DEVELOPMENT OF A STANDALONE JAVA-BASED MOLECULAR SIMULATION ENVIRONMENT

by

Bryan C. Mihalick

February 1, 2001

A thesis submitted to the

Faculty of the Graduate School of

State University of New York at Buffalo

in partial fulfillment of the requirements for the

degree of

Master of Science

Department of Chemical Engineering

Acknowledgement

I take this opportunity to express my thanks and appreciation to my advisor, Dr. David A. Kofke for his endless patience and insight throughout the creation of this work. I would also like to extend my gratitude towards him for his financial support throughout the course of my work. I am grateful to Dr. Alexander Cartwright and Dr. T.J. Mountziaris for being on my committee, critiquing my thesis, and for suggesting improvements to its content.

Additionally, I would like to thank the members of the Kofke research group, especially Sang Kyu Kwak, Jayant Singh, and Jhumpa Adhikari for all of their help during my research.

Finally, I wish to thank all of the new friends I have made here at Buffalo for making my stay not only memorable, but enjoyable.

Table of Contents

List of Figures

Abstract

This thesis describes the creation, implementation, and documentation of a Java-based molecular simulation development environment. Its purpose is to allow for the construction and serialization of interactive graphical simulations. The environment will allow these simulations to be created without the need for user programming, although it will provide ways of extending the functionality of the simulations by providing a platform for the integration of the user's own Java classes. The structure of the environment is a collection of primary Java classes that can be pieced together in a numerous array of configurations through the use of a graphical user interface and its corresponding menus. The environment supplies researchers with a powerful means for creating and saving a wide variety of interactive simulations, and provides instructors with a highly useful visualization tool that can increase their students' understanding by displaying a specific phenomenon with graphics rather than just words.

1. Introduction

As the performance of computers continues to grow, molecular simulation has become an attractive means of predicting and studying behavior of real materials. By using simulations instead of actual lab experiments, researchers can not only decrease the amount of time necessary to complete a project, but they also can decrease the cost of such research by replacing large laboratory equipment and expensive materials with relatively inexpensive computers. Additionally, simulation makes it possible to perform "experiments" on theoretical configurations that would be impossible in the material world. Further by implementing a graphical interface, professors can use simulations as visual teaching tools that go beyond the reach of blackboard drawings in helping students understand complex phenomena. With all of these advantages, simulation does however have its drawbacks. First of all, commercial software development packages are very expensive especially when considering the lack of portability created by licensing agreements. On top of this high cost, most packages are not designed for the user's specific project or need, making specific applications more difficult to accomplish. Secondly, commercial molecular simulation is based on older software technologies, and consequently is not very interactive. For example, it does not allow computational steering by the researcher. With steering, simulations can be adjusted on the fly, which can avoid iterations at sub-optimal conditions. Third, most home made (non-commercial) simulations simply print out massive amounts of numbers that make it difficult to see

patterns or trends in the data, particularly at run time. These trends would be made more visible by being displayed in a GUI environment.

To address these issues, the goal of this research is to design, develop, and test a standalone Java based molecular simulation environment suitable for a wide range of applications. This environment will be named Etomica. Etomica is based on an open-source molecular simulation API[1], and is thus designed at its core to be fully extensible. Etomica will be low cost and fully portable since no commercial components will be necessary. Additionally by implementing the Java Swing API, Etomica will be a fully functioning GUI. This will provide almost immediate feedback to the user instead of requiring an extended analysis of the data. This also will make Etomica a powerful teaching tool by providing students with an interactive picture rather than just words. With the enhancements that Etomica will provide to the simulation development process, the presentability of the results that are generated will be enhanced whether these results are for research facilities or student lectures.

1.1. What is a Molecular Simulation?

The dictionary definition of a simulation is "an imitation of a system or process by means of another." But what is a Molecular Simulation? A molecular simulation is a computer experiment based on a molecular model. Elements of a molecular simulation generally consist of objects that represent molecules coupled

[1] API, application programming interface
An API is a series of functions that programs can use to make the operating system do their tedious work. Using the Molecular Simulation API, for example, the Etomica environment can create molecules and displays by passing a single instruction to the operating system.

with potentials that describe the interactions (e.g. collisions) between the molecules. Molecular simulations provide methods of isolating preferred configurations or states (which might be impossible in the material world), and further, measuring the physical conditions of these states. Therefore, any programming tool intended as a molecular simulation package must incorporate objects that sufficiently represent the aforementioned participants of a molecular simulation. This includes, molecule objects that describe all the properties of molecules and atoms (diameter, mass, shape), potential objects that describe these molecules' interactions, environment objects that spell out the space occupied by and the phase containing these molecules, meter objects for measuring physical conditions of the system, and finally display objects for drawing graphical pictures of the system or displaying results from the meter objects.

Examples of molecular simulations are Molecular Dynamics (MD) and Monte Carlo (MC). Molecular Dynamics is a deterministic method that integrates the equations of motion to examine the evolution of a system over a period of time. Contrary to this, Monte Carlo simulations are stochastic processes that do not have an element of time. Instead they use ensemble averaging to generate a large number of randomly selected configurations of a system. In both MD and MC simulations, averages are taken over the ensemble of configurations, and these can be used to "measure" the physical properties of the model system.

1.2. Advantages and Disadvantages of Molecular Simulation

Now with a definition of Molecular Simulation in place, the next big question is, "What benefits do they offer and what are their limitations?"

1.2.1. Advantages

First of all, simulations are relatively inexpensive compared to normal lab instrumentation. Generally, scientific experiments require expensive, experiment-specific instrumentation to perform the most precise calculations as possible. In contrast with this, molecular simulations require only the use of computers that, in today's world, are inexpensive. Granted, software can become increasingly a cost burden, but even so computers are still much more cost effective.

Another benefit of molecular simulation is the ability to compile large amounts of data in short amounts of time. Here again, credit is due to the advancements of the computer to unheard of speeds and computational power. Now a simulation can accomplish in hours, if not minutes, what a research student or lab technician might have taken months, or even years, to finish.

Finally, molecular simulations open doors to researchers to test chemical systems unthinkable in the material world. Potentially hazardous or toxic materials can be experimented without harm or worry to the environment. Even further, uncontrollable reactions, such as nuclear bombs, can be simulated without destroying land or putting lives in danger. Even if the outcomes would normally be harmful or catastrophic, molecular simulation truly provides an appealing medium for conducting large-scale novel experiments effectively allowing researchers to extend the cutting edge of technology.

1.2.2. Disadvantages

Although molecular simulation is powerful, it does have its drawbacks. First, since simulations are based on a molecular model, it is necessary to have a model capable of accurately describing a system. If a sufficiently accurate model is not available, simulation is not a viable solution. Of course the results generated from a faulty model are inherently faulty themselves. Simulation results taken from such a model may be about as good as guessing what the outcome will be.

A second drawback is that simulations take an extreme amount of computational power to model even the simplest of systems. The computer industry has taken tremendous strides in extending the limit of simulations by increasing the speed and memory of computers, but even so, large compounds, such as proteins, still can only be run for short periods of real time. Even with this storage capacity, there are too many computations per iteration; it can take even the fastest machine days to complete. Presently, simulations are limited to systems and phenomena that extend over micron lengths and nanosecond time scales. Larger scales can be reached, but at the cost of a greater degree of approximation in the calculation.

1.3. Layout of Thesis

In chapter 2, Platform Specifications, the platform used for the Etomica environment will be discussed. These include the language used to create the

environment along with the reasons and benefits for using the language. In chapter 3, Molecular Simulation API, the API will be presented. The layout of the API is shown along with the features and abilities that define the API. Chapter 4, Etomica Molecular Simulation Environment, contains instructions for using the Etomica environment. Screen shots are included to provide the user with an idea of how the environment will look in certain situations. Accompanying these screen shots are descriptions to familiarize the user with the options that are available for creating and modifying simulations. Chapter 5, Features of the Etomica Environment, lists all of the features that the Etomica environment provides the user for making the development and running of simulations quicker and easier. Chapter 6, Errors and Error Handling, presents the user with some of the known errors and limitations of the Etomica environment so that they are aware of what can and cannot be accomplished using the environment. Chapter 7, Future Work, lays out some of the things that are not currently available but are soon to be developed or implemented into the Etomica environment. Finally, Chapter 8 is an appendix that lists the tools that were used for developing the Etomica environment along with the naming structure that was developed to make class names consistent for easy searching and finding.

2. Platform Specifications

2.1. Why Java?

When developing new software for any application, it is paramount to put as much time and effort as possible into the design of the package. This includes

determining what language gives the most opportunity to achieve the functionality that the developer, and more importantly, the user require. This functionality consists of the compatibility of the language with the hardware that is already in place, the flexibility the language provides the developer when designing the environment, and the power the language offers the developer to allow the easiest implementation of the most difficult features.

2.1.1. Benefits of Java

By using Java, an army of classes is available due to the extensive depth and breadth of the already existing Java APIs. These provide developers with a broad spectrum of tools that accomplish the common tasks of software development. These tools permit them to spend their time writing the classes and methods specific to their needs.

Java also features cross-platform compatibility. Therefore, regardless of the hardware they are using, any interested user can implement software created with Java. This makes distribution and integration much easier.

By providing a means of mass viewing any simulation, namely Applet technology, Java increases this distribution even more. Applets are components that can be added to a webpage for easy viewing across the Internet by means of a web browser. In this way, other scientists or students will be able to utilize or conduct simulations even if they are on the other side of the globe.

Now that there are ways of distributing simulations, how can users save the results or the specific conditions of a simulation? Users need a method of persistence

that will save the components of a simulation in their current state or condition. Java accomplishes this with Serialization.

Serialization provides the essential capability of storing and retrieving Java objects from applications, and therefore the simulation components, created by the user, of a simulation environment. The key to storing and retrieving objects is representing the state of objects in a serialized form sufficient to reconstruct the object(s). Java accomplishes this by creating a serialized byte stream (saved as a *.ser file) that contains not only all the pertinent information from the object that is being serialized, but also from every object that is reachable or referred to by that object. This method of persistence insures the existence of the object in the future and rids the user of needing to recreate any previously created simulations.

The last benefit, but certainly not least, is Java's ability to determine, and more importantly, alter the properties of an object, or in this case, a simulation element. Java accomplishes this by way of the JavaBeans API and its Introspection technology. Introspection allows users to not only see what properties a simulation component is defined by, but also gives users the power to change these properties to alter the current state of an object. What this means to the user of a simulation environment is that they now will have the ability to change the size, color, etc., of a simulation component effectively changing the current conditions of a system. Even better, these conditions are changeable while a simulation is running, so simulations will have the desired property of being steerable. For the simulation environment this means that as members of the group develop new classes, they will automatically be included in the selection lists of the simulation environment (assuming their

respective class names follow the predetermined naming structure). Ultimately, this means the environment will grow with its group of developers.

All of the benefits expressed above show the power of the Java language and its usefulness for this project. For this reason, Java was the language of choice for the Etomica Molecular Simulation environment.

2.2. Which Java Release?

Now that a language is determined, the next step is to decide which release of Java is in the best interest of the environment. A good choice of compiler needs to be made so that the environment will be compatible with the accompanying software a user may implement.

After weighing the options, a decision was made to use the Java 1.2.2 compiler from the Java Runtime Environment 1.3. Although this compiler would allow for all current Java classes to be utilized, it would make some compatibility issues arise when using web browsers to display applets. Web browsers are always behind the technologies they are trying to display and for this reason some components will not be viewable in certain browsers. With this mind, one must also remember that eventually the web browsers will evolve to incorporate the new technologies and eventually these applets will be fully functioning in all web browsers. Due to this fact, using the most up-to-date Java compiler was deemed the best option.

2.3. Compatibility Issues

As with any software implementation, some the main problems usually become apparent when compatibility issues are considered. The Etomica environment and the files it produces are no exception. Some of these problems are minimized by Java's cross platform compatibility. Whether it is Windows NT, Windows 98, or Unix, this cross platform compatibility allows the Etomica environment to run on any of the standard operating systems. Problems do arise however when applets that are generated by the environment are attempted to be implemented on a website. These exist due to the inability of web browser manufacturers, such as Microsoft (Internet Explorer) or Netscape (Navigator), to keep up with cutting edge developments by the makers of Java. As a result, plug-ins are necessary in some cases to provide the additional functionality to the browser to allow for the applet to be displayed and run in its entirety. And unfortunately in some cases, certain applets are not viewable even with the latest plug-ins. Eventually web browsers will catch up to the capabilities of the Etomica environment's output, so this problem is minimal and constantly diminishing. Therefore, the compatibility of the Etomica environment is very high, and will only increase with time.

3. Molecular Simulation API

The Molecular Simulation API is a collection of classes capable of representing all the components of a simulation in an object-oriented manner. Its purpose is to empower users with the tools necessary for simulating a system of molecules under user-specified conditions, along with the features required for viewing the molecular configuration and measuring the properties of this system.

The API accomplishes this by providing a collection of elements that were designed to work efficiently with one another. Further, the API is extensible, allowing users to enhance its functionality by introducing appropriate elements that adhere to the standards.

3.1. Molecular Simulation API Classes

The following lists all of the major components of the Molecular Simulation API along with their implementation and functionality. Classes are written that represent species of molecules, the space that they occupy, the phase that they are in (gas, liquid, solid), and the potentials or molecular interactions they have between one another. This API also has classes that perform MD or MC sampling, control the beginning and end of the simulation, and meter the thermophysical properties of the system. The API effectively provides a complete framework for developing and running an experiment on a molecular model of a chemical system.

3.2. Simulation

The Simulation class of the Molecular Simulation API is the main class that organizes the elements of a molecular simulation. The simulation class contains a set of Linked Lists (one for each type of simulation component) that hold a handle to every component object currently contained within the simulation. This allows for easy iteration through all the objects of a given component type. Register and unregister methods are also present in the simulation class to allow for addition of components to the linked lists and therefore to the simulation itself.

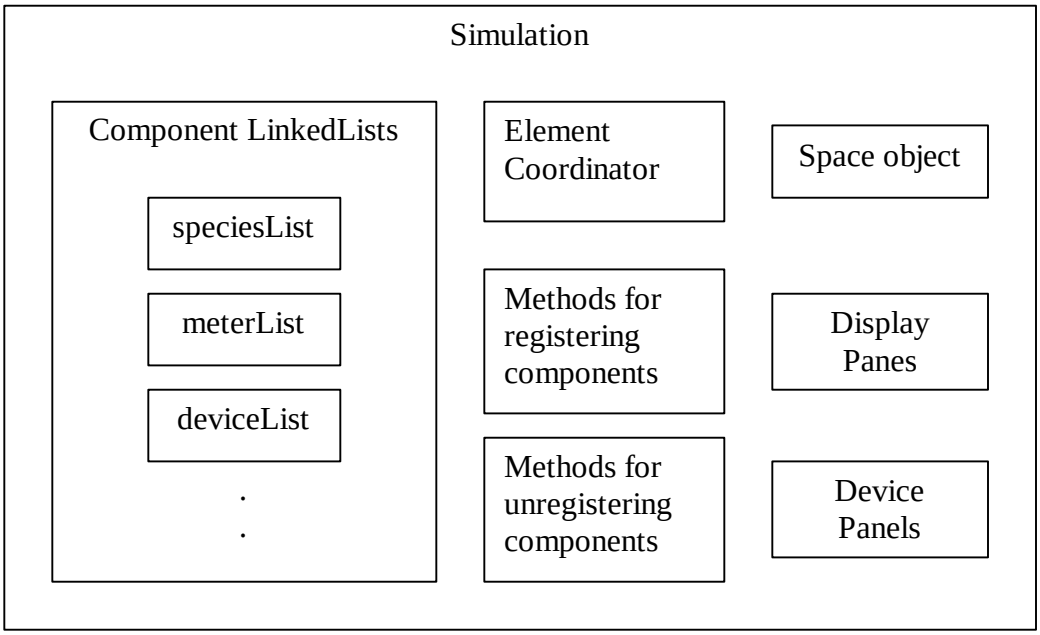


Figure 3.1. Layout of Simulation Class

Currently, simulation provides a static 'instance' object that acts as a container of all the components of the simulation. Eventually, this static instance will be phased out so that multiple simulations can be run simultaneously. Within this instance is a static Space object that is referenced by all the simulation components whenever spatial values are needed. Also included in this instance is a graphics object that can be used to display the simulation in a GUI, as well as an object that specifies the unit system used as a default for all I/O to displays and meters.

Finally, the simulation instance contains the element coordinator object. This object is responsible for linking all the components in the simulation together just

before runtime. As currently implemented, this design prohibits addition of new components after beginning the simulation. Future releases will want to implement a different approach that will allow these additions.

3.3. Space

Space provides some general physical framework of a simulation. This includes the ability to create vectors, tensors, boundaries, and coordinates. Any subclass of Space, therefore must define a vector, tensor, and boundary appropriate to that Space, based on its specific dimension (i.e. 1-D, 2-D, 3-D).

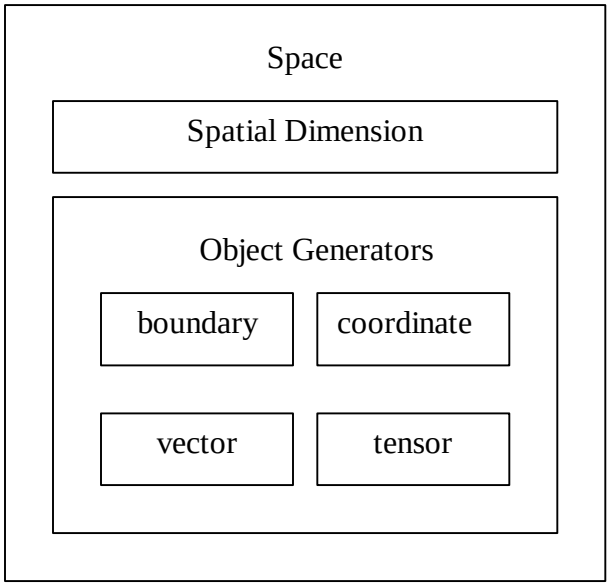


Figure 3.2. Layout of Space Class

3.3.1. Vector

Space defines a vector sufficient for identifying a point in space. Utility methods are written for determining the length and dimension of a vector as well as for retrieving the individual component values of the vector. Manipulation methods

also are provided for adding (PE, plus-equals), subtracting (ME, minus-equals), multiplying (TE, times-equals), and dividing (DE, divide-equals) vectors. Finally, evaluation methods are present for finding the square (squared), the dot product (dot), and the cross product (cross) of a vector.

3.3.2. Tensor

Beyond vectors, the space class also provides a tensor class capable of representing tensor quantities. Constructors exist for making tensors by multiplying two vectors together (in 3-D), or simply by giving a list of nine values corresponding to each component of a tensor. Like vectors, tensors provide utility methods for retrieving and altering its individual component values. In addition, manipulation methods exist for adding (PE) and multiplying (TE) tensors.

3.3.3. Boundary

Space also has methods for creating the boundary conditions of a phase. The phase is responsible for calling these methods to create and hold an instance of the boundary that is needed. This is done by the following code in the constructor of the Phase:

```
setBoundary(Simulation.space.makeBoundary());
```

This call works as follows. First, the makeBoundary method of the space object held in the Simulation object is called. Calling this method without specifying a boundary causes the boundary to default to PERIODIC_SQUARE that implements Periodic Boundary Conditions (PBC). Secondly, the setBoundary method of Phase is

called to define the simulation boundary as this PBC implementation created by Space.

Calling makeBoundary and supplying a boundary parameter allows creation of other types of boundaries. Supplying the parameter Boundary.NONE creates a boundary object with no outer limits that allows all occupants of the Phase to move around freely. Creating a hard boundary condition (Boundary.HARD), effectively inserts walls at the Phase edge with which the occupants of the Phase interact with a hard sphere potential. Implementing periodic boundary conditions (PERIODIC_SQUARE), which is the default used in the example above, creates a Phase with boundaries that account for occupants leaving the Phase by translating their vector coordinate to the opposite side of the Phase so that they enter in the opposite side. Lastly, SLIDING_BRICK boundaries are used for implementing flow fields in viscous fluids by having periodic images move across each other.

All Boundary objects have nearestImage and centralImage methods. NearestImage is used to determine the distance from one molecule to the nearest image of another molecule in the surrounding periodic images. This is needed for determining nearest-neighbor interactions. CentralImage takes a vector or coordinate as its parameter and moves the Molecule at that vector or coordinate position to the central image. Boundary objects of type NONE, HARD, and PERIODIC_SQUARE provide volume and inflate methods for determining the volume of the space inside the boundary and for inflating the volume of the space by a given amount. They also have dimension and makePotentialField methods for determining the dimensions of a Space and for making a Potential between the Boundary and all Molecules that may

interact with it. SLIDING_BRICK Boundary objects hold a ChronoMeter object to determine how far periodic images will have moved over a period of time. They also have methods for accessing (getShearRate) and setting (setShearRate) the shear rate, and for returning the origin of the periodic image (imageOrigin).

3.3.4. Coordinate

Coordinate is responsible for collecting all Vectors (position and momentum) that describe a point in space. All subclasses of Coordinate must have parent, position, momentum, and kineticEnergy methods. Parent returns the occupant (Atom, Molecule, etc.) that has this object as its coordinate. The position and momentum methods respectively return the Cartesian coordinate vector and the momentum vector describing an occupant of a Space. Finally, kineticEnergy evaluates and returns the kinetic energy associated with a Space occupant.

Coordinates with a specific Orientation can be created for determining orientation specific interactions such as those found with polar Molecules. These objects have an angle method that returns the angle of rotation from a fixed point as well as rotateBy methods for rotating the object by a certain angle. A rotation matrix, A, operates on the components of a vector in the space-fixed frame to yield the components in the body-fixed frame. ConvertToSpaceFrame and convertToBodyFrame methods provide a means for moving between the two prospectives.

3.3.5. Continuum Space

The space class currently has 4 subclasses that all represent a continuum. These are 1-D, 2-D, 2-D Cell, and 3-D spaces. All continuum space types have their respective occupants positioned based on the Configuration object residing in Phase. The default Configuration is Sequential. In this Configuration, occupants are placed on a lattice whether it is a 1-D, 2-D, or 3-D lattice configuration. 2-D Cell differs from this by having its space divided into cells of equal size. Neighbor lists are created with the occupants of the same cell being members of the same neighbor list. In some cases, this implementation of Space can be highly efficient. For example, by minimizing the amount of Space observed when determining collisions, 2-D Cell increases the collision determination efficiency of the system effectively alleviating the computational burden.

3.3.6. Non-Continuum Spaces

Currently the only Non-Continuum Space class that has been developed is the Lattice. This can be used, for example, to model a magnet by having polar occupants capable of changing orientation (flipping up or down) at each lattice site. Implementations of the lattice could also yield simulations characterized by occupants moving from site to site on the lattice instead of being free to float through space.

3.4. Controller

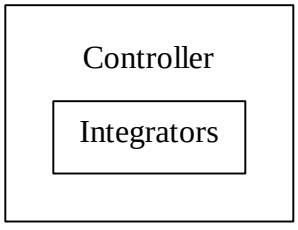


Figure 3.3. Layout of Controller Class

Controller is the overseer of all actions performed by the integrators in a simulation. It is capable of starting and stopping any and all integrators whenever necessary. Correspondingly, the controller has methods for pausing and continuing the execution of the integrators. Generally, these are called when the graphical button object is clicked which performs a list of tasks. If the button is clicked for the first time, the Controller calls the start method that performs the following function calls:

```
Simulation.elementCoordinator.go();
runner = new Thread(this);
runner.start();
```

This tells the elementCoordinator of the Simulation object to link all of the components together. By doing so, all of the Integrators in the simulation are added to the LinkedList of Integrators held in the Controller. The code above then creates a thread for the Controller to run on and starts the Controller on this thread. Once started, the Controller calls its own run method:

```
for(java.util.Iterator iter=integrators.iterator(); iter.hasNext();
)
       ((Integrator)iter.next()).start();
```

This code loops through the LinkedList of Integrators and starts each of them on its own thread. If the Controller was already running, the following calls are made:

```
for(java.util.Iterator iter=integrators.iterator(); iter.hasNext();
)
       ((Integrator)iter.next()).pause();
```

By looping through the Integrators and calling each of their pause methods, this effectively pauses the execution of the simulation. Finally if the simulation was already in a paused state when the button was pressed, the following is called:

```
for(java.util.Iterator iter=integrators.iterator(); iter.hasNext();
)
       ((Integrator)iter.next()).unPause();
```

This loops through the LinkedList of Integrators and calls their respective unPause methods effectively continuing the simulation.

3.4.1. Controller Implementations

A Controller can be implemented in different ways. The first method is by implementing a button device that is displayed on the screen. This is the default implementation and is the one discussed earlier. Once pressed, this button begins or halts the activities of the integrators. A second implementation of the controller allows users to define a fixed number of cycles that the integrators can execute. Once these cycles are completed, the controller interrupts the progress of the integrators and stops them from continuing. Another implementation allows the integrators to run several simulations over a series of states. After the required simulations are completed, the controller once again steps in and blocks the integrators from continuing. Other possible Controllers could take the values obtained from its list of Integrators and perform even more computation on the data, for example by thermodynamic integration.

3.5. Integrator

The integrator class generates new configurations of the occupants of the phase by moving atoms. For example, moves can consist of adding or deleting

molecules, translating the position of a molecule, or moving molecules based on their present velocity and according to the equations of motion.

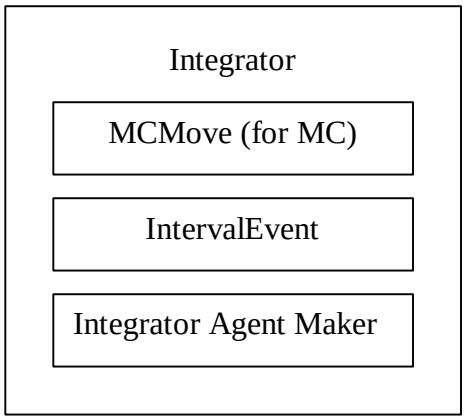


Figure 3.4. Layout of Integrator Class

Moves occur every time a call is made to the abstract doStep method of the Integrator. Each Integrator provides a different implementation of this doStep method depending on the type of move that it performs.

Every atom has an Integrator Agent that holds the information regarding its movement. The content of this information differs from Integrator to Integrator. By using this information, Integrators can store atom-specific information needed to accomplish its move.

The Integrator continues to make moves until a certain number of these have transpired. At this point, the Integrator completes an "interval" and fires an intervalEvent to signal the end of this iteration. Listeners of the Integrator respond by updating themselves to reflect the changes to the configuration of Molecules in the Phase. Integrator continues making iterations and firing intervalEvents until the

Controller ends its progression or it completes the pre-designated number of doStep calls.

3.5.1. Integrator Agent

Integrator Agents are the informants of the Integrator. Every Atom of every Phase that is associated with an Integrator gets an Agent object supplied to it by the respective Integrator. This Agent holds the Atom- and Integrator-specific information that allows the Integrator to complete its configuration changes.

Depending on the type of Integrator, the Agent can consist of a variety of information. For example, a Hard Potential Integrator used for Molecular Dynamics simulations includes the time until the atom's next collision, the object the atom will collide with next (collisionPartner), the potential that describes the interaction between the colliding atoms, and so on. Alternatively, the Velocity Verlet Integrator used for Soft Potentials in a Molecular Dynamics simulation holds the force vector that would be applied continuously to all Atoms in a Phase.

Armed with this information, the Integrator loops through all atoms of a phase and based on the information offered by the Agent makes the move that is necessary for that atom.

3.5.2. Interval Event

IntervalEvents let objects (i.e. meters, displays) that are listening to the Integrator know that a new configuration has been reached. This is done to notify

listeners that it is time to update based on the new configuration. For example, a meter can recalculate the current temperature or energy, or a display can show what the new configuration looks like. As mentioned previously, IntervalEvents do not occur after every configuration change the Integrator makes to the phase, but rather only after a sufficient, user-specified number of moves have been made.

3.5.3. Molecular Dynamics

Integrators used in molecular simulation vary and can be vastly different, although the general ideas discussed above remain a standard. Two types of Integrators currently found in the Molecular Simulation API are those used for Molecular Dynamics simulations and those used for Monte Carlo simulations.

Integrators used for Molecular Dynamics simulations determine new configurations by integrating the equations of motion and determining the new coordinate positions of the occupants of the Phase based on the results of these integrations.

In the case of Hard Potentials (see Section 3.1.7.), since collisions between occupants of the Phase usually occur before the end of the time step (editable by user but has a default setting of 0.05 ps), the integrator always predetermines the time of the next collision and moves all of the occupants only a distance equivalent to that which could be traveled during that time. These movements are done with the help of an iterator that allows the integrator to walk through all the occupants contained in the Phase. At the point of the collision, the integrator accounts for the collision by altering the colliding occupants momenta as calculated by the equations of motion

and based upon the potential that exists between the colliders whether it is HardSphere, SquareWell, etc. The next collision is then determined and all occupants are again moved the distance corresponding to the elapsed time before that collision. This process continues until the Controller overseeing the execution of the integrator thread halts its progress.

When dealing with Soft Potentials, the Integrator uses iterators to loop through the Atoms of a Phase. While doing so, the Integrator adds the force vector created by the PotentialField and Potentials to the vector currently associated with the Atom's Agent. The resultant of this addition yields the force vector that determines the new coordinate position of the Atom. The Integrator then moves each Atom to its new coordinate position. Once again, the Integrator continues this process until the Controller halts its progress.

3.5.4. Monte Carlo

Unlike Molecular Dynamics simulations, integrators used for Monte Carlo simulations have no element of time. Therefore, moving occupants through a time step is not appropriate. Rather, Monte Carlo integrators are stochastic (moves are selected at random) and have a pool of available moves that are specified to them by objects of type MCMove.

The type of moves supplied by the MCMove object define the ensemble of the simulation. For instance, by using the MCMoveInsertDelete object, a Grand Canonical ensemble is created. The MCMoveInsertDelete object randomly inserts and deletes molecules from Phase. Similarly, by implementing the MCMoveAtom

object that displaces Atoms a random distance between upper and lower limits, a Canonical or NVT ensemble is implemented. In this manner and by implementing other types of MCMoves, Monte Carlo simulations having different ensembles can be created.

The Monte Carlo Integrators (IntegratorMC and its subclasses) move with a probability specified by the MCMove objects, and periodically check the acceptance rate. Based on this rate, the step size of the moves is adjusted to approach a target acceptance rate (e.g. 50%).

3.5.5. MCMove

The Molecular simulation API currently provides MCMove objects for volume or molecule insertion or deletion, volume or molecule exchanges in the Phase, molecule or atom movement, or molecule rotation. Subclassing the MCMove class and supplying a thisTrial method that defines the mechanics of the move gives users a way to implement other types of moves. Although the user determines the general type of MCMove object to implement (e.g. MCMoveInsertDelete, MCMoveVolumeExchange, etc.), the random number generator contained in that object determines the precise move (Insert or Delete a Molecule, Add or Remove Volume) caused by the object from trial to trial.

Once a controller engages an integrator, the integrator performs random moves based on the current type of the MCMove object, as well as the current value of the random number associated with that object. Once a trial move is completed, the Metropolis criterion is applied to determine the acceptance or rejection of a new

configuration. If the Metropolis criterion accepts this move, the next move is tried. If it rejects the move, the previous configuration is restored and another move is tried. During this process, the ratio of the number of acceptances versus the number of trial moves is constantly calculated. The goal is usually to keep this ratio near 0.5. If this ratio is too high (accepting new configurations too often), the moves being attempted are not altering the configuration sufficiently and therefore the magnitude of the move (whether the size of the volume inserted or deleted, the displacement of the Atom, etc.) is increased. If this ratio is too low (rejecting new configurations too often), the moves being attempted are altering the configuration excessively and therefore the magnitude of the move is decreased.

3.6. Phase

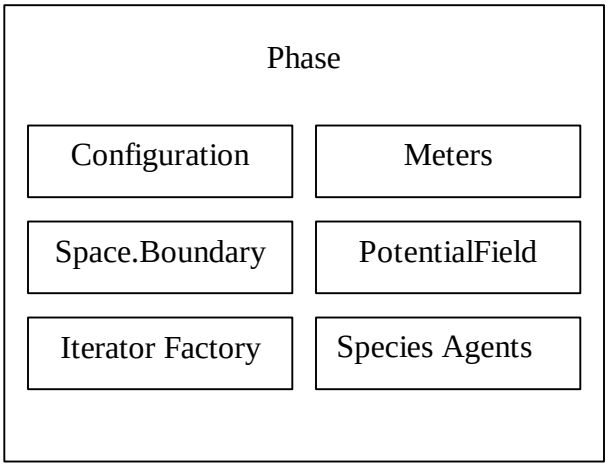


Figure 3.5. Layout of Phase Class

A Phase component is responsible for grouping atoms or molecules that interact with one another. Phase holds a configuration object that lays out the

arrangement of the molecules. Further, Phase holds a boundary object that defines the condition at the edge of the Phase. In addition, to allow for easy reference to the molecules that reside in it, a Phase has an IteratorFactory that creates iterator objects that aid in looping through the molecules. Finally, Phase contains the Meter objects that measure all the physical properties of a given Phase, as well as the one-body PotentialField objects used to add forces produced by gravitational, electric, or magnetic fields.

3.6.1. Species Agent

A Species Agent assists a Phase by doing some of the tedious work of a Phase. This includes adding, deleting, and counting molecules. Species Agents also provide iterators for looping through molecules of a Species in a Phase. A Species Agent is created by every Species, and an instance from each is placed in every Phase.

3.6.2. Configuration

Configuration determines the initial configuration of a Phase. By way of the Species Agent, Configuration assigns coordinates to the molecules in a Phase. The default configuration of Phase is sequential. The sequential configuration aligns all current occupants of a phase in a square lattice structure, placing each atom in sequence on the lattice. This Configuration is instantiated by the Phase constructor and added to the Phase with the call:

```
add(new ConfigurationSequential());
```

When Configuration comes across Species of type Wall, these are ignored and not included in the lattice structure.

3.6.3. Boundary

Boundary objects are created by Space (refer to 3.3.3). They define the outer limits of a Phase and therefore the farthest distances a molecule can reach. A handle to the boundary is located in the Phase. Currently, Boundary can be of type NONE, PERIODIC_SQUARE, HARD, or SLIDING_BRICK. The default Boundary is set to PERIODIC_SQUARE and is instantiated and set in the Phase constructor by the call:

```
setBoundary(Simulation.space.makeBoundary())
```

This utilizes the makeBoundary method of the Space object to create the Boundary and then uses the Phase's own setBoundary method for saving a handle to the new Boundary.

3.6.4. Iterator Factory

IteratorFactory creates single-atom and pair-atom iterators. Every Phase has at least one iterator created by the IteratorFactory for use in looping through the atoms in that Phase. Iterators can be made by the factory for searching up or down regular atom lists or neighbor lists.

3.6.5. Meters

A list of Meter objects is held within the Phase. These objects measure and average physical properties of the atoms contained in a Phase. These Meters update

their averages every time the Integrator notifies them by firing an IntervalEvent. Using the collection of Display objects, the current value of any Meter held within Phase can be shown. By default, an EnergyMeter is created by the Phase to allow for determination of the potential and kinetic energy of the atoms in the Phase.

3.6.6. PotentialField

Unlike two-body potentials, one-body PotentialField objects are held in the Phase. These potentials can account for the effects of gravitational, magnetic, and electric fields on the configuration of atoms in a Phase. Currently only a gravity field (PotentialFieldGravity) one-body potential is implemented in the Molecular Simulation API, but by subclassing PotentialField many other one-body potentials can be created.

3.7. Species

A Species is a collection of identically formed Molecules. Molecules can consist of one or more Atoms. Species holds the AtomType of the atoms in the Species' Molecules, as well as the Configuration of the atoms in each of these Molecules. When they are instantiated, the Simulation class brands each Species with a specific SpeciesIndex. At this time, the Species itself makes a Species Agent that is placed in every Phase. The index is used to associate each Species with its respective Potentials, and the Agent is created to assist the Phase in manipulating and accounting for Molecules in that Phase. Finally, each Species has a Reservoir of Molecules so that as Molecules are added and deleted they are not created and destroyed, but rather

saved for later use or recalled from the inventory. This rids Species of the computationally expensive task of creating new Molecules. In most cases, Reservoir is not used that extensively, but in Grand Canonical Monte Carlo simulations where Molecules are randomly inserted and deleted it is an invaluable tool.

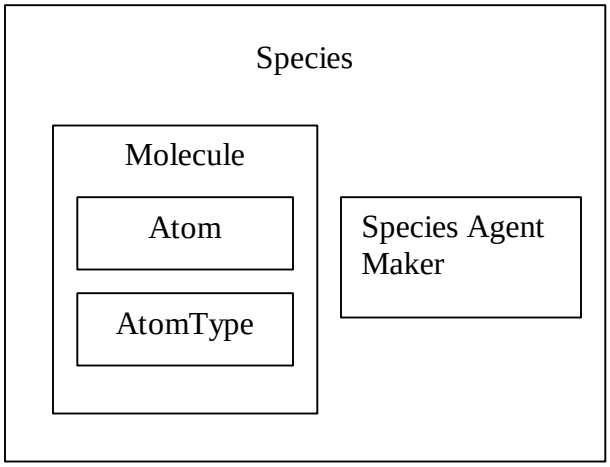


Figure 3.6. Layout of Species Class

3.7.1. Atom

Atom defines the properties of one physical Atom. It has a corresponding coordinate position given to it by Space once the Atom is instantiated. This vector coordinate is operated on to determine all kinetics and dynamics quantities related to the Atom. Atoms are branded with an index number so that each can be distinguished within a Molecule. Every Atom has an AtomType that determines how the Atom will be drawn by the Display, and to permit other features to be associated with it (e.g. size). Atoms have Integrator Agents associated with them for storage of any information pertinent to the Integrator.

3.7.2. AtomType

AtomType is the object that holds the parameters of an Atom. These include mass and color. By subclassing AtomType, the general features of an Atom can be specified. Current subclasses include Disk, Rod, Sphere, Well and Wall. Rods (1-D), Spheres (3-D), and Wells (any Dimension) are subclasses of Disks (2-D), and all of these AtomTypes specify a mass, color, and diameter. Other types can have an associated orientation. This can be used, for example, by an orientational potential that only causes a reaction when the Atom is in a certain orientation. Wells have a field that defines the diameter of the well. Finally, Walls are just what they sound like, an object that can be part of a Species that looks and acts as a wall. Currently these work only in 2-D space, but subclasses could be written to incorporate Walls in other spaces. Walls are defined by there thickness, length, and angle (defines orientation with respect to x-axis). Walls can be given a temperature that can alter the temperature of Molecules that collide with them or be set as adiabatic (the default).

3.7.3. Molecule

Molecules are collections of Atoms. Molecules differ from one another by the number of Atoms they comprise and the AtomType of those Atoms. Usually Molecules are made up of Atoms with the same AtomType (this is the case of Molecules created by a Species object), but can have Atoms of different AtomTypes

by subclassing Molecule directly and supplying an array of AtomTypes to the constructor.

Molecules provide methods for altering their respective coordinate position. There are methods for altering velocity by acceleration or deceleration and changing position by direct translation or by a specified displacement. Molecule also provides Iterators looping through Atoms of a given Molecule. A Container interface is present that gives Reservoirs and Phases the ability to contain any number of Molecules. Finally, a Configuration object is stored inside of Molecule that defines the layout of the Atoms of a Molecule.

3.7.4. Species Defaults

Default values of a newly created Species of Molecules are located in the Default class and are as follows. The default number of molecules (MOLECULE_COUNT) is twenty. This number can be changed at runtime to increase or decrease the species size. The default mass (ATOM_MASS) is 40.0 daltons, the default size (ATOM_SIZE) is 3.0 Angstroms, and the default color (ATOM_COLOR) is black. All of these can be changed at any time during the simulation, but should be adjusted at the outset if defaults are to be applied when objects are instantiated.

3.8. Potential

The potential class defines the interactions between Atoms. A Potential can have the property of being Hard, Soft, or both.

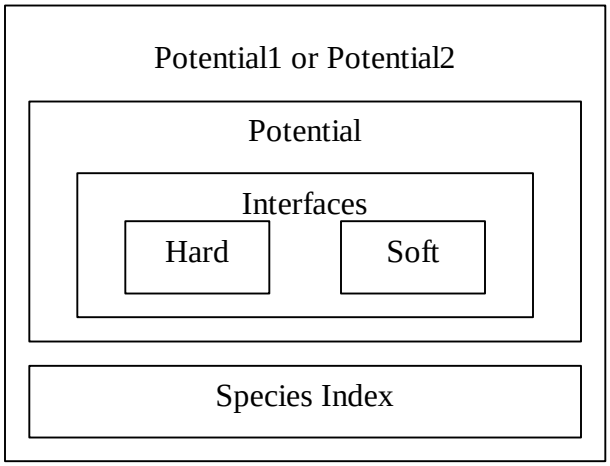


Figure 3.7. Layout of the Potential Classes

Hard potentials describe impulsive interactions between Atoms. Energy curves of this type of interaction have discontinuities. Atoms engaging in this type of interaction have definite collision times, which occur when the collisionDiameter of the two Atoms is equal to their distance apart from each other. Due to this collisionTime, Potentials interfacing the Hard Potential must provide a collisionTime method (for acquiring this time) and a bump method (for implementing the collision dynamics at that point in time).

Soft potentials are non-impulsive interactions between Atoms. Energy curves of this type of interaction are smooth with no discontinuities. Atoms engaging in this type of interaction do not have a definite collision time, but rather are constantly subjected to the force caused by the Potential. These potentials provide a force method to acquire the effect of this force.

Potentials can be grouped into a subclass of either Potential1 or Potential2 depending on what type of interactions they are describing. However, all Potentials of type Potential1 or Potential2 must provide a getPotential method that when given two Atoms returns the corresponding Potential between them. A potential exists between two objects if a Potential object exists with the corresponding speciesIndex value(s) of the molecule(s) or wall(s). This interaction would follow the properties of the corresponding type of Potential between the objects (i.e. HardSphere, SquareWell, DiskWall, etc.).

A few examples of Potentials are PotentialHardDisk, PotentialLJ, and PotentialSquareWell. PotentialHardDisk is the basic Hard-Sphere Potential characterized by infinite energy when Atoms overlap and zero otherwise. Potential LJ is the Lennard-Jones Potential that follows the form u(r) = $4*e*\left[\left(\frac{s}{r}\right)^{12}-\left(\frac{s}{r}\right)^{6}\right]$ where epsilon (ε) describes the strength of the pair interaction, and sigma (σ) is the atom size parameter. Finally, PotentialSquareWell is the Square-Well Potential which is characterized by infinite energy when Atoms overlap, -ε if Atom distance is less than the wellDiameter[2] and not overlapping, and is zero otherwise.

3.8.1. Potential 1

Potential1 collects all the interatomic potentials operating between the atoms of a single molecule. These are generally used for tethering Atoms of a given

Molecule together. Otherwise, Atoms would fly apart from each other rather than staying together in a chain. In Molecular Simulation API terms, the Potential1 object, P1TetherHardDisk, accomplishes this by creating and collecting PotentialTether Potentials between every Atom-Atom pair of a Molecule. Since it subclasses Potential1, P1TetherHardDisk must and does provide a getPotential method that returns the Potential, which in this case is PotentialTether, between two given Atoms of a single Molecule.

3.8.2 Potential 2

Potential2 collects all the interatomic potentials operating between the atoms of two molecules. Similar to Potential1, Potential2 creates and collects Atom-Atom Potentials. However these Potentials are now between two different Molecules rather than the same Molecule. These two Molecules can be of the same Species or of differing Species. Potential2 holds the SpeciesIndex numbers of the two interacting Species in either case. Like Potential1, all Potential2 objects must provide getPotential methods for returning the Potential between two supplied Atoms of differing Molecules.

To assist in Potential2 object creation, a P2SimpleWrapper class is written that when given a Potential it defines all of the Atom-Atom Potentials of two Molecules as being the supplied Potential.

3.9. Display

[2] Well diameter is equivalent to $\lambda*\sigma$, where λ is a multiplier to get the range of the well and σ is the

Display type classes provide visual forms of simulation data, whether it is their table of values, plots of data curves, or the molecule configuration of their phase. To accomplish these capabilities, every display provides a GraphicalElement object that is passed to the Simulation object for rendering the display on the computer screen. Each display also provides update methods that can be called once the Integrator's intervalEvent notices simulation changes. Aside from these general features, each display contains its own set of specific methods and objects based on what it is trying to display. For example, since DisplayPhase is showing a picture of the Phase and the Molecules contained within, a ColorScheme object is necessary for defining the scheme to be used during rendering. And for Displays that are showing results in the form of numbers, such as DisplayBox or DisplayTable, a Unit object is necessary for defining the units corresponding to the numbers or results being displayed.

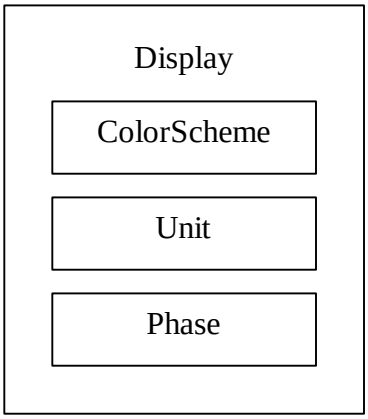


Figure 3.8. Layout of Display Class

3.9.1. ColorScheme

---

coreDiameter.

ColorScheme defines the rule used to determine atom colors when drawn by DisplayPhase. A call to colorAllAtoms alters the color of each Atom based on the results obtained from the algorithm. Atoms are looped through using an Atom Iterator and each Atom has its color changed to the correct color.

Currently, ColorSchemes for coloring by Species, AtomType, Temperature, and Wall Temperature are written. Simply subclassing ColorScheme can create other schemes that can color Atoms based on other criteria.

3.9.2. Unit

The Unit class specifies the physical units to be used when inputting or outputting a quantity. A Unit generally consists of a Prefix (e.g. kilo, milli, Mega) and a BaseUnit (e.g. gram, Kelvin, Newton). Once a Unit is created, its BaseUnit cannot be changed, but using the setPrefix method allows alteration of a Prefix. For internal calculations, all values are converted to simulation units, which are all derived from the picosecond, Dalton, and Angstrom. These conversions are facilitated by the toSim and fromSim methods residing in Unit. For example, the following code is located in DisplayBox:

```
value.setText(format(unit.fromSim(meter.mostRecent()),precision));
```

To begin, a call is made to the mostRecent method of Meter. This returns the current value of the Meter in simulation units. The fromSim method,

```
public final double fromSim(double x) {return from*x;},
```

is then utilized to convert the Meter readout into class units (considering the prefix) by multiplying the Meter reading times the ‘from’ field of the Unit class which is the

predefined conversion factor for the BaseUnit being used. Alternatively, if DisplayBox objects are implemented to allow user input of a given property, the input value will be converted into simulation units using the toSim method,

```
public final double toSim(double x) {return to*x;}.
```

Similar to fromSim, toSim multiplies the input value times the 'to' field of the Unit class which represents the predefined conversion factor for the user-defined property's BaseUnit.

By using these conversion methods, Unit can display results in relevant units by way of a DisplayBox or DisplayTable or take input values and convert them into workable simulation units for running a simulation. When results are displayed, Unit provides a label that shows the corresponding BaseUnit and Prefix.

3.9.3. DisplayPhase

DisplayPhase provides a graphicalElement object that contains the phase of the simulation and all of its occupants, which include molecules and boundaries. A Canvas onto which the configuration of molecules is drawn does most of the work of the DisplayPhase. This canvas provides methods for changing the size of the display as well as listeners for mouse or key events input by the user. Valid key events for the DisplayPhase are any of the number keys, as well as the letters b, s, and o (Assuming the cursor is inside the bounds of the canvas). While periodic boundary conditions (PBC) are implemented, pressing any of the number keys, 0 – 9, changes the number of periodic shells that are displayed (default value is '0', meaning no shell; shell numbers increase up to '9'). Pressing 'b', toggles the presentation of

boundaries on the screen, pressing ‘s’ toggles the scale, and pressing ‘o’ toggles the overflow (Having overflow ‘on’ shows all parts of the molecules from surrounding periodic images that are partially in (‘overflowing’ into) the central periodic image). Default conditions have the boundaries being shown, the scale not being shown, and the overflow parts of molecules from surrounding periodic shells not being shown.

3.9.4. DisplayBox

Displaybox provides readouts from meters that are scanning the thermophysical properties of the phase. As a result, DisplayBox holds a handle to the meter whose current value is being displayed. It also holds a precision field used for setting the number of digits to be displayed, as well as a Unit field for determining and displaying the corresponding Units. By right clicking on a DisplayBox, a DeviceUnitEditor pops up to allow changing the prefix (mega, kilo, milli, etc.) of the unit being displayed, as well as, the base unit itself (e.g. from Celsius to Kelvin).

3.9.5. DisplayPlot

DisplayPlot draws graphs of data values received from a meter. All plotting done by the DisplayPlot is accomplished by using an outside software package known as Ptplot 3.1 developed as part of the Ptolemy project at UC-Berkley. Ptplot is a 2D data plotter and histogram tool implemented in Java. Besides being freeware, it was chosen for its functionality and its ease of integration into the Molecular Simulation API due to its code being exclusively Java based.

Along with the Plot object, DisplayPlot also holds a handle to the meterFunction whose values it is displaying. Values are read in by the meterFunction, which are then plugged into the function represented by the meterFunction and then a plot of the resulting values is displayed. Similar to the DisplayBox, DisplayPlot also has a unit field that corresponds to the values being displayed. However DisplayPlot has two unit fields, one for the x-axis and one for the y-axis.

3.9.6. DisplayMeterHistogram

DisplayMeterHistogram is similar to DisplayPlot. It also uses Ptplot 3.1, and is capable of displaying values from meterFunctions. However, unlike DisplayPlot, DisplayMeterHistogram can display values from meters as well. As you would expect, it holds a handle to the meter or meterFunction whose values it is displaying. If this handle is to a meter, values are just displayed in a histogram form. If it is a meterFunction, values are read in by the meterFunction, which are then plugged into the function represented by the meterFunction and then a histogram of the resulting values is displayed. DisplayMeterHistogram has the same unit fields (x and y axes) as DisplayPlot.

3.9.7. DisplayScrollingGraph[3]

DisplayScrollingGraph like most of the other displays holds a meter object whose values it displays, as well as a unit field for displaying the value's

corresponding units. However, unlike the other displays, DisplayScrollingGraph shows a current value versus time plot that constantly scrolls through the results like a stock ticker. The resetScale method provides the scrolling feature by changing the min and max values of the time axis as a simulation progresses. This way the leading edge of the graph, corresponding to the most recent Meter reading, is always displayed. A 'clear' method is provided for erasing the graph. DisplayScrollingGraph has methods for changing the placement of tick marks on the graph, but the user generally does not call these.

3.9.8. DisplayTable

DisplayTable provides users with a table of all the running averages of the meters currently present in the simulation. It holds an array of these meter objects, for easy access to the current values of the respective meters. DisplayTable also always has a kinetic and potential energy meter, but does not display their values by default. These meters are kept in the phase, but DisplayTable holds a handle to the phase and has access to these meters through that handle. Average kinetic and potential energy values can be displayed by calling the setShowingKE or setShowingPE methods with a parameter of true.

3.10. Modulator

Modulator is an interface that permits changes to the properties of another object. To determine the physical dimensions (e.g., mass, length, pressure, etc.) of

[3] Developed by Richard Gonsalves, Chair of the Dept. of Physics, University at Buffalo, Buffalo, NY,

the property being modified, all Modulators have a getDimension method that returns these dimensions.

In its simplest implementation, Modulator can be hard coded to a given property's accessor (get) and mutator (set) methods. Through these methods, a Modulator can alter the current value of a property (setValue) or simply return the value of the property (getValue).

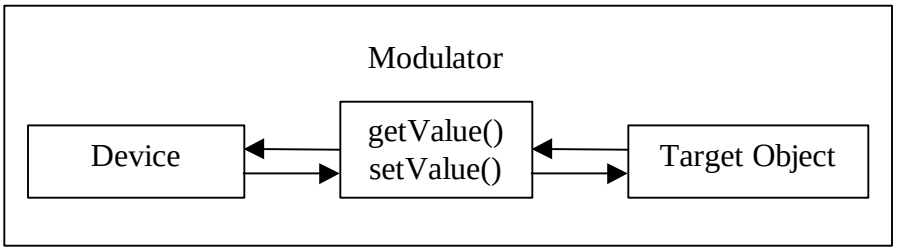


Figure 3.9. Layout of Modulator Interface

A more advanced implementation of Modulator uses Introspection to obtain the accessor and mutator methods of a property. Once located, calls to these methods can either alter the current value of a property, or return the current value of the property similar to the implementation above. Modulators of this type can alter the same property of several objects at once (i.e. temperature of several different objects can be set at simultaneously).

Modulators leave the object or property (e.g. temperature) being modified unknown to the object (e.g. DeviceSlider) using the modulator. For example, a Device, such as a DeviceSlider, can be linked to the temperature of an Integrator by way of a Modulator, but the Device will have no idea that it is linked to the Integrator

in this way. The Modulator will simply read the current value of the DeviceSlider and modify the temperature of the Integrator accordingly.

3.11. Device

Devices are objects that provide access to and methods for changing or modifying fields of other objects. Devices provide graphical interfaces for modifying these fields, as well as a Unit object for determining the corresponding Units. Most Devices will be linkable to a Modulator through which the properties of another object can be modified based on the current value or readout of the Device.

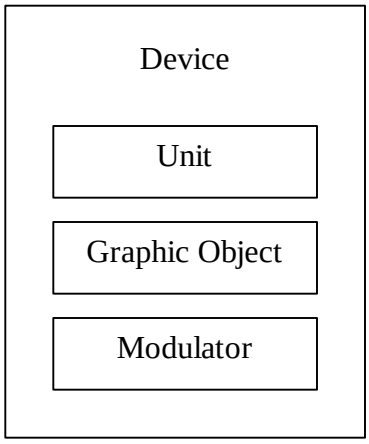


Figure 3.10. Layout of Device Class

3.11.1. DeviceUnitEditor

A simple example of a device is the DeviceUnitEditor associated with a DisplayBox. The DeviceUnitEditor provides a scrollable popup menu as its graphical

interface that displays all the prefixes (mega, kilo, milli, etc.) associated with a unit, as well as all base Unit types relevant to the thermophysical property displayed in the DisplayBox. Once a prefix and a base Unit are chosen, the DeviceUnitEditor automatically updates the Unit label associated with the DisplayBox.

3.11.2. DeviceSlider

A more sophisticated device is the DeviceSlider. The DeviceSlider provides a slider mechanism as its graphical interface for changing the fields of an attached object. The DeviceSlider always holds a component object that is having modulation done to its property field, as well as a property object corresponding to the property be modulated. The minimum and maximum values of the slider are set by default to zero. Therefore, it is necessary to use the PropertySheet to set the minimum and maximum values of the property being changed. Calling the setMinimum and setMaximum methods of the slider can alter these values. To assist in the modulation, the DeviceSlider utilizes a Modulator object. When the slider bar of the DeviceSlider is moved, the Modulator reads the new value from the slider and then updates the property field of the object associated with the DeviceSlider.

3.12. Meter

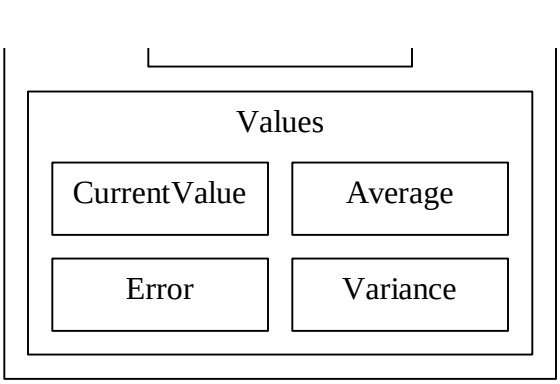


Figure 3.11. Layout of Meter Class

A Meter is responsible for measuring and averaging properties during a simulation. When instantiated, Meters are added to the phase with the setPhase method. Once added, Meters perform measurements on that Phase. Once values are measured, however, Meters are not capable of displaying the results. Instead, Meters depend on Displays, such as the DisplayBox or DisplayTable, to present results to the user. Therefore, Meters must be connected to a Display object otherwise they are useless.

When Meters do conduct measurements, all values are reported in simulation units. The Display object that is presenting the meter data completes conversion to other units. During a simulation, Meters keep a running average that is updated after a certain number of integrator interval events. Meters also have methods for computing the variance and the 67% confidence limit error bar. These statistical values rely on the Accumulator object of MeterAbstract, which maintains all of the measured values of a meter. If needed, a Meter has a reset method that resets these statistical values to zero. Finally Meters have a histogram feature that by default is turned off, but can be utilized by calling setHistogramming.

Meters can be one of two types, a general Meter or a MeterFunction. Meters report scalar quantities (e.g. temperature), while MeterFunctions report vectors or function properties (e.g. radial distribution function). By subclassing MeterAbstract,

other types of Meters can be created for measuring (e.g. Meters measuring Vector or Tensor quantities). In most cases, when creating new Meters it is only necessary to write a currentValue method that defines the property being measured and provides a way of obtaining the value of the property. The other methods present in MeterAbstract take care of evaluating statistics based on this value.

3.12.1. Histogram

Histogram takes values from Meters and determines the distribution of the results obtained from that Meter which can then be viewed in an appropriate Display. Histogram utilizes the Plot feature of Ptolemy Plot 3.1. DisplayPlot uses the getPlot method of Histogram to obtain this Plot. Histograms also have an autoScale feature that automatically scales the DisplayPlot object so that all results are visible. This is done by determining the maximum and minimum values of the Histogram and then calling the redistribute method to calculate the size and placement of the new Histogram bars.

3.13. Default Conditions of a Simulation

Most default conditions of the Molecular Simulation API are held within the Default class. These include everything from molecule specifications to directory pathnames for class libraries and component icons. Default conditions can be modified, by simply changing the values of the fields in the Default class.

The working class directory is set as "D:/Etomica/simulate". Therefore all simulate classes must be located in this directory unless the CLASS_DIRECTORY

variable is modified accordingly. Similarly, the working image directory is set as "file:/D:/Etomica/images/". This directory holds all *.gif files used as icons for Jbuttons in the Etomica environment. Once again, unless this pathname is altered, all image files must be kept in this directory.

By default, atoms have an ATOM_MASS of 40.0 daltons, an ATOM_SIZE of 3.0 Angstroms, and an ATOM_COLOR of black. All species of molecules begin with a MOLECULE_COUNT of 20 in the base SEQUENTIAL configuration of the phase.

The phase of a simulation is displayed in a default simulation BOX_SIZE of 30.0 Angstroms (corresponds to 300 pixels on the computer screen). The initial TEMPERATURE of the phase is set at 300.0 K, with a PRESSURE of 1.0 atm. The TIME_STEP for integrators of a phase is set at 0.05 picoseconds.

4. Etomica Molecular Simulation Environment

The Etomica Molecular Simulation Environment is a graphical user interface (GUI) designed to provide point-and-click access to the simulation tools contained in the Molecular Simulation API

4.1. What is a GUI?

GUI is an acronym for Graphical User Interface. In general terms, a GUI allows interaction with a computer using pictures and symbols rather than just entering typed text at a command prompt. Examples of GUIs include Microsoft Windows, as well as the X-Window System. The purpose of a graphical interface is

to make a computer easier to use. Rather than having to memorize many complicated commands and type them in precisely, you may point and click with an input device, usually a mouse, to run programs or manipulate files.

4.2. Benefits of the Etomica GUI

The Etomica GUI provides the user with a number of functions. First and foremost, the Etomica GUI empowers users with a means of interacting with the Molecular Simulation API without the user needing any experience or knowledge of the Java programming language.

Secondly, by having a GUI that allows real time modification of simulation conditions (by way of the Etomica Property Sheet, Section 4.3.1), users can effectively steer simulations in response to observations made while they are running.

Finally, the Etomica GUI presents users a picture of the results from the simulation. This provides users with visual feedback that complement the usual simple logs of numbers. This way trends (e.g. clustering of molecules, inflection points in graphs), as well as, outliers in data become visibly apparent almost immediately even before lengthy data analysis begins.

4.3. Etomica Molecular Simulation Environment GUI Components

After becoming acquainted with the Molecular Simulation API and the methodologies of a GUI, it is now time to see the specific implementations used in the Etomica environment to interface with the Molecular Simulation API.

4.3.1. Graphical Hierarchy of Classes

The following is a hierarchy diagram of the Etomica environment. Although in some cases the diagram shows an inheritance hierarchy, inheritance is not implied throughout.

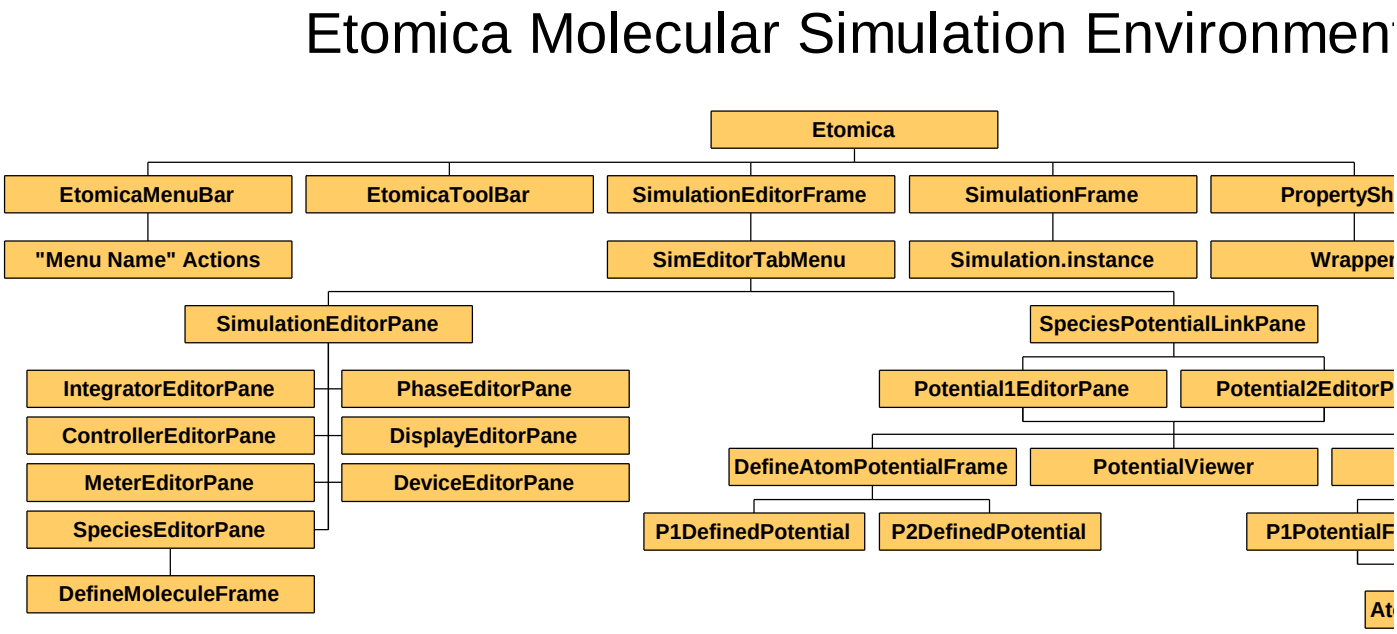


Figure 4.1 Hierarchy of Etomica Molecular Simulation Environment

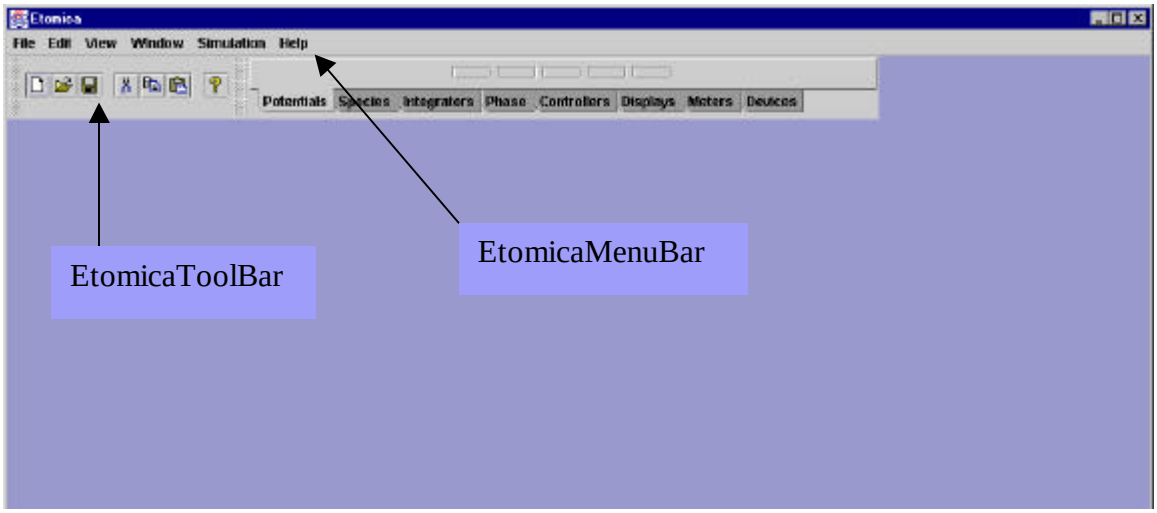


Figure 4.2 Etomica Frame

4.3.2. Etomica

Etomica is the main class for the Etomica Environment. It creates a JFrame (Figure 4.1) that serves as the main window, as well as a JDesktopPane that acts as the desktop of the environment. Within this desktop, Etomica creates a menubar (EtomicaMenuBar) and a toolbar (EtomicaToolBar) that provide all the options to the user for working with simulations.

4.3.2.1. EtomicaMenuBar (refer to Figure 4.1)

EtomicaMenuBar (JMenuBar) includes all the drop-down menus (JMenu) used in the main window of the Etomica Environment. These include the File, Edit, View, Window, Simulation, and Help menus. All selections on these menus are connected to static Actions via ActionListeners. File selections are connected to the FileActions set of Actions, Edit selections to the EditActions, and so on. Currently

not every selection under these menus is functional. Consequently, only those selections that are functional are explained below.

The File menu includes all the operations given to the user to perform on a Simulation object. These are serializing ('Serialize (Applet Form)') applets, serializing editable forms of ('Serialize (Edit Form)'), loading ('Load'), printing ('Print'), clearing ('Clear'), or exiting ('Exit') a simulation.

Although they have EditActions associated with them, the Edit menu items are currently not being used. The Edit items consist of 'Copy', 'Paste', 'Cut', 'Customize', and 'Bind Property'. 'Customize' is the only one of these selections that is functional on the Edit menu. When selected, the Customizer of the currently selected component from the EditorPanes will be shown in a separate JInternalFrame. The remainder of these selections could be helpful if a form design feature is ever implemented into the Etomica environment for dragging and dropping components into a window to create a simulation. This way added components could be copied, cut, or pasted to/from a simulation, and added components could be bound to other components as listeners of events.

The View menu provides a 'PropertySheet' selection that when selected opens the static instance of the PropertySheet. No more than one PropertySheet object is ever permitted in the Etomica environment. For this reason, the PropertySheetItem checks to see if one was previously created by itself or another object. If so, it displays that instance of the PropertySheet. Only if it determines that one does not exist will it create an instance.

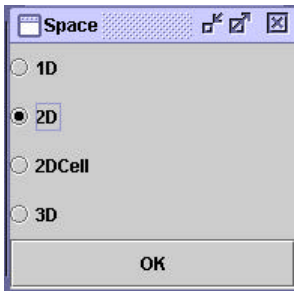


Figure 4.3 Select Space Frame

The Simulation menu has 'Select Space' and 'Edit Simulation' as its selections. 'Select Space' creates a JinternalFrame with the valid dimensions for a Space object (Figure 4.2). Choosing a dimension for the Space is the first step in creating a simulation. The 'Edit Simulation' option opens the SimulationEditorFrame (Figure 4.3) corresponding to the currently open simulation. If a simulation is not currently open, this menu item does nothing.

The Window menu contains choices of 'Next,' 'Cascade,' 'Tile,' and 'Drag Outline.' The Etomica environment keeps a list of all frames or windows contained on its desktop. By choosing 'Next,' the frame or window that is next in sequence is given the focus. Selecting 'Cascade,' cascades all the frames and windows currently contained in the desktop. Selecting the 'Tile' option, tiles all the frames and windows by resizing them and moving them so that they each receive an equivalent amount of the desktop but do not overlap. Finally, 'Drag Outline' toggles the outline feature of each frame or window. When turned on, if a window is dragged, only its outline is drawn to show where it will be placed. This is more efficient and helps the video card by allowing it not to have to constantly repaint the whole window pixel by pixel as it is drawn.

The Help menu has the options ‘Help’ and ‘About.’ The ‘Help’ option opens a JInternalFrame capable of displaying web pages. It defaults to the documentation pages of the Etomica environment. Selecting ‘About’ opens a window that displays version and author information of the Etomica environment.

4.3.2.2. EtomicaToolBar (refer to Figure 4.1)

The ToolBar of the Etomica environment is also connected via ActionListeners to the Action classes. However, these currently have no functions. If needed, functionality can be added to these buttons by adding code to the actionPerformed methods of the Actions classes that these buttons call.

The .gif files that serve as ImageIcons for these buttons reside in the \images directory. The pathname for the placement of these files can be found and altered in the Default class of the Molecular Simulation API.

4.3.3. SimulationEditorFrame

The SimulationEditorFrame (Figure 4.3) is a JInternalFrame that holds the SimEditorTabMenu. After a dimension is chosen for the Space object used in a simulation, this frame is instantiated and displayed.

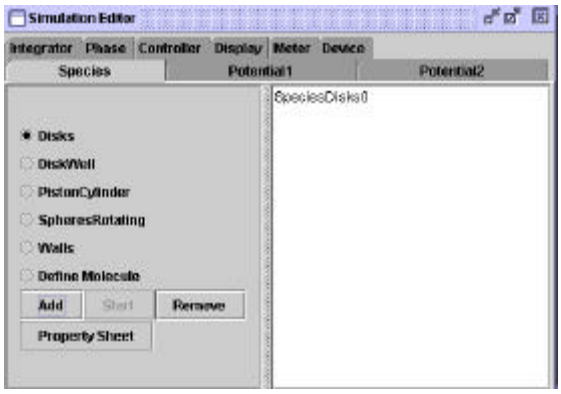


Figure 4.4 SimulationEditorFrame

4.3.3.1. SimEditorTabMenu

The SimEditorTabMenu (refer to Figure 4.3) is a JTabbedPane that holds static handles to the EditorPanes used in the Etomica environment. These panes allow users to add and remove components to/from the Simulation. SimEditorTabMenu provides methods for enabling the start (setAllStart) and remove (setAllRemove) buttons from all of the EditorPanes simultaneously. To supplement these, methods for checking the status of the start (allStartEnabled) and remove (allRemoveEnabled) buttons of the EditorPanes are also present. By using the reset method (resetAllComponentLists) in SimEditorTabMenu, all of the JLists of the EditorPanes containing all the components currently added to Simulation can be erased. When 'Clear' is selected from the File menu, this method is called.

4.3.4. SimulationEditorPane

SimulationEditorPane (refer to Figure 4.3) is the superclass of all EditorPanes contained in the SimEditorTabMenu. These are SpeciesEditorPane, IntegratorEditorPane, PhaseEditorPane, ControllerEditorPane, DisplayEditorPane,

MeterEditorPane, and DeviceEditorPane. SimulationEditorPane sets the layout of each EditorPane and draws all of the GUI objects residing in the EditorPanes.

4.3.4.1. Structure

The SimulationEditorPane is a JSplitPane containing a JPanel set in a JScrollPane on the left side and a JList set in a JScrollPane on the right side. The JPanel contains a set of JRadioButtons (one for every component of that Panes type; i.e. Display would have a button for every Display found in the class path). These are created by the EditorPanes themselves by calling the makeRadioButtons method and supplying an array of component classes specific to each EditorPane that will be linked to each radio button. The Etomica environment creates the supplied arrays automatically when it is initialized. The folders listed in the class path are scanned for any classes that follow the specified naming structure of the Molecular Simulation API. As these classes are found, they are added to the arrays for later use.

Four other JButtons are contained on the JPanel. These are 'Add,' 'Remove,' 'Start,' and 'Property Sheet.' The JList starts out being empty, but as components are added to the simulation, it displays the names associated with the added components.

4.3.4.2. Functionality

All the functionality of the SimulationEditorPane (Figure 4.3) is performed by the JButtons. Clicking 'Add' instantiates the component corresponding to the currently selected JRadioButton and adds it to the simulation. The 'Remove' button deletes the component of the JList that is currently selected. Pressing 'Start' (when

enabled), displays the SimulationFrame (Figure 4.11) that holds the Simulation object and all the components added to it. And finally, the 'Property Sheet' button displays the PropertySheet (Figure 4.12) and updates it to show the properties corresponding to the component of the JList that is currently selected.

4.3.4.3. EditorPanes

EditorPanes that subclass SimulationEditorPane are written for each type of simulation component found in the Molecular Simulation API except Potentials (Potentials have special panes written for them that subclass SpeciesPotentialLinkPane). Each EditorPane relies heavily on the inherited features of SimulationEditorPane for their layouts, graphics, and functionality. EditorPanes only individualize themselves by sending a different array of classes to the makeRadioButtons method of SimulationEditorPane and by modifying the actionPerformed methods called by the add and remove buttons to compensate for the differences of the classes packaged in those arrays. One exception exists to this rule. The SpeciesEditorPane incorporates everything that has been listed, but also adds an extra selection to the list of classes. This is the 'Define Molecule' selection. Selecting 'Define Molecule' opens the DefineMoleculeFrame (Figure 4.4) object. This frame allows users to create a Molecule Atom by Atom for use in a Species rather than simply using an already existing Species class. This provides users with an endless list of possibilities for a Species.

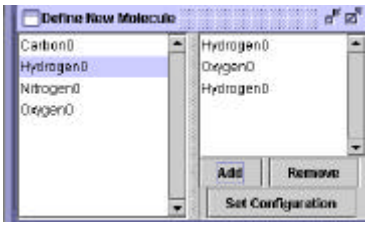


Figure 4.5 DefineMoleculeFrame

4.3.4.4. DefineMoleculeFrame

DefineMoleculeFrame (Figure 4.4) is a JInternalFrame consisting of two JLists and three JButtons. The left JList shows all the AtomTypes written in the Molecular Simulation API that define elements found on the Periodic Table of Elements. The JList on the right shows all of the Atoms (in the order they will be attached) that have been added together to form the Species' new Molecule. Clicking 'Add' creates the Atom or element selected on the left JList, appends it to the end of the right JList, and attaches it to the new Molecule. Pressing 'Remove', removes the selected Atom of the right JList from the Molecule being created. Finally, the 'Set Configuration' button when pressed opens the DefineConfigurationFrame for selecting the configuration in which to put the Atoms of the new Molecule.

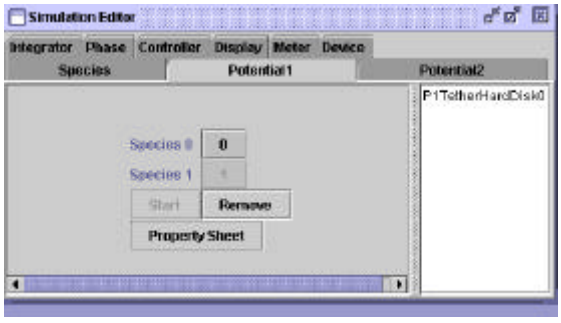


Figure 4.6 SpeciesPotentialLinkPane (specifically Potential1EditorPane)

4.3.5. SpeciesPotentialLinkPane

SpeciesPotentialLinkPane (Figure 4.5) is the super class of the PotentialEditorPanes. These are the Potential1EditorPane and the Potential2EditorPane. Similar to the SimulationEditorPane, SpeciesPotentialLinkPane sets up the graphics and functionality for its subclasses. The subclasses only duty is to provide an update method that governs the layout with which objects will be redrawn in the event of a change to the simulation.

The only change that necessitates a call to update is the addition or removal of a Species to/from a simulation.

4.3.5.1. Structure

The SpeciesPotentialLinkPane is a JSplitPane containing a JPanel set in a JScrollPane on the left side and a JList set in a JScrollPane on the right side. The JPanel contains a set of JButtons that are labeled with the Species indices of the Species that are referred to by each button. These buttons provides a means for defining a Potential between Molecules of two different Species. Three other JButtons are contained on the JPanel. These are 'Remove,' 'Start,' and 'Property Sheet.' The JList starts out being empty, but as components are added to the simulation, it displays a name corresponding to the added Potential.

4.3.5.2. Functionality

When pressed, the buttons open the LinkPane's corresponding PotentalFrame (P1PotentialFrame, Figure 4.6, for Potential1EditorPane and P2PotentialFrame,

Figure 4.7, for Potential2EditorPane) that displays all the relevant Potentials. Clicking 'Remove' deletes the component of the JList that is currently selected. Pressing 'Start' (when enabled), displays the SimulationFrame (Figure 4.11) that holds the Simulation object and all the components added to it. And finally, the 'Property Sheet' button displays the PropertySheet (Figure 4.12) and updates it to show the properties corresponding to the Potential of the JList that is currently selected.

4.3.5.3. LinkPanes

As was stated before, the LinkPanes inherit all of their functionality and graphics from SpeciesPotentialLinkPane. Potential1EditorPane and Potential2EditorPane only provide an update method that contains an algorithm for drawing the JButtons in a column format for Potential1 (since they only deal with a single Species) and an upper triangular matrix format for Potential2 (since they refer to two Species).

4.3.5.4. PotentialFrame

PotentialFrame is the super class of all frames holding a list of Potentials. These are P1PotentialFrame (Figure 4.6), P2PotentialFrame (Figure 4.7), and AtomPotentialFrame (Figure 4.8). PotentialFrame works in a similar manner to SimulationEditorPane's setting up of its subclasses, the simulation component EditorPanes. PotentialFrame determines the layout of each subframe and supplies a makeRadioButton method through which the subframes create radio buttons for their

respective array of Potentials. PotentialFrame also supplies the 'Add' and 'Cancel' buttons used by each subframe, however, the actionPerformed method of these buttons is defined by the subframes themselves.

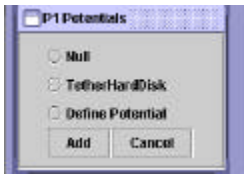


Figure 4.7 P1PotentialFrame

4.3.5.4.1. P1PotentialFrame

P1PotentialFrame (Figure 4.6) displays all intra-molecular Potentials located on the class path. Since the Molecular Simulation API uses a naming structure that requires all Potentials to begin with the word Potential (e.g. PotentialHardDisk, PotentialSquareWell, etc.), Potentials are located by a static block of code that scans the class library directory supplied by the Default class and returns all classes that begin with 'Potential.' These are stored in an array called potential1Classes and are supplied to the makeRadioButtons method of PotentialFrame for implementation onto the frame. In addition, P1PotentialFrame creates a 'Define Potential' radio button that when selected opens the DefineAtomPotentialFrame (Figure 4.9) for Atom-by-Atom creation of a Potential between two molecules. P1PotentialFrame also supplies the actionPerformed methods of the 'Add' and 'Cancel' buttons contained on the frame.

P1PotentialFrame is linked by a buttonListener object to the buttons contained on the Potential1EditorPane (Figure 4.5), and therefore, is only opened when one of

these buttons is pressed. Whenever the 'Add' button is pressed, the button that originally called the P1PotentialFrame will store a handle to the currently selected Potential, as well as, add the Potential to the JList of the Potential1EditorPane.

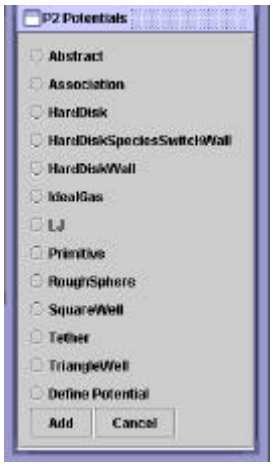


Figure 4.8 P2PotentialFrame

4.3.5.4.2. P2PotentialFrame

P2PotentialFrame (Figure 4.7) displays all inter-molecular Potentials located on the class path. Similar to P1PotentialFrame, the naming structure of the Molecular Simulation API is utilized by a static block of code that scans the class library directory supplied by the Default class to locate and return all the relevant Potentials. These are stored in an array called potential2Classes and are supplied to the makeRadioButtons method of PotentialFrame for implementation onto the frame. In addition, P2PotentialFrame creates a 'Define Potential' radio button that when selected opens the DefineAtomPotentialFrame (Figure 4.9) for Atom-by-Atom creation of a Potential between two molecules. P2PotentialFrame also supplies the actionPerformed methods of the 'Add' and 'Cancel' buttons contained on the frame.

P2PotentialFrame is linked by a buttonListener object to the buttons contained on the Potential2EditorPane, and therefore, is only opened when one of these buttons is pressed. Whenever the 'Add' button is pressed, the button that originally called the P2PotentialFrame will store a handle to the currently selected Potential, as well as, add the Potential to the JList of the Potential2EditorPane.

4.3.5.4.3. AtomPotentialFrame

AtomPotentialFrame (Figure 4.8) displays all Potentials located on the class path. Once again, a static block of code that scans the class library directory supplied by the Default class to locate and return all the relevant Potentials utilizes the naming structure of the Molecular Simulation API. The Potentials are stored in an array called potentialClasses and are supplied to the makeRadioButtons method of PotentialFrame for implementation onto the frame. AtomPotentialFrame also supplies the actionPerformed methods of the 'Add' and 'Cancel' buttons contained on the frame.

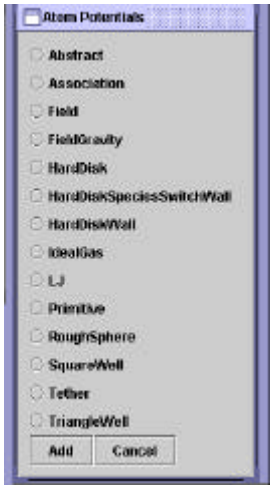


Figure 4.9 AtomPotentialFrame

AtomPotentialFrame is not linked to any of the EditorPanes, but rather is linked by a buttonListener object to the buttons of the DefineAtomPotentialFrame (Figure 4.9). It is used to define Potentials for Molecule interactions in an Atom-by-Atom format. Whenever the 'Add' button is pressed, the button that originally called the AtomPotentialFrame will store a handle to the currently selected Potential.

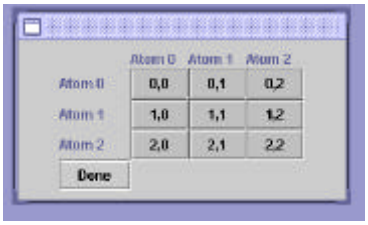


Figure 4.10 DefineAtomPotentialFrame

4.3.5.5. DefineAtomPotentialFrame

DefineAtomPotentialFrame (Figure 4.9) is a JInternalFrame that holds an array of buttons that represent the interaction Potential between Atoms of a Molecule. The 'Add' buttons of P1PotentialFrame and P2PotentialFrame call DefineAtomPotentialFrame when the 'Define Potential' option is selected. At this point, DefineAtomPotentialFrame determines the number of Atoms present in the two Molecules in which a Potential is being defined. Based on this number an array of buttons is created and displayed with each button representing a single Atom-Atom potential. For example, if Molecule A has three Atoms and Molecule B has four, then a 3X4 matrix will be created. When pressed, each of the buttons in the matrix opens the AtomPotentialFrame (Figure 4.8) and the button will hold a handle to the Potential that is selected by the user from this frame. When all Atom-Atom

interactions are defined, the ‘Done’ button can be pressed. This instantiates a new Potential (either P1DefinedPotential or P2DefinedPotential depending on the frame that originally called DefineAtomPotentialFrame) and adds it to the JList of the current SpeciesPotentialLinkPane (either Potential1EditorPane or Potential2EditorPane, Figure 4.5).

4.3.5.5.1. P1DefinedPotential

P1DefinedPotential is a subclass of Potential1 from the Molecular Simulation API. Therefore all Potentials of this type define inter-molecular interactions. DefineAtomPotentialFrame (Figure 4.9) creates Potentials of this type when the PotentialFrame that originally opened it is of type P1PotentialFrame (Figure 4.6). The Atom Potentials that make up this Molecule Potential are stored in an array. Like all Potentials, P1DefinedPotential provides a getPotential method that searches through this array and returns the Potential between the two given Atoms of a single Molecule.

4.3.5.5.2. P2DefinedPotential

P2DefinedPotential is a subclass of Potential2 from the Molecular Simulation API. Therefore all Potentials of this type define intra-molecular interactions. DefineAtomPotentialFrame (Figure 4.9) creates Potentials of this type when the PotentialFrame that originally opened it is of type P2PotentialFrame (Figure 4.7). The Atom Potentials that make up this Molecule Potential are stored in an array. Like all Potentials, P2DefinedPotential provides a getPotential method that searches

through this array and returns the Potential between the two given Atoms of two different Molecules.

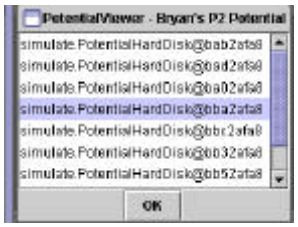


Figure 4.11 PotentialViewer

4.3.5.5.3. PotentialViewer

PotentialViewer (Figure 4.10) is a JInternalFrame that contains a scrollable JList for viewing the individual Atom-Atom Potentials of a given Molecule Potential. This frame is opened when a Potential of type P1DefinedPotential or P2DefinedPotential is selected from a SpeciesPotentialLinkPane (Figure 4.5). The Atom-Atom Potentials of this DefinedPotential are then listed in the viewer. If any of these Atom-Atom Potentials are selected, the static PropertySheet object of the Etomica environment is opened and will display the properties of the given Atom-Atom Potential.

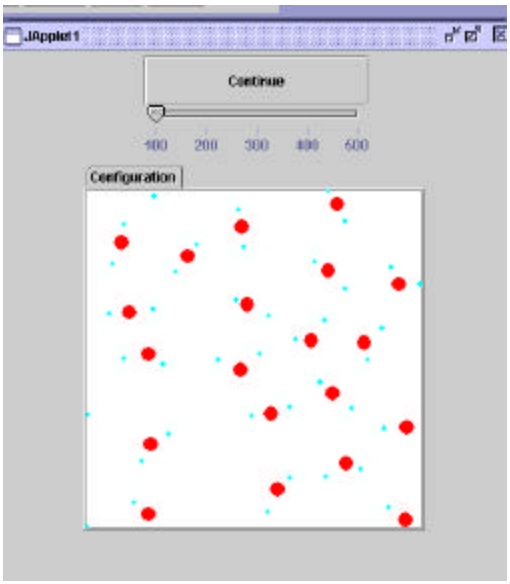


Figure 4.12 SimulationFrame

4.3.6. SimulationFrame

SimulationFrame (Figure 4.11) is a JInternalFrame that holds and displays a Simulation object. This object is always the static Simulation.instance object created by the Simulation class. SimulationFrame provides users with a means for interacting with and viewing a simulation created by the Molecular Simulation API. SimulationFrame is opened when the 'Start' button of any of the EditorPanes is pressed.

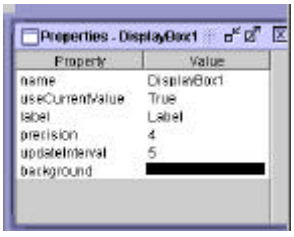


Figure 4.13 PropertySheet (Displaying properties of DisplayBox)

4.3.7. PropertySheet

PropertySheet (Figure 4.12) implements Java Beans Introspection technology to allow modification of object properties with the use of a GUI component rather than by writing code. This is done utilizing Java's ability to locate accessor (get) and mutator (set) methods corresponding to a given property so that a property's value can not only be displayed, but also modified while a simulation is in progress.

4.3.7.1. Structure

PropertySheet is a JInternalFrame that contains a JPanel that utilizes a JTreeTable (a modification of the JTree) to display the labels of an object's property and their corresponding values in a format similar to Windows Explorer. Each property label and its value is held in a node of the tree. The CellRenderer displays a graphical view of each property label and the corresponding value. When properties of an object need to be modified, the CellEditor provides a relevant means for easily changing the properties value depending on the data type of the value being shown. For Color objects, a PropertyCanvas is displayed. For properties with multiple choices of abstract data type objects a PropertySelector is implemented. And finally, for Integer and String objects a PropertyText object is used.

4.3.7.2. Functionality

PropertySheet works by using Introspection to locate the set and get methods of the selected property. The get method is then used to determine the current value of the property so that it can be displayed in the JTreeTable of the PropertySheet via

the CellRenderer. When a property value is clicked, the getTableCellEditorComponent method is called and it responds by returning the relevant interface for modifying that property (PropertyCanvas, PropertySelector, etc.). When a new value is entered for the property, the PropertySheet uses the set method that it obtained to update the current value of the property to the newly input value. At this point, all objects utilizing the property that has just been modified are notified of the change so that the entire simulation can be updated to reflect the new value.

4.3.7.3. Advantages of the PropertySheet

The object property accessibility of the PropertySheet provides users a means of altering the default conditions of the currently added simulation components. Therefore simulation creation and setup is highly simplified and completely free of user knowledge of Java. This accessibility also provides users with the sought after means of steering a simulation once in progress.

4.3.7.4. PropertyCanvas

PropertyCanvas is a JPanel that displays the color of the current Color object. It listens for mouse clicks and responds to them by opening a color editor through which a user can select a new color for the object. It then notifies all other objects using this property of the change.

4.3.7.5. PropertySelector

PropertySelector is a JComboBox that displays the title of the value of the current object. It listens for itemEvents that occur when the drop-down menu is used to select a value differing from the current one. It responds to these events by changing the currently displayed title to the title of the option that was just selected. It then notifies all other objects using this property of the change.

4.3.7.6. PropertyText

PropertyText is a JTextField that displays the String representation of the Integer value or title of the current object. It listens for keyEvents that occur when the keyboard is used to type in a new value. It responds to these events by changing the currently displayed String representation to the one that corresponds to number or letter keys that were just pressed. It then notifies all other objects using this property of the change.

4.3.7.7. PropertyNode

PropertyNode is a node of a PropertyTree representing an object. It subclasses DefaultMutableTreeNode and has the ability to hold a JLabel (label) and a component (view). The JLabel is a title representation of a property, and the component is a graphical interface to a PropertyEditor that can be used to modify the value of a property. PropertyNode provides a label method for accessing the label field and a view method for accessing the view field.

4.3.7.8. PropertyModel

PropertyModel subclasses AbstractTreeTableModel and defines the layout of the JTreeTable used in the PropertySheet. It sets the model as two columns (one for the label fields and one for the component fields listed in the PropertyNodes). A method is provided for returning the value of a node (getValueAt). Methods are also provided for accessing a particular column name (getColumnName), node of a column (getChild), number of nodes in a column (getChildCount), and the number of columns (getColumnCount).

5. Sample Hard-Sphere Simulation

To get a better understanding of the steps that are necessary to create a simulation, this chapter demonstrates how to put together a hard-sphere molecular dynamics simulation along with a Meter for measuring the current temperature of the system and a DeviceSlider that can alter the temperature of the system by way of a Modulator.

5.1. Necessary Components

- 2-D Space
- SpeciesDisks
- PotentialHardDisk
- IntegratorHard
- Phase
- Controller
- DisplayPhase
- DisplayBox
- MeterTemperature
- DeviceSlider

5.2. Creating a Simulation that Contains these Components

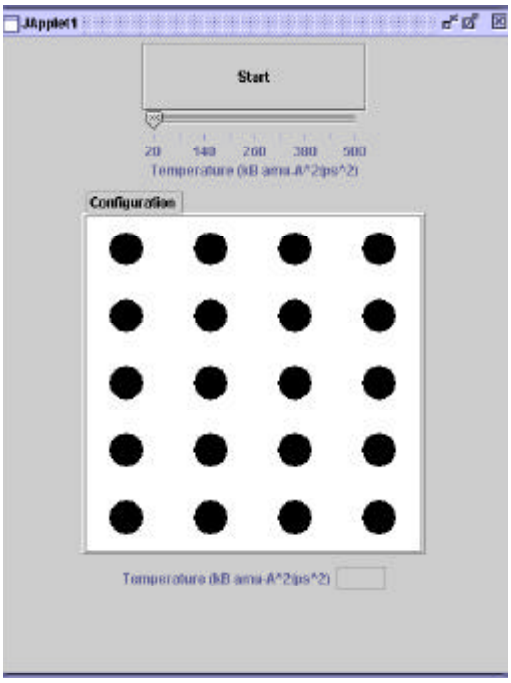
Begin by selecting File->New Simulation. The SpaceSelectionFrame will appear displaying 1-D, 2-D, 2-Dcell, and 3-D as choices of the Space object. Choose 2-D and click 'OK.' The SimEditorTabMenu (Fig. 4.3) will then appear with tabs present for each type of simulation component. The Species tab will be on top. Systematically go through each tab and select the following choices. After selecting a choice, press the 'Add' button on each tab.

Menu Tab	**Component Selection**
Species	Select 'Disks'
Potential2	Press '0,0' button then select 'HardDisk'
Integrator	Select 'HardDisk'
Phase	Press 'Add Phase' button
Controller	Select 'Controller'
Display	Select 'Box'
Display	Select 'Phase'
Meter	Select 'Temperature'
Device	Select 'Slider'

Now that all of the components are added, select the Display tab again. In the list on the right of the tab, select 'DisplayBox.' The PropertySheet (Fig. 4.12) will appear displaying the corresponding properties of the DisplayBox. Click on the cell to the right of the word Meter in the DisplayBox PropertySheet. A drop-down menu

will appear that contains the MeterTemperature meter that was added to the simulation. Select this Meter. This will link the Temperature Meter to the DisplayBox so that the system's temperature will be displayed. Now select the Device tab. Click on 'DeviceSlider' in the list on the right. The PropertySheet (Fig. 4.12) will now display the properties of the DeviceSlider. Next the word Modulator, click where it says 'click to edit.' The ModulaterEditorPanel will appear. Select 'IntegratorHard' from the list of simulation components. A list of properties will appear on the right. Select 'Temperature' and then press the 'Done' button. This will link the DeviceSlider to the Temperature of the system so that the temperature can be altered during runtime. The ModulatorEditorPanel will disappear. Again in the PropertySheet, click on the integer to the right of the word 'Maximum.' Type in '500.' Now click on the integer to the right of the word 'Minimum' and type in '20.' Finally, press the 'Start' button on any of the tab menus from the SimEditorTabMenu. At this point, the SimulationFrame (Fig. 4.11) will be shown displaying all of the added simulation components similar to the screen shot below.

5.3. Screenshot of Hard-Sphere Molecular Dynamics Simulation



5.1 SimulationFrame(Displaying Hard-Sphere MD Simulation)

6. Features of the Etomica Environment

The Etomica Molecular Simulation Environment not only provides users with a complete means of creating simulations, but also with ways for saving and modifying these simulations in an easy to understand graphical user interface without the necessity of having any knowledge of Java. The package includes:

- Easy Point and Click interfaces for the creation of simulations
- Property Sheet editor for modifying and viewing object properties throughout the life of a simulation
- Serialization feature that provides persistence of a simulation
- Ability to create applets that contain a simulation

- Method of Introspection for easy incorporation of newly created classes into the Etomica environment

6.1. Point and Click Creation of Simulation

The Etomica environment provides simple Point-and-Click creation of simulations. This is done through the help of editor panes that list all of the simulation components in a categorized format, as well as a Property Sheet that serves as a means for changing the default properties of any component that is added to a simulation even while the simulation is running.

6.2. Serialization

A Serialization feature is present in the Etomica environment. It has the power to save the initial conditions of a simulation for easy distribution or for archiving. This is done with the help of the serialization technology of Java that has the ability to turn a collection of objects into a byte stream of data that can be easily saved as a text file. To serialize a simulation, simply create one by using the EditorPanes and before starting it select File->Serialize. Two serialize options exist. These are the Editable form and the Applet form. The editable form (File->Serialize (Edit Form)) allows a simulation to not only be brought back to life, but to have its EditorPanes brought back in their exact form for easy addition and removal of components to the serialized simulation. The applet form (File->Serialize (Applet Form)) is not editable (meaning components can not be added or removed to/from it), but can be distributed and viewed via an html page and a web browser. Currently a

bug exists in the applet creator of the Etomica environment. This causes a security violation error in web browsers and makes the applets inoperable.

6.3. Applet Creation

As stated in the Serialization section above, simulations can be saved in an applet form by selecting File->Serialize (Applet Form). By this method, simulations can be created by a professor and then viewed by an entire class of students from any number of different locations by way of a web browser. This way, students can view simulations when it is convenient for them. Applets also provide a way for results of simulations to be shown to colleagues without the need of reading papers that document the results. Instead, interested parties can actually run the simulation for themselves and see the results unfold in front of them. This makes it easier for new findings to get notoriety, as well as, for these findings to be understood.

6.4. Property Sheet

The PropertySheet and its functions and abilities were described in Section 4.3.7. This tool offers a simple means to modify the conditions of a simulation. The most exciting part however, is that these modifications can be made while a simulation is running. Therefore, researchers can steer a simulation in directions or toward model configurations that give the highest probability of returning the sought after results.

6.5. Introspection

The Etomica environment has the ability to grow with the researcher that is developing with it. This is due to the introspection capabilities of the environment. By following the naming structure of the Molecular Simulation API, a researcher can develop new classes that the Etomica environment is capable of finding and adding to its component lists. This way it is not necessary to recompile the environment to enhance its capabilities when a new type of Species or Potential is created. Rather just store the new class in the class library whose pathname is in the Default class of the API, and the Etomica environment will display the new Species or Potential as a selection on the EditorPanes automatically. This provides a means for the environment as well as the researcher to grow.

7. Errors and Error Handling

With all of its functionality and features, the Etomica environment does have its drawbacks and errors. First, the environment at this point does rely on the user to have some simulation experience. This is mainly because the environment will allow simulation components to be put together even if they have no relevance to one another. Also, the environment will allow simulations to be run even if some components have been added without the addition of other components that it may need. To the researchers dismay, the environment will simply lock up without warning under these circumstances. Second, the applet feature does have some problems when attempting to be run because of security issues. These security violations render simulations inoperable unless run via the appletviewer of the

VisualCafe IDE. Finally, the environment does have toolbar buttons and menubar buttons that are not linked to anything. These are present with the intention of having functionality added in the future. If they are deemed unnecessary, they will be removed.

These are the problems and errors that are currently known. As with all software, many errors are usually undetected until distribution to clients or other users. Until then, it is impossible to know of or list every error. Eventually it is hoped that the errors listed above and any further errors that are found will be eradicated.

8. Future Work

The Etomica environment is definitely not a finished product, but rather a beginning to what may be a never-ending line of improvements, additions, or modifications. Some of these changes will correct some of the errors that are presently known or are sure to be found at a later date. The remainder will enhance the current features and extend their possibilities.

8.1. Smart Environment

The biggest need is to have the environment be "smart enough" to update the component lists of the EditorPanes to reflect the addition of new components. The reason this is needed is to rid users of the task of making sure that the added components are compatible. Also, error messages need to be implemented that will tell a user that a component was added but that another necessary component has not

been added or a supplemental selection has not been made prior to a simulation being started. This will stop the simulation from locking up unexpectedly.

8.2. Multiple Simulations

In its present state, the Etomica environment does not permit the creation of multiple simulations simultaneously. Allowing multiple simulations would be helpful to users and therefore is something that will be altered in future releases. The current hang-up in making this feature a reality is the presence of static fields inside the Simulation class. Removal of these static fields is necessary to permit the running of multiple simulations that contain unique sets of simulation components.

8.3. Scripting

A useful tool that could be implemented into the environment would be a scripting class capable of scanning a created simulation and writing out the Java code that corresponds to that simulation. In this way, users could see exactly what is being done during a simulation, as well as modify the code to conform to their specific needs.

8.4. Data Logging

So that the data output by a simulation can be easily filed for later analysis, a logging class is needed that can take the data supplied by a Meter and export it to a user-designated file or location. Rather than simple text, this logging class could be

designed to output data in formats specific to the analysis software a user may want to utilize.

8.5. Printing Capabilities

Currently the only method of printing the configuration of the phase or the readout of a meter is to use the print screen option of Windows and paste the copy of the screen into Word for printing. Not only is this a hassle, but it prints the entire screen when the user just wants the frame containing the simulation to be printed. Therefore, a printing feature needs to be added that specifically prints the simulation frame to any printer on the network.

9. Appendix

9.1. Naming Structure

The Etomica environment uses introspection techniques to locate all of the class files that define the available components. By using introspection, not only can the environment locate the components packaged with the environment, but also those that are developed by the respective user. However, introspection is only effective if the user names the components according to the predefined naming structure. Since every component subclasses one of the nine general types (species, potential1, potential2, integrator, phase, controller, display, meter, or device), the developed naming structure demands that each user created component be given a title that begins with the name of the general component type that is subclassed followed by a name that is specific to the component. For example, if a meter was

created that measures temperature, a valid name for the component would be MeterTemperature. As long as this format is followed, no other steps are necessary for implementing new components into the Etomica environment.

9.2. Why Visual Café?

Once a language was decided upon, a development environment needs to be acquired that is designed to work with the language that was chosen, namely Java. After searching through the list of current IDEs (Integrated Development Environments) it was determined that the best choice would be WebGain's VisualCafé. For the development of the Etomica Molecular Simulation Environment, VisualCafé version 4.0 Expert Edition was used exclusively. VisualCafé is a 100% pure Java development environment containing a trio of Java 2, Just-In-Time (JIT) compilers, as well as a full-fledged debugger.

9.2.1. Benefits

Some of the benefits of using an IDE, namely VisualCafé 4.0 is its ability to automatically generate some of the more complex file structures. These include Standalone Executables (EXEs), Dynamic Link Libraries (DLLs), and Java Archives (JARs).

First and foremost, in order to make the Etomica environment fully functional without the help of an IDE such as VisualCafé, it is necessary to be able to create a standalone executable. VisualCafé provides this feature by allowing any project that contains a group of classes to be exported in a standalone executable format. This

allows the environment to be distributed and used without the need for all users to have access to VisualCafé.

Secondly, to increase performance of the environment it is necessary to have the ability to create DLL files. Once again, VisualCafé provides this feature as part of its specifications. By doing so, all relevant classes to the environment can be bundled in an efficient library that is readily accessible by the computers processors. This increases the efficiency of the environment and most importantly the execution time of processes.

Finally, to allow for viewing on websites, applets must have all relevant classes bundled in a JAR file. To complete the trifecta, JAR file creation is also an included capability of VisualCafé. This allows for complete distribution across the Internet or any other network in the world.

As you can see, Visual Café supplies all the necessary functionality to the creation of any Java environment. It is because of this that VisualCafé 4.0 Expert Edition was chosen as the IDE for the completion of this project.