data could be made extremely easy through cloning.

This type of data visualization could be also used for genetic comparisons. Each cell's data could represent a genetic code. The similarity between two genetic codes could be analyzed by storing these two codes as distinct data sets in two cells. We could then establish a relationship among these two cells to identify the genetic similarity. The color difference between the two cells could be used as a parameter to identify the genetic similarity. The cells could be programmed such that any change in the genetic sequence of one cell is immediately processed and reflected.

Collective problem solving: The spreadsheet programming model encourages collective problem solving. The effective division of each cell of a spreadsheet as an object enables parallel development of Objects for collective problem solving.

The visualization of data when combined with some effective programming tools can make the spreadsheets an excellent tool for analyzing "what if" conditions of the real world. That is, it could server as a simulation tool.

For many applications the computation of data might not be the largest part of the problem. The big challenge might be to present the data in such a format that it is easy to understand and analyze. Also, applications, which basically take a system from one state to another, are good candidates for a spreadsheet model. These applications could be modeled using Finite Automaton.

One of the other main advantage is that the spreadsheets allows the features for “what if” conditions to be included into the application without any additional difficulty. Spreadsheets also eliminate one at a time approach through dependency propagation, which allows automatic recomputation of values among cells.

Also, spreadsheet approach proves effective in cases where we are trying to build an experimental model which is constantly changed as the parameters are changed. For example, experimenting with the weather model perfectly suits this scenario. In these cases, the power to visually see the effect of change in the parameters of the experiment goes a long way in finding the ideal solution.

The technique of cloning is very fundamental to the power of spreadsheet. Cloning is the process by which copies of cells (objects) could be created. A cell could be made as an identical copy of another and this process is called "**deep cloning**". In this case, the state and functionality of the two cloned cells are identical. Two or more cells could effectively represent the same object and this is called "**shallow cloning**". Any change made to the state or functionality of the common object of these cells is reflected in all the other cells.

We can take an example to demonstrate why taking a spreadsheet approach to problem solving can be extremely effective.

Problem-1

The problem is to model a physical environment, which consists of bodies at various temperatures in thermal contact with one another. Let us assume that new entities at different temperatures could be continually added to the system and existing entities could be removed anytime. We are interested in analyzing how the system reaches a thermal equilibrium condition and how the system adjusts to the introduction of new objects into the system and the removal of existing objects.

Modeling such a system involves the following:

1. An appropriate model has to found to represent the system as a whole

2. Each entity within the system has to be defined. The objects at various temperatures are the entities within the system. The state of the object is its temperature and the functionality is how the body reacts to changes in its environment. The functionality (reaction to environment) depends on its state (that is, how hot it is).

3. We have to specify the relationship between the objects of the system, which will define their functionality. Two bodies are "related" if they are in thermal contact with each other. That is, the name of the relationship is "thermal closeness"

4. We have to provide flexible easy ways to introduce new objects into the system and naming facility of objects has to be provided for easy coding. It also increases the ease at which testing and maintenance could be done. It makes the code for each cell more readable. Also, it is easier to understand the visualization.

5. We have to provide facilities to remove objects from the system

6. There has to be ways to define objects that are similar to existing objects. This will reduce the need to code identical objects again. Two objects are identical when their state and functionality are the same.

7. We have to define a graphic representation of the entire system so that a user could study the changes in the system as and when they occur.

8. We have to identify the situation when the system has reached thermal equilibrium.

The spreadsheet programming model is a very effective tool to program a system such as the above.

1. The entire spreadsheet could be used to represent the system.

2. Each cell of the spreadsheet represents an Object in the system. This fits in perfectly with a scenario where a system is a collection of its constituent entities, just as a spreadsheet is a collection of cells.

3. The cells could be made to represent the bodies at various temperatures by defining their state and functionality. The state of a cell could be described with a set of variables and the functionality could be defined using a set of equations that define the relationship of this object with the others.

   For example: For each thermal object (cell) of the system (spreadsheet) we could define a variable to hold the temperature value of the object. Variables could be defined to hold the temperature, mass and specific heat values of the object.

   To define the functionality of a cell, we could use equations (similar to formulae in traditional spreadsheets) to define the relationship between the various variables of a cell. We could use arithmetic operators and assignment statements to express the relationship thorough an infix expression.

4. The physical attributes of the cell could be modified to reflect the state of the cell. For example, the red color component of a cell could be made proportional to the temperature of the cell. This really helps in visualization of the system when it tries to attain thermal equilibrium.

5. Introduction of new thermal objects can be done in two ways :

   The first way is that a user clicks on a cell and that cell could be **programmed**. That is, the state and functionality of this cell could be defined. This becomes a new object in the system that is ready to interact with other objects when the spreadsheet is "executed"

   The second way is to deep clone an already existing object. **Deep Cloning** is a very useful feature in the spreadsheet model for programming. A new cell (object) that is identical to the original object could be created through an extremely simple "copy and paste" interface. Thus, new objects in the thermal system, which are identical to the already existing objects, could be created very fast. Thus new Objects could be introduced into the system very fast and the effect could be seen immediately.

   New functionality could be added to the cloned thermal object by defining more equations. Thus the child could have more added functionality than the parent. Also, the spreadsheet model I developed supports both absolute referencing and relative referencing. When an object is cloned, both its absolute and relative references are copied.

Shallow Cloning is a way in which different cells of the spreadsheet could be made to refer to the same object (cell). That is, any change made to the source cell ’A’ will be reflected in all the other cells that are shallow cloned from cell ’A’. We can analyze one situation in this thermal system scenario where shallow cloning would be useful. Lets assume that we are to analyze a very large thermal system with possibly a few hundred objects. Lets say that the person in charge of maintaining and analyzing this thermal system is just interested in the temperature changes and the final equilibrium temperature of just a few objects which are located at different locations separated by a large number of cells. In this case, a separate area of the spreadsheet cells that are adjacent to each other could be "shallow cloned" from these objects of interest. Thus the temperature changes of these "critical" objects could be observed by concentrating on one area of spreadsheet without having to jump around for the critical objects and observing their changes. Thus shallow cloning allows easy "preference of view" for objects of "supreme interest".

Each newly created thermal body can be given a name. This increases the readability of the cell code and also makes testing and maintenance easy.

6. Removal of thermal Objects.

   In case a thermal object has to be removed from the system, it could be very easily done in a spreadsheet programming model. The object could be very simply deactivated. A deactivated thermal object is not part of the thermal system anymore. It does not interact with any other cell (object) of the spreadsheet. Thus addition and removal of thermal system components could be done very easily. This gives a lot of flexibility in designing the system.

7. Thermal objects similar to the already existing objects could be made part of the system very fast using cloning methods described above.

8. As the spreadsheet itself is inherently graphical with visible cells representing objects and color attributes, no separate graphic programming is needed for the visualization of the system (**1**) .

9. **Identifying when the system reaches equilibrium:** This task is made extremely simple in my model for spreadsheet programming. This is because of the concept of cell stability that is built into the model. Each cell can define one of its state variables to be the stability variable. When the stability variable value does not change over two or more iterations, the cell is declared to be stable. When all the active cells of the spreadsheet are stable, the spreadsheet (system) is said to be stable. In this model, each cell (thermal object) could declare its "temperature" variable to be the stability

variable. Thus when the thermal object's (cell's) temperature does not change after successive interaction with its neighbors, the object is declared to be stable. When all the thermal objects are stable, the system as a whole is declared to be stable.

Thus, software projects such as the above problem of modeling the thermal system using spreadsheets is an example of how using spreadsheets simplifies the development of software applications.

1. The whole task of programming was greatly simplified as seen in the above example. This report also contains an example thermal system analysis. You can see that the number of lines of code is indeed very small. Trying to program the same model **using conventional languages** would have been much more complex.

2. Because the spreadsheet is a conceptually simple, **the task of programming** was simplified as seen in the above example. **The time taken** to program such a system is also drastically reduced compared to the time it takes when modeling in conventional languages.

3. If we had tried to program the same application in **conventional languages**, we might have had to deal with **complicated flow of control**. The spreadsheet model eliminated the complicated flow of control.

4. The addition of new thermal objects was made extremely simple and easy as it is just a mouse click operation. The user had to just program an already existing cell. The creation of other thermal objects, which are functionally similar to already existing ones, is made conceptually very simple through the **process of cloning.** Thus, the programmer is saved the task of having to program the behavior of new object. This **code reuse** saves a lot of time in the development of such applications. The process of trying to add and program new objects using **conventional languages** would have been a much more **complicated and difficult task.** Thus the spreadsheet offers much more flexibility in the design and execution of the program.

5. The removal of existing objects could be done through a single line "deactivate" statement.

<u>Problem-2</u>

Let’s say we are trying to model an illumination system for a room. The optical behavior has to be studied as light sources are introduced into the room. Each part of the room has a desired particular illumination value. We have to find out the optimal lighting arrangement for the room which achieves this. This process might be a trail and error process where new sources might be added, removed, repositioned until the desired illumination level is achieved. This problem applies equally well to acoustical studies also.

Modeling such a system involves the following:

1. We have to find a model that accommodates the whole system. We can see that the spreadsheet suits this situation perfectly. **The spreadsheet as a whole could represent the room.**

2. We have to identify and model light sources. Again we could see that the cells of the spreadsheet could be programmed to represent the light sources. This very easily goes with the model. The spreadsheet contains cells as a room might contain light sources.

3. We must be able to program the light sources. For easy visualization, the color of the light source could be made to reflect the intensity. The state information and functionality of each cell has to be defined. We have to declare variables to represent the intensity of the light source. Thus the state of each light source is defined. Each light source has an effect on its neighbors. **Thus the functionality of each light source is the way it affects the illumination of its adjacent area**. We can see that this perfectly suits the spreadsheet model I had developed.

- We can see that the spreadsheet model is inherently graphical. Thus each cell that represents a light source could have its color components depend on the intensity variable. Thus it is extremely easy to do and suits perfectly with the user visualization.

- The spreadsheet cells have default functionality associated with its neighbors. Thus it takes very less time to define the functionality of cells (light sources). This problem perfectly suits the **“neighbor friendly model” of the spreadsheet.**

- Each cell could affect the state of its neighbors, which is exactly what happens in the room illumination problem. Spreadsheet fit perfectly the conceptual visualization of this problem.

4. Once each cell has been programmed to represent a light source, we can see the effect immediately on the spreadsheet. Each cell affects its neighbor’s intensity and those in turn affect their neighbor’s intensity. We can visually see the effect immediately after programming each light source. Because both the physical attributes and the data are encapsulated within the cell, it takes very little time to program the system and see the effects.

5. **Addition** of new light sources to the system is made very simple through the process of cloning. Because the behavior of the light sources towards its neighbors are pretty much the same (namely, it obeys the inverse square law of light intensity transmission), we can duplicate light sources easily. Because the behavior is similar, only the state information (intensity, in this case) has to be changed, we can very easily do it with a single assignment statement. Thus creating new light sources is as simple as a “copy and paste” and a single assignment statement if the state information has to be changed.

6. **Removal** of existing light sources is also an operation that has to be repeated over and over again until the illumination of the room is satisfactory. The removal of an existing light source can be done very simply through a single "Activate false" statement on the cell. The effect of the removal is immediately reflected with the color change in all the affected cells of the entire spreadsheet. This will correspond to the corresponding change in lighting of the room.

7. **Repositioning of Light Sources**

   One of the fundamental operations to be done to achieve the ideal illumination of the room will be to change the position of light sources in the room and analyze the effect. The repositioning of light sources correspond to two very simple operation on the spreadsheet.

   - Deactivate the cell corresponding to the light source
   - Clone the cell and paste it into the desired position

   In fact, the ease at which any cell of the spreadsheet could be deleted (deactivated, which corresponds to removal of the light source) and cloned makes the model so inherently easy to program. The effect of the repositioning is immediately reflected with the corresponding change in the color of the cells, which in turn corresponds to the level of illumination of the room.

   Thus we can see that the system could be programmed very easily using spreadsheets, whereas it would have been a very time consuming and complex process when we try to use the traditional languages. The spreadsheet model makes the programming of "what if" scenarios very easy and it also provides immediate feedback on any changes to the system.

   Shallow cloning allows the user to concentrate on the cells which are important at that moment. This is particularly useful when you are dealing with a large number of light sources, which results in a very large spreadsheet. We can shallow clone the cells (light sources) of interest to physically adjacent locations of the spreadsheet to better observe the effect of changes in the lighting configuration.

3. Traditional Spreadsheets: A Comparison

Spreadsheets first appeared as a primary numerical calculation tool which revolutionized the way people used computers for numerical calculation. People started using spreadsheets for their numerical calculation needs rather than using high level languages such as FORTRAN.

The first electronic spreadsheet application to make a huge impact was **VisiCalc (2).** It was invented by Daniel Bricklin and Bob Frankston. By the fall of 1978, Bricklin had programmed the first working prototype of his concept in integer basic. The program helped users input and manipulate a matrix of five columns and 20 rows. The first version was not very "powerful" so Bricklin recruited an MIT acquaintance Bob Frankston to improve and expand the program. Bricklin calls Frankston the "co-creator" of the electronic spreadsheet. **VisiCalc** was a great success and was a real motivation for people to buy PC's.

The **VisiCalc magic** did not last long. In 1983, Mitch Kapor developed Lotus, which became an Industry Standard. **Lotus 1-2-3** made it easier to use spreadsheets and it added integrated charting, plotting and database capabilities. Lotus 1-2-3 established spreadsheet software as a major data presentation package as well as a complex calculation tool. Lotus was also the first spreadsheet vendor to introduce naming cells, cell ranges and spreadsheet macros.

The **Lotus-Improv** interface was different from that of traditional spreadsheets. It allowed addition of structures. It allowed data to be grouped into categories and groups and enabled operations on such groups. This was the first known spreadsheet to go beyond cell boundaries. The lotus-Improv took an **Object Oriented approach** in some sense that its formulae can be applied to Objects of a particular category, not just cells.

The next milestone was the **Microsoft Excel spreadsheet**. Excel was originally written for the 512K Apple Macintosh in 1984-1985. Excel was one of the first spreadsheets to use a graphical interface with pull down menus and a point and click capability using a mouse pointing device. The Excel spreadsheet with a graphical user interface was easier for most people to use than the command line interface of PC-DOS spreadsheet products. Many people bought Apple Macintoshes so that they could use Bill Gates' Excel spreadsheet program.

These traditional spreadsheets which are designed to be more friendly as a numeric calculation tool lack the features to be used as a general programming model. The traditional spreadsheet model is very much error prone because of the structure of the formulae definitions. The cells are referred using a (letter Number) string. Using such references to cells in formulae is **extremely error prone**. This is because it is very easy to commit an error and it is an extensive time consuming process to debug the error. This property of traditional spreadsheets makes them an unlikely candidate for a programming tool.

In order to make a spreadsheet a powerful programming tool the following needs has to be addressed.

- The definition of formulae has to be more readable and understandable. This in turn makes the testing and maintenance of the software developed using the spreadsheets easier.

- The debugging of the spreadsheet has to be possible

- The cells of the spreadsheet have to be made programmable. That is, it should be possible to define a set of variables for the state and equations for functionality of cells.

- The cells must be able to interact with one another in a clearly defined manner that is easy to understand and debug

- There must be features to specify the physical attributes of cells if the spreadsheet has to serve as effective graphic programming tool

In this report, it will be shown that the spreadsheet programming model which I had developed tries to satisfy all of the above requirements in order to serve as an effective programming tool.

We also have to note that the spreadsheet model is not intended to replace the procedural languages. Some of the application may be inherently suited to procedural languages. But there are certain types of applications for which spreadsheet provides a more natural and a convenient environment.

It is interesting to take a look at some of the other spreadsheet languages, which have already attempted successfully to make the spreadsheet a good programming environment.

Forms/3

Forms/3 is perhaps one of the most successful visual programming language which follows the spreadsheet paradigm **(3)**. In this language each cell is programmable through formulae which can contain references to other cell values, its own cell values and constants. One of the interesting features of Forms/3 is the ability of the cell formulae to refer to cell values at a previous moment in time. Forms/3 provides facilities of a general purpose programming language.

Forms/3 contains the following features:

1. **Ability to resize a cell**: This is not implemented in the **model** I had developed.

2. **Immediate feed back on change in cell parameters**: Any change made to the cell formulae is immediately reflected when the user processes the formulae through an "Accept" button. This type of immediate feedback is extremely useful. **In the model that I had developed,** I had implemented immediate feed back when values are set. But the formulae are not processed until the spreadsheet is executed. The formulae are just accepted and the results are reflected only when the spreadsheet is executed. Thus in Forms/3, the programmer gets immediate feed back on

3. **Deletion of cells**: Unwanted cells could be deleted by just "cutting" them. In my model, the user has to deactivate the cell through a "Activate false" statement in the program box for that cell.

4. Forms/3 allows **reusable abstractions**. It also uses some very interesting concepts such as automatic generation of reusable generalized version of formulae. This is very effectively demonstrated in the explanation of the program to calculate fibonacci numbers (for more information, please refer to Dr.Burnett's web page. Please see references at the end of this report) Forms/3 also provides **some very powerful time travel features**. It allows the definition of a sequence of values over time. The cell formulae could refer to earlier values which were held by the cell in the past.

5. Forms/3 also contains **features for animation.**

Forms/3 gives more power to a cell than many other spreadsheet models. Also, Forms/3 shows the result of change in formulae of a cell instantly without waiting for any event, whereas in my model, the formulae are accepted, but the effect is not shown until the spreadsheet is executed.

<u>Model Master:</u>

Model Master is an **Object Oriented front-end** for a Spreadsheet **(5)**. It allows the users to program the spreadsheets in an Object Oriented Programming language called model master. The Model Master compiles the program to spreadsheet formulae. The Model Master is developed in JAVA and uses the JavaCC (Java Compiler Compiler) as a parse generator. This is a new idea to integrate conventional Object Oriented programming with spreadsheets. Instead of numbered cells, which are easy to confuse, Model Master uses named variables. This reduces the chance that the programmer will make a link to the wrong cell or column.

In the **Model Master** the spreadsheet as a whole could be programmed using an object oriented programming syntax of the model master. When this code is compiled, code for the spreadsheet cell is generated. The Model Master provides advantages such as using names in place of cell references. My model also allows the user to specify their own preferred names to cells, which could be used in coding too. This makes the spreadsheet less prone to errors. The Model Master detects typical compilation errors such as cell values that are never set but used and cell values set and never used. The compiler for Model Master is written in JAVA. Model Master allows re-use of code through cut and paste. **Code re-use in my model could be implemented using cloning.** One of the other major differences between Model Master and my model is that, the Model Master treats a whole spreadsheet as an Object and allows creation of similar spreadsheets by cloning the whole spreadsheet. **This is different from my model where the cloning is at cell level.**

Model Master allows declaration of arrays to represent values of cells in each column. Each row represents the successive values of the column at successive time points.

Also, it is important to note that Model Master makes the difference between the user of the spreadsheet and the programmer more explicit, whereas my model and various others like the Forms/3 tries to make the line of distinction more thin. In my spreadsheet model there is no underlying language model on which the spreadsheet is based.

NoPumpG

There were languages such as NoPumpG, which greatly emphasized the spreadsheet model for programming as a tool to simplify the creation of interactive graphic applications when compared to other approaches **(6)**. This spreadsheet model is more geared towards graphic applications.

NoPumpG allows the cells to declare arbitrary number of complex formulae. The reason for this is to reduce the number of cells needed. Also, in NoPumpG, all the cells of the spreadsheet are divided into actual application cells and control cells. This type of division is not present in my model. The clock is used as a counter variable. This is useful in animations as well as to implement something like a while loop iteration in the code for each cell.

<u>4. Project Description</u>

My aim in this project is to expand the spreadsheet model to include fundamental techniques of programming in order to make spreadsheets a very effective programming environment.

The spreadsheet is viewed as a collection of cells where each cell is a programmable object. Thus the spreadsheet is a collection of objects. The cells of the spreadsheet are distinct entities, which interact with one another for a solution to a common problem.

By programmable, I mean that the **state and functionality** of each cell could be defined.

<u>State:</u> We could declare and initialize variables to define the state of a cell. The state of a cell also holds other critical information. **The color attributes** of a cell are stored as a part of the definition of cell state. The information on whether the cell is in a currently **activated or deactivated** state in stored as a part of the state information. The **stability** of a cell is a very important piece of the state information. The spreadsheet is as a whole considered being in a stable state when all the cells are stable. State also holds the user given **name** for the cell.

<u>Functionality:</u> The functionality of each cell is defined by specifying the way in which each cell interacts with its neighbors. In my model, each cell has default neighbors as well as explicitly declared neighbors. The default neighbors are those cells, which are physically adjacent. We can also declare two distant cells to be neighbors through explicit syntax. The functionality, which in other words defines the relationship the cell shares with its neighbors, could be defined with a set of infix expressions.

A fundamental programming technique in using the spreadsheet as a programming is **Cloning**. This spreadsheet model implements two types of cloning; the **Shallow Cloning** and the **Deep Cloning**. Deep Cloning is the process by which cells identical to an already existing cell could be created. The cloned cell retains the state and functionality from its parent cell. Cloning is an extremely efficient programming technique, which results in quick creation of new objects in the system. Because the state and the functionality are retained in the newly cloned cell, we are saved the time consuming activity of having to code new objects. Thus this enables **code reuse** at cell level. Thus deep cloning saves a lot of time in development efforts. **Shallow Cloning** results in the creation of cells that actually refer to the parent cells. All the cells that are shallow cloned share a single object of the parent. That is, identical copies of the Object are not created, but instead, cells that refer to the same single object are created. One of the main uses of shallow cloning is that it allows the implementation of "preference of view" in situations were we are dealing with a large spreadsheet with a few hundred cells (please refer the section "why take a spreadsheet approach for an example based discussion). In these cases, mostly, the user is just interested in some of the cells. But these cells might be physically separated by large distances in the spreadsheet. This makes the analysis difficult. In such situations, the cells of interest could be shallow cloned into an "interest area" of adjacent cells and their changes could be monitored.

Cell stability is also one of the building concepts of this model. Each cell could declare a variable to be the "stability variable". When the value of the stability variable does not change over two or more iterations then the cell is declared to be stable. When all the cells of the model are stable then the spreadsheet is said to have reached the stable state and the spreadsheet execution (object interactions) stop.

Lets take a closer look now at the features that make the spreadsheet model a powerful programming environment.

The Spreadsheet that I developed is a table of cells. The user could specify the initial number of cells of the Spreadsheet. Once the spreadsheet is launched the programming of each cell is done in a text window that pops up when a user clicks on that cell. Each of these cells has a default background and foreground color and a name, which could be modified by the programmer. The following are some of the features that define the power of a cell as a programmable object.

Specify the name of the Object

This could be made the name by which this Object will be known to all the other Objects in the spreadsheet. The reference of each cell through its name rather than the traditional forms of cell reference makes it easier to program, test and maintain. This also makes reference to neighbors easier. This also helps to make the code more readable.

Specify its initial color and later the make the color dependent on values of its variables during execution

This is very useful in case of simulating physical systems. The color of the Object is defined by three in-built variables for the three primary color components red, green and blue. These "color" variables could be assigned values of program variables. This gives the power to render a specific color to an Object depending on how the variables of the Object change.

Activate/Deactivate an Object

An Object could be activated or deactivated depending on whether the programmer chooses it to participate in the program execution (which takes the spreadsheet from an "unstable" to state to a "stable" state. More on this concept later). In case the user decides to "delete" a cell, this can be done using the "Activate false" statement inside the PDE Box.

Set variables

The programmer can define variable and set values for them. All the variables together define the state of the Object.

Set Infix expressions.

The programmer can define certain infix expressions to be evaluated. These expressions can involve variables inside this Object as well as variables of Objects that has registered with this Object as its neighbor. These expressions are evaluated one after another.

Set the name of "stability variable"

One of the variables of the Object (cell) could be set as the "stability variable". When the value of the stability variable does not change over two consecutive processing of the cell Object, the Object is said to have reached a stable state and it does not participate in further processing. The stability variable must be a local variable of the Object.

Set neighbors (similar to "friends" in C++)

The Object could register itself as a neighbor (friend) to another cell. This helps Objects friends access to state variables of this Object and provides an easy mechanism for referral. All the cells, which are adjacent to a cell, are its default neighbors.

Advertise

An Object can also register its name with all the other cells of the spreadsheet with one command (notification) called advertise. This can be done if all or most of the cells of the spreadsheet will need to access this cell.

Clone

One of the most powerful features of the spreadsheet-programming model is the ability to clone cells. That is, using a very simple mechanism of "copy" and "paste" the programmer could create new Objects that are identical to the existing objects. This is similar to the copy constructor in C++ or the cloneable interface in JAVA.

There are primarily two types of cloning that are possible. The "Shallow Cloning" and the "Deep Cloning"

Shallow Cloning: In case of shallow cloning, the cloned object (cell) is not created new, but is a pointer to an existing Object. That is, any change made to the "source cell" will be reflected in the "destination cell" and vice versa. There is just one Object that is pointed to by two different cells of the spreadsheet.

Deep Cloning: Deep cloning is more consistent with the real world cloning such as the Dolly sheep cloning. Another new Object (cell) is created with all the state, behavioral and physical attributes of the original cell. The source and destination cells are identical and independent copies of one another. This is more consistent with the copy constructor in C++ and cloneable interface in JAVA.

An Object Oriented Feature: This spreadsheet does provide the **inheritance** feature available in the conventional Object Oriented programming environments. This power is provided through the "Cloning" primitive described above. An "child cell" could be deep cloned from the parent cell. Then additional equations could be added in the child cell from the PDE Box.

Debug

The programmer can check at any time, the variable values of cells**.** This helps in debugging a cell.

<u>Spreadsheet Execution</u>

Once all the cells of the Spreadsheet have been programmed, the spreadsheet is ready for execution. When the spreadsheet is executed the cells undergo the following events to take the spreadsheet from an unstable state to a stable state.

1. Each cell starts executing its program. The code for each cell (which was typed in the PDE Box) is stored inside that cell.

2. When the cells are interacting with one another (talking to each other), the color of each cell might change depending on the how it was programmed. This could be used as a visual feedback on the status of each of these cells (Objects).

3. When a cell’s "stability variable" does not change over two iterations, it announces that it has reached a stable state.

4. When all the cells of the spreadsheet reach a stable state, the spreadsheet as a whole is said to be stable.

5. When the spreadsheet reaches a stable state, the execution has stopped.

6. The final result of execution is the set of states of all the participating cells of the spreadsheet. The visual representation of the result is useful in simulation systems.

5. Spreadsheet Model for Programming: A Detailed Look at the Solution

In this Project, I developed a spreadsheet based programming interface. This was done primarily using the Java Programming and mostly used the Java Swing Components.

A listing of the syntax and features of this programming environment could be found in the Appendix. I will try to demonstrate the power of the Spreadsheet paradigm using an example.

Programming each cell in the PDE Box

In order to program a cell in the spreadsheet the following powerful features could be used

1. Activate <boolean value>

If you want a cell to participate in the spreadsheet execution, the cell has to be activated through a "Activate true" statement inside the PDE Box. This will allow that particular cell to interact with its neighbors. A cell could be deactiavted anytime using a "Activate false" statement. This feature is useful in cases where you would want to "delete" an Object. All the cells are initially deactivated by default.

Example: Activate true

2. SetColor <red_int_value> <green_int_value> <blue_int_value>

The user is allowed to set the initial color of the cell using this command in the PDE Box. The three integer arguments represent the red, green and blue components of the Object color. The maximum value of the a color component can be 255.

Example: SetColor 255 50 255

3. SetName <name_string>

The user can also specify the initial name of the cell using SetName function. This name gets reflected on the spreadsheet cell. Initially, each cell's name is its co-ordinate positions inside the spreadsheet. This could be changed using the SetName command. This is very useful for the programmers and users to easily recognize the real world entity they might be trying to model. This name could also be used in the "Advertise" function described later. This results in a very readable and hence an easily maintainable.

Example: SetName HotBody

4. SetNeighbor <co-ordinates> <name_string>

The SetNeighbor function lets a cell to declare itself as a neighbor of another cell in the spreadsheet. The co-ordinates are used to specify the row and column position of the cell which the source cell wants to befriend. The "name_string" argument specifies the name by which the source cell will be known to its friend.

Example: SetNeighbor 4,5 HotBody

Lets assume that this command is executed from the PDE Box of the cell 3,3(named HotBody). This command informs the cell 4,5 that the cell HotBody wants to be a friend and can be refereed using the "HotBody" prefix in the equations inside the PDE Box of cell 4,5. For example, inside the PDE Box of cell 4,5, a expression HotBody.temp would mean a reference to the value of variable "temp" in cell 3,3.

5. Advertise <name_string>

This is an extremely useful function, which enables the cell to inform all the other cells in the spreadsheet about how to refer to it. This again results in a very readable code and gives a way for all the other cells to access this cell. This is similar to SetNeighbor, but advertises its name to all the cells in the spreadsheet. An equivalent method would be to use "**SetNeighbor**" function with the co-ordinate argument for each cell in the spreadsheet.

Example: Advertise HotBody. If we assume that the source cell is 3,3 then all reference to HotBody in all the other cells would be translated to cell 3,3.

6. Set <variable_name> <value>

This command lets the users to declare and initialize a variable for the cell. This variable could then be used in other equations and expressions declared in the PDE Box.

Example: Set temperature 100. This results in initialization of variable named temperature to a value 100 in the source cell.

7. SE <infix_expression>

This is used to specify the expression to be evaluated when each cell is executed during the spreadsheet execution. The expression is an assignment statement and the right hand side is an infix expression to be evaluated. This infix expression could contain the following:

1. Variables from this cell
2. Variables from cells those have explicitly registered as friends of these cells
3. Variables of the 8 adjacent cells

The default neighbors can be refereed using the fixed reserved prefixes

P	Present Cell (Source Cell)
N	Top Cell
S	Bottom Cell
E	Right Cell
W	Left Cell
NE	Top Right Cell
NW	Top Left Cell
SE	Bottom Right Cell
SW	Bottom Left Cell

For example, the expression SE P.temp=E.temp+W.temp, adds the "temp" value in the right cell and the left cell and assigns the result to the temp variable in the present cell.

There are some default reserved fixed words for each cell to denote the color components. For example, P.red denotes the red component. In case you do,
SE P.red=255, SE P.green=0 and SE P.blue=0, the component takes a full red color. You can make the color component reflect some variable values as the spreadsheet executes. For example, if the red component has to depend on the temperature value of the cell, then SE P.red=P.temperature will do the trick.

All the variable names and their values of the cell are stored in a hashtable of that cell. All the equations are Stored as a Java Vector and are executed in order when the cell executes.

8. SetStable <variable_name>

This equation is very crucial to the stability of a cell. The variable defined in this function is used to determine whether a cell is stable or not. A spreadsheet is said to be in a stable state when all the variables in it are in a stable state.
The variable argument to the SetStable function is called the **Stability variable**.

Cell Stability:
A cell is said to be in a stable state when the value of the Stability variable does not change over two consecutive iterations. A cell that has reached a stable state does not participate in anymore executions. That is it stops interacting with other cells.

Spreadsheet Stability:
When all the cells in the spreadsheet are NOT in the unstable state.

Example: SetStable Temperature
In this case, the cell executes until the value of temperature variable does not change over two iterations (when the system has reached thermal equilibrium).

9. The Implementation of Cloning Primitive

The PDE Box contains facilities to implement the cloning described earlier.

Copy Button

The user can click on the "**Deep Copy**" or "**Shallow Copy**" Button depending on whether the he or she wants the cell to be shallow copied or deep copied (please see the explanation of these two concepts in the "Project Description" section above). This button is available in the PDE Box to copy the source cell of the cloning process. Then the user has to click on the destination cell. The PDE Box pops up. When the user then clicks on the "**Paste**" button. The destination cell is then a clone of the source cell.

Paste Button

The destination cell is cloned when the user clicks on the "Paste" button on the PDE Box of the destination cell.

10. Object Oriented Features

- The Spreadsheet is very much inherently an object oriented programming tool. Each cell of the spreadsheet is an object by itself. The data and the physical attributes of a cell (object) is fully encapsulated inside the object.
- The access to a cell's (object's) variables is restricted to its neighbors. The neighbors of a cell can be implicit or explicit

  Implicit Neighbors: Cells implicit neighbors are all the cells surrounding this cell. For example, for the cell 4,4 there are eight implicit neighbors, which could be referred using the N, S, E, W, NE, SE, NW, SW prefixes.
  Explicit Neighbors: We can declare a cell to be a neighbor of another cell using the "SetNeighbor" and "Advertise" functions which were explained earlier.

- "Cloning" provides a base for implementing much more sophisticated inheritance concepts

6. Example Based Analysis

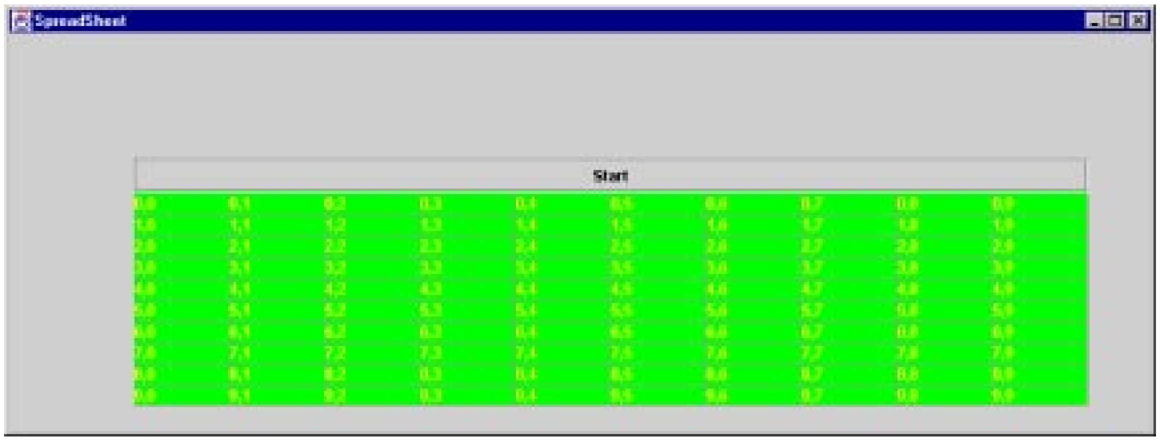
Problem:

Simulate a thermal system, which initially consists of bodies at various temperatures. Show the state of the system when it finally reaches a stable state. The Specific heat of all the bodies is assumed to be the same, but they differ in mass and temperature.

Solution using Spreadsheet programming

Steps

1. Launch the spreadsheet with the desired number of rows and column



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9

2. **HotBody Temperature: 100 F, Mass: 100**. Click on a cell (4,5) and do the following :

- Activate the cell(Object)
  Activate true

- Name the cell as the HotBody
  SetName HotBody

- Advertise the cell name so that all the other cells will know how to access this cells parameters.
  Advertise HotBody

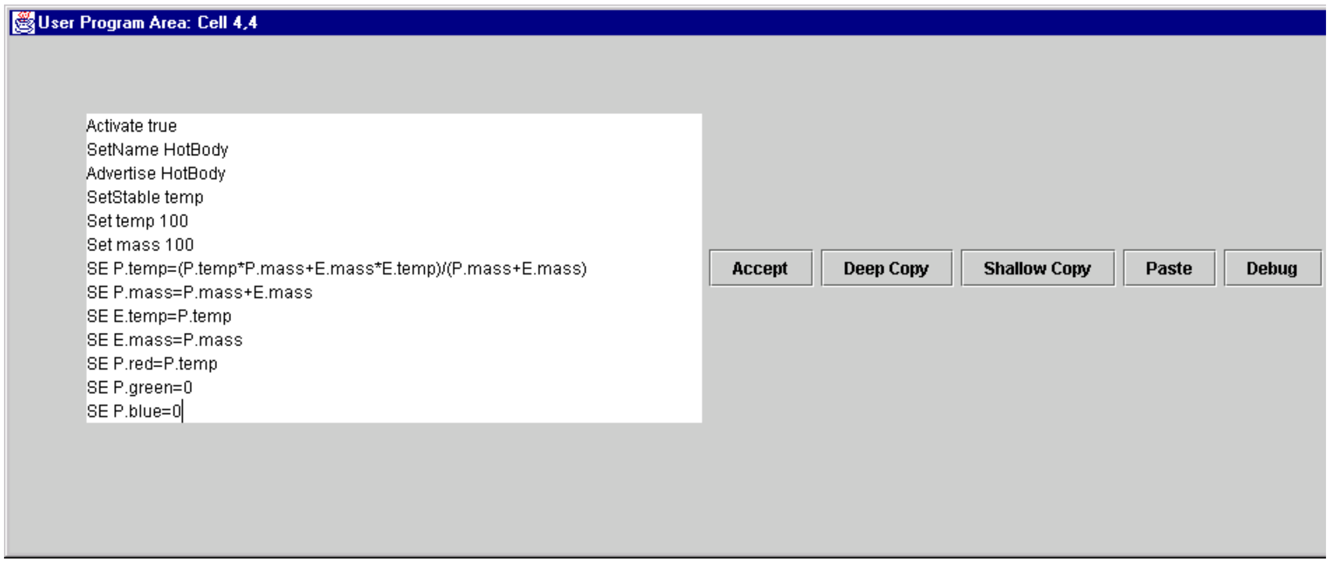
- Set the cell variable (that is the HotBody characteristics). The temperature and mass.
  Set temp 100
  Set mass 100

- Set the Stability Variable. In this case, it is the temperature of the body. The cells(thermal objects) will stop interacting with each other when the system reaches thermal equilibrium(when the temperature of an Object does not change when it interacts with its surrounding objects)
  SetStable temp

- Set the thermodynamic equations to be evaluated. **Set the red component of the Object to reflect how hot it is.**
  SE P.temp=(P.mass*P.temp+E.mass*E.temp)/(P.mass+E.mass)
  SE P.mass=P.mass+E.mass
  SE E.temp=P.temp
  SE E.mass=P.mass
  SE P.red=P.temp

SE P.green=0

SE P.blue=0



```
Activate true
SetName HotBody
Advertise HotBody
SetStable temp
Set temp 100
Set mass 100
SE P.temp=(P.temp*P.mass+E.mass*E.temp)/(P.mass+E.mass)
SE P.mass=P.mass+E.mass
SE E.temp=P.temp
SE E.mass=P.mass
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```

3. **MediumA: Temperature 50 F, Mass 50**

    Similarly, execute the following for MediumA.

Activate true

Advertise MediumA

SetName MediumA

Set temp 50

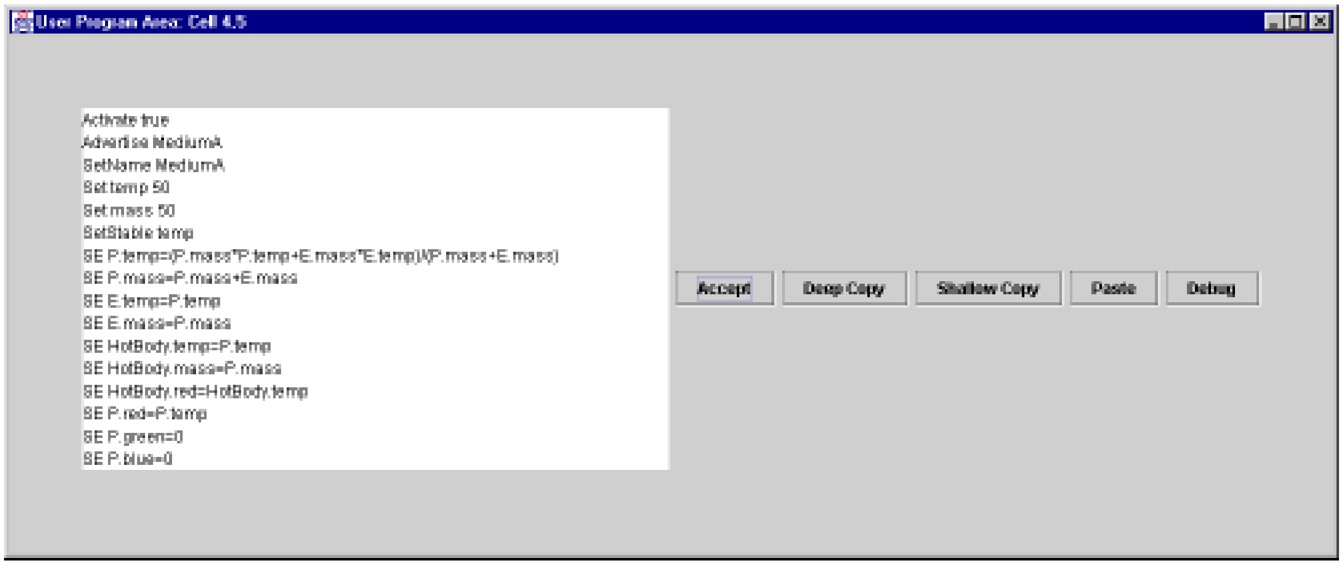
Set mass 50

SetStable temp

SE P.temp=(P.mass*P.temp+E.mass*E.temp)/(P.mass+P.temp)

SE P.mass=P.mass+E.mass

```
SE E.temp=P.temp
SE E.mass=P.mass
SE HotBody.temp=P.temp
SE HotBody.mass=P.mass
SE HotBody.red=HotBody.temp
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```

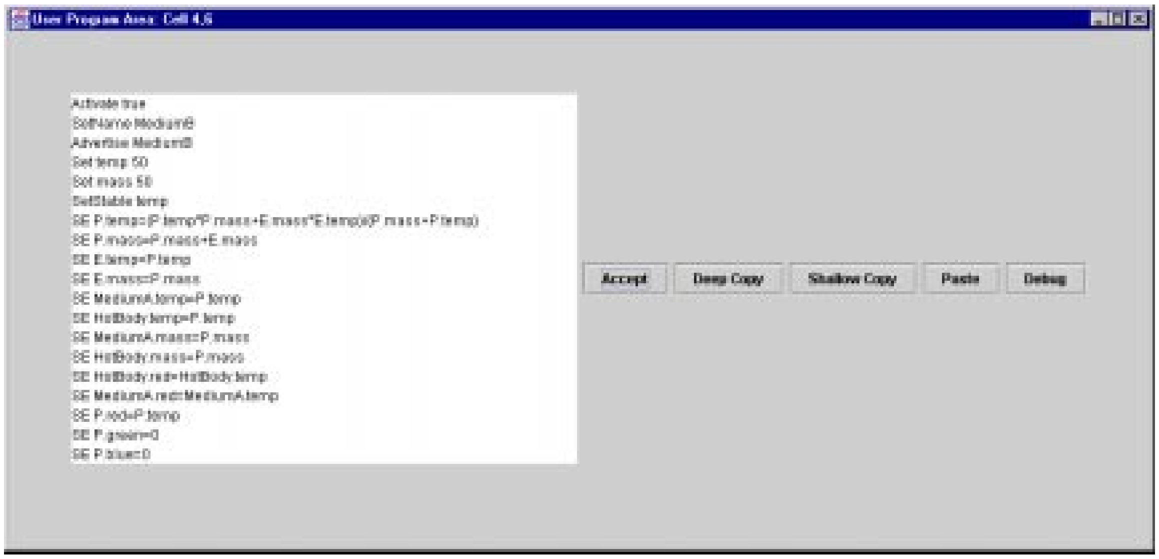


```
Activate true
Advertise MediumA
SetName MediumA
Set temp 50
Set mass 50
SetStable temp
SE P.temp=(P.mass*P.temp+E.mass*E.temp)/(P.mass+E.mass)
SE P.mass=P.mass+E.mass
SE E.temp=P.temp
SE E.mass=P.mass
SE HotBody.temp=P.temp
SE HotBody.mass=P.mass
SE HotBody.red=HotBody.temp
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```

4. **MediumB: Temperature 50 F, Mass 50**

Similarly, execute the following for MediumB

```
Activate true
Advertise MediumB
SetName MediumB
Set temp 50
Set mass 50
SetStable temp
SE P.temp=(P.mass*P.temp+E.mass*E.temp)/(P.mass+E.mass)
SE P.mass=P.mass+E.mass
SE E.temp=P.temp
SE E.mass=P.mass
SE MediumA.temp=P.temp
SE HotBody.temp=P.temp
SE MediumA.mass=P.mass
SE HotBody.mass=P.mass
SE HotBody.red=HotBody.temp
SE MediumA.red=MediumA.temp
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```



```
Activate true
SetName MediumB
Advertise MediumB
Set temp 50
Set mass 50
SetStable temp
SE P.temp=(P.temp*P.mass+E.mass*E.temp)/(P.mass+P.temp)
SE P.mass=P.mass+E.mass
SE E.temp=P.temp
SE E.mass=P.mass
SE MediumA.temp=P.temp
SE HotBody.temp=P.temp
SE MediumA.mass=P.mass
SE HotBody.mass=P.mass
SE HotBody.red=HotBody.temp
SE MediumA.red=MediumA.temp
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```

5. ColdBody: Temperature 0 F, Mass 50

Similarly, set the cold body parameters

Activate true

SetName ColdBody

Set temp 0

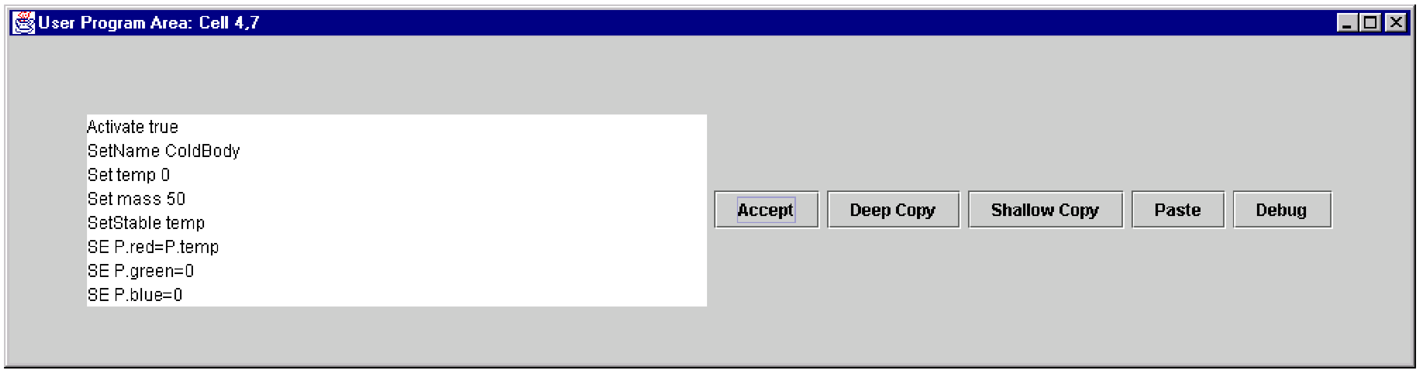
Set mass 50

SetStable temp

SE P.red=P.temp

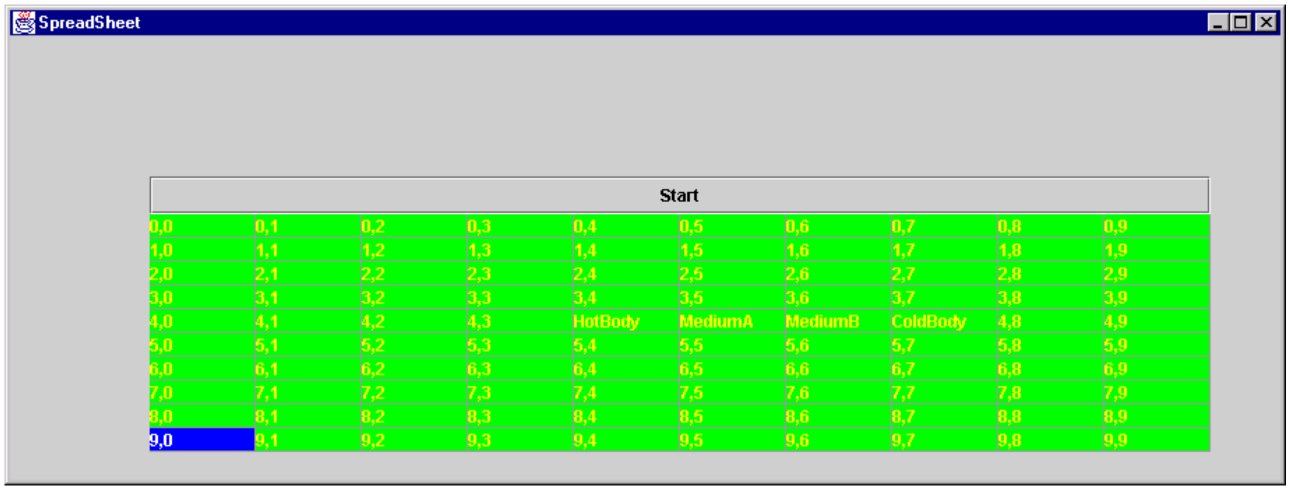
SE P.green=0

SE P.blue=0



```
Activate true
SetName ColdBody
Set temp 0
Set mass 50
SetStable temp
SE P.red=P.temp
SE P.green=0
SE P.blue=0
```

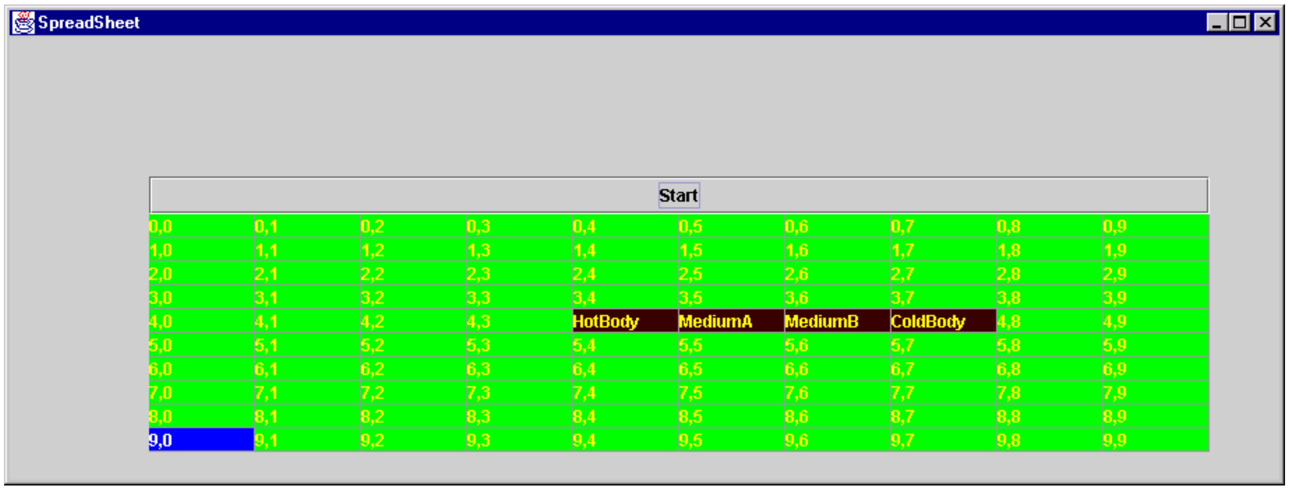
After all the above statements have been **"Accepted"** the spreadsheet looks something like the following



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9
4,0	4,1	4,2	4,3	HotBody	MediumA	MediumB	ColdBody	4,8	4,9
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9

Spreadsheet Execution

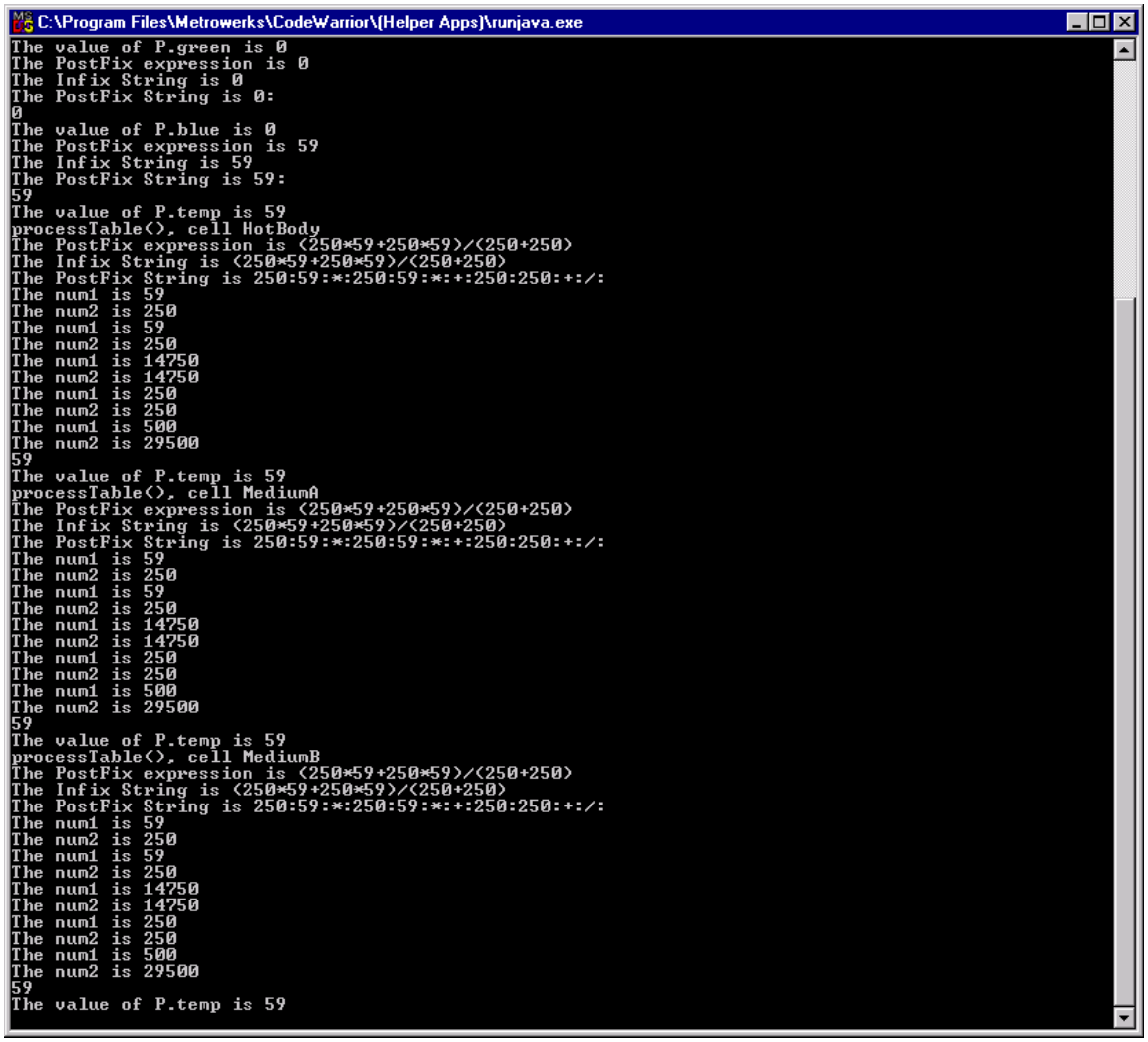
The spreadsheet is executed when the user clicks on the "Start" Button on the spreadsheet programming environment box. The Equations of the cell are executed one after another and all the cells are constantly checked for stability. The spreadsheet looks like the following when the final stable state is reached.



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9
4,0	4,1	4,2	4,3	HotBody	MediumA	MediumB	ColdBody	4,8	4,9
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9

All the bodies reach a thermal equilibrium state when the spreadsheet has finished execution.

A sample of the debugging output window which kind of traces how the program executes is shown in the next window


C:\Program Files\Metrowerks\CodeWarrior\(Helper Apps)\runjava.exe

```
The value of P.green is 0
The PostFix expression is 0
The Infix String is 0
The PostFix String is 0:
0
The value of P.blue is 0
The PostFix expression is 59
The Infix String is 59
The PostFix String is 59:
59
The value of P.temp is 59
processTable(), cell HotBody
The PostFix expression is (250*59+250*59)/(250+250)
The Infix String is (250*59+250*59)/(250+250)
The PostFix String is 250:59:*:250:59:*:+:250:250:+:/:
The num1 is 59
The num2 is 250
The num1 is 59
The num2 is 250
The num1 is 14750
The num2 is 14750
The num1 is 250
The num2 is 250
The num1 is 500
The num2 is 29500
59
The value of P.temp is 59
processTable(), cell MediumA
The PostFix expression is (250*59+250*59)/(250+250)
The Infix String is (250*59+250*59)/(250+250)
The PostFix String is 250:59:*:250:59:*:+:250:250:+:/:
The num1 is 59
The num2 is 250
The num1 is 59
The num2 is 250
The num1 is 14750
The num2 is 14750
The num1 is 250
The num2 is 250
The num1 is 500
The num2 is 29500
59
The value of P.temp is 59
processTable(), cell MediumB
The PostFix expression is (250*59+250*59)/(250+250)
The Infix String is (250*59+250*59)/(250+250)
The PostFix String is 250:59:*:250:59:*:+:250:250:+:/:
The num1 is 59
The num2 is 250
The num1 is 59
The num2 is 250
The num1 is 14750
The num2 is 14750
The num1 is 250
The num2 is 250
The num1 is 500
The num2 is 29500
59
The value of P.temp is 59
```

The temperature of the final system is 59 F.

7. Possible Future Enhancements

The spreadsheet model that I had developed can actually serve as a base for a much more powerful spreadsheet-programming model. The following are some features that could be added to make the spreadsheet a much more powerful and a comfortable environment to program in.

Control Features

Adding the conventional control statements available in almost all procedural languages will greatly enhance the power of this spreadsheet. Right now, all the cells of the spreadsheet have only "value based" dependency. If the PDE Box allows control statements such as the for, if, while and the switch statements, then it is possible to implement a "flow of control" based dependency among the cells.

Object Oriented Features

Although the current spreadsheet is inherently object oriented in nature some more conventional object oriented features could be added. For example, one of the cells could be defined as the "interface cell" which could be defined to provide some basic functionality which all the cells "cloned" from this has to implement.

Enhancing Cell Properties

The programmer could be given more flexibility in handling cells. The shape of the cell could be a variable, which could be changed. The programmer could be given an option to merge cells. Features that deal in great detail about the structural attributes of the cell could be added.

Distributed Computing Features

This project assumed that the Objects reside on the same machine. A lot more flexibility could be added if we could define a cell to be a remote object residing at a remote position on the network. That way, the spreadsheet can hide the details of the remote

Objects location on the network and the programmer can treat the remote object just like any other cell with a co-ordinate position. There might be lot of concurrency control and other issues to deal with.

Multimedia Features

At present, each cell has its "data and physical attributes" encapsulated. Audio and Video attributes could also be added to cells.

References

1. Ed Huai-hsin Chi, Joseph Konstan, Phillip Barry, Joh Riedl. A Spreadsheet Approach to Information Visualization. ACM Proceedings 1997

2. http://www.bricklin.com/visicalc.htm. Web page about visicalc from its creator.

3. http://www.cs.orst.edu/~burnett/Forms3/forms3.html. Dr.Burnett's Forms/3 homepage

4. Margaret Burnett, Herkimer J Gottfried. Graphical Definitions: Expanding Spreadsheet languages through direct Manipulation and Gestures

5. Jocelyn Paine. Model-Master: an object oriented spreadsheet front end. Presented at CALECO97 BRISTOL - UK 25-26 SEPTEMBER 1997. Web link: http://www.ifs.org.uk/~popx/mm_demo.html

Appendix

Filename:	MS Project1
Directory:	C:
Template:	C:\Program Files\Microsoft Office\Templates\Normal.dot
Title:	MS Project
Subject:	
Author:	Mahesh
Keywords:	
Comments:	
Creation Date:	06/06/00 9:42 PM
Change Number:	2
Last Saved On:	06/06/00 9:42 PM
Last Saved By:	Mahesh
Total Editing Time:	1 Minute
Last Printed On:	06/07/00 12:42 AM
As of Last Complete Printing	
Number of Pages:	45
Number of Words:	8,332 (approx.)
Number of Characters:	47,494 (approx.)