Etomica: An Object-Oriented Framework for Molecular Simulation

Andrew J. Schultz and David A. Kofke

Department of Chemical and Biological Engineering, University at Buffalo, The State University of New York, Buffalo, NY, USA

David A. Kofke, kofke@buffalo.edu

Published as:
Etomica: An Object-Oriented Framework for Molecular Simulation

We describe the design of an object-oriented library of software components that are suitable for constructing simulations of systems of interacting particles. The emphasis of the discussion is on the general design of the components and how they interact, and less on details of the programming interface or its implementation.

Keywords: object-oriented; etomica; application programming interface; software

1 Introduction

1.1 Motivation

The scope of molecular simulation has expanded enormously since its initial application to the study of hard-sphere dynamics and freezing. Simulation methods now are employed in the study of a very diverse range of materials (e.g., polymers, small organics, electrons, liquid crystals, colloids, spin lattices, ionic liquids, proteins, glasses, etc.) and phenomena (e.g., condensation, adsorption, solvation, diffusion, reaction, gelation, vitrification, etc.), many of which have been made feasible by enormous strides in computing hardware and simulation algorithms that have taken place in the fifty-or-so years since the first studies were published. Although employed in such diverse ways, there is a clear unifying structure to the way molecular simulation is conducted in all these applications: we repeatedly move around many particles according to some underlying physical law while they interact according to some model, and for some of these configurations we calculate and average quantities that characterize the behavior that emerges. If the molecular model is good, and the simulation is conducted properly (e.g. performs a sufficient number of particle movements), then the results from the simulation should give a true description of the behavior being modeled.
The unity underlying all these applications is difficult to realize in practice, in that most if not all simulation codes are each formulated to apply to only a small subset of these examples. This is understandable: the design of codes is driven by the interests of the developers and their targeted community of users, and it takes time to make a code more broadly useful than needed for its immediate purpose; moreover there may be a performance penalty in doing so. Still, there is value in formulating simulation codes in a way that does not limit them to the applications immediately driving their development. Put simply, such codes are more versatile—their pieces are more easily re-used, and they are more readily extended to new models, methods, problems, and platforms. They are also easier to understand and maintain. Indeed, students of molecular simulation can develop a deeper understanding of the structure of simulation methods and codes if they learn from a well-abstracted design. A good design will transcend any particular implementation, and to the extent it is realized in otherwise independently developed codes, those codes can be made interoperable to a useful degree.

Object-oriented programming constructs have emerged as a popular conceptual approach for the development of complex applications via the assembly of interoperating components, which are selected from a diverse but well-structured library. We assume the reader has some familiarity with object orientation as a programming paradigm, and will not attempt to motivate or explain this approach here. The general ideas are well stated elsewhere.\textsuperscript{1,2} There are nevertheless some important and perhaps less-well-known concepts that are worth highlighting in this regard, and we do this in Sec. 2.

Several object-oriented molecular simulation codes have been developed and described over the past decade or so.\textsuperscript{3-6} Our own activities in this area date back to
1996, when with the advent of the Java programming language we initiated the
development of interactive, graphically-oriented web-based molecular simulations for
teaching thermodynamics and related topics. This motivated the development of an
object-oriented library of molecular simulation components, which we call etomica.
Since then, the etomica code base has grown to more than 2000 classes through use in a
variety of educational and research applications. The original context of etomica’s
development—as an aid to developing educational simulations covering potentially any
topic in statistical physics—led it to be formulated with a strong emphasis on generality.

In this paper we describe the general structure of etomica, emphasizing some of
its key features that facilitate broad applicability. We do not attempt a comprehensive
specification, but instead we describe in words (and a bit of computer code) the way the
component library is structured to promote extensibility and code re-use, while being
suited to cover the broad array of applications suggested in Sec. 1.1. Our goals in
presenting this framework are twofold. First, we hope to contribute ideas that others
may adopt in constructing general-purpose simulation interfaces and libraries of their
own design. Second, we want to provide a reference work for users of etomica, so that
they can grasp its structure as they attempt to apply or extend it.

1.2 Design criteria

Etomica is designed as a programming framework that can accommodate almost any
type of particle-based interaction model and any type of law governing the particles’
behavior. We want, for example, to allow both hard and soft potentials, any spatial
dimension, on- or off-lattice, pairwise-additive or multibody potentials, Monte Carlo
(MC) or molecular dynamics (MD) (or other) simulations, and so on. We want it to
support both batch and interactive uses. The latter encompasses not only the ability to
interact with a functioning simulation, but also the ability to construct or modify a
simulation on-the-fly. Thus we would like the framework to allow the user to select
components to add (or remove), to change fields that affect behavior, and to observe the
evolution of the system as the simulation progresses. We note that the graphical user
interface (GUI) that implements this capability is distinct from the molecular simulation
components, such that simulations can be constructed completely independently of
whether a GUI is involved. Still, the requirements of such a GUI do affect the design of
the simulation components, often dictating a strict separation of their capabilities that
requires a well structured and robust framework for how they interact.

Molecular simulation is a computationally intensive process, so in addition to
extensibility we also include computational performance as an important design
criterion. These two criteria are largely at odds with each other. We find, however, that
any design choices that do not impact the core calculation of intermolecular energies,
forces, etc. has a negligible effect on performance. Nevertheless, it is clear that we
could not fully optimize performance while adhering to a general object-oriented
design. As a compromise, the design leaves open the possibility of some “hacks” that
break the object orientation in a limited way to enable a more streamlined calculation of
potential energies and forces.

The remainder of this paper is organized as follows. In the next section, we
briefly review some of the key concepts related to object-oriented programming that
appear frequently in the subsequent discussion. We also detail two object-oriented
design patterns that are used repeated by etomica to facilitate its modularity. In Sec. 3
we survey the major top-level components that form the etomica framework. These
constructs are general and fundamental enough that they could be used to develop a
molecular simulation programming interface independent of etomica. In Sec. 4, we
describe a few details of interest that are specific to our etomica implementation, and we
present some code examples that demonstrate how etomica can be used to construct different types of simulation. We offer some concluding remarks in Sec. 5.

2 Definitions and Useful Constructs

2.1 Definitions and Examples

We briefly introduce here some of the common terminology employed in object-oriented programming, and Java in particular, focusing on those terms appearing often in the discussion below. Some example Java code is given in Figures 1 to 3 to illustrate the concepts. In the following, we will reference these examples with square brackets enclosing names and line numbers [name/lineNumber(s)]. The classes/interfaces IVector, Vector3D, Space3D, and Atom are listed in Fig. 1, and GibbsEnsemble and HSMD2D are given in Figs. 2 and 3, respectively. The code examples in Fig. 1 do not represent the so-named complete classes or interfaces defined in etomica, and furthermore they are modified in small ways for the sake of this example. The classes in Figs. 2 and 3 show working simulations that are constructed using the etomica library; these are discussed further in Sec. 4.

An object is a general programming construct defined by a collection of fields (variables) [Atom/9-12] and methods (functions and subroutines) [Vector3D/10-20]. A class refers to a specific definition of an object, having a well defined set of fields and methods [Atom; Vector3D]. Once a class is written, or coded, a developer makes use of it by creating an instance of it [Space3D/5, GibbsEnsemble/47] and invoking its methods [GibbsEnsemble/48]. From this standpoint, the only important feature of a class is the set of methods that it presents to the developer, which taken together form an interface. Thus it is quite common to refer to the interface itself as an abstract entity, such that one can define interfaces alone as abstract classes without specifying the
implementation [IVector]; we follow a convention in which the names of these interfaces are prefixed by the letter “I”. Then a class written to include the methods specified by the interface is said to implement the interface [Vector3D/4], and may be used anywhere an object with that interface is expected [Atom/18]. By using different implementations of an interface, different behaviors can be realized [HSMD2D/44 versus GibbsEnsemble/51, for molecular dynamics versus Monte Carlo simulation].

For example, IVector is an interface defined in etomica, and it has a method (among others) that returns its magnitude [IVector/8]. Vector3D implements this interface by defining three coordinate fields [Vector3D/7], which are used and accessed by appropriate versions of the interface methods [Vector3D/10-20]. When a class is instantiated, memory elements are set aside to hold values of the fields defined for that class; different instances of a class [HSMD2D/60-61] have independent memory elements and in general have different values for their fields. A particular instance of a class is referenced by a field name associated with another class (e.g., the Atom class has a field named “position” that references an instance of a Vector class). A colloquialism is to refer to the fields referencing an instance of a class as a handle; a given instance of a class can have multiple handles referencing it, and often it is quite convenient to be able to do this. In addition to referencing instances of other classes [Atom, 9-11], fields can reference primitive values [Atom/12], which are the familiar integer, floating-point, or boolean variables encountered in most programming languages.

A subclass of a given class extends its definition by (optionally) adding new fields and methods, perhaps overriding the definition of its methods to give a different behavior. Otherwise the subclass inherits the fields and methods of the parent class, and an instance of it can be treated as any other instance of the parent. This ability of a class
to exhibit different behaviors via subclassing is referred to as *polymorphism*. We typically define new simulations by subclassing Simulation, and use the constructor subclass to construct and assemble the simulation components [HSMD2D, GibbsEnsemble].

It is considered good practice to *encapsulate* the fields of a class, so that they cannot be manipulated directly; instead they are accessed or changed using methods defined for this purpose [Atom, 22-30]. Accordingly the fields themselves are not usually included as part of the definition of the interface. A class can be defined to implement one or more interfaces, which requires it then include each of the methods in all the interfaces it implements. A large group of classes and their interfaces, taken together, constitute a programming interface, and the implementation of these interfaces form an object library, such as that described in this paper.

In what follows, our convention is to capitalize class names, *e.g.*, Potential. Classes that implement a given interface element are given a compound name formed from the interface name with a descriptive suffix (or suffixes). The names are formatted using the “CamelCase” convention often used in Java programming, *e.g.*, PotentialSoft.

In object-oriented software development, there are certain types of problems or needs that arise regularly, and a set of standard, reusable design patterns have been formulated to address them. Two such patterns are of particular value in the design of *etomica*, and we describe them here before proceeding with a survey of the major classes.

### 2.2 Agents

It is customary to define object classes such that quantities that vary from one instance of the class to another are incorporated as fields in the definition of the class. We have found that this practice is not always feasible when developing an extensible
and highly interoperable set of components for molecular simulation. The problem is particularly acute when dealing with the class representing an atom, and it arises to a lesser degree for some other simulation elements. Atoms will always require position coordinates in their definition, but only sometimes will it be necessary to associate a velocity, or a force, or an orientation, etc., with each atom. It is possible to develop atom subclasses that include such fields, and use them when needed, but the number of combinations required to cover all reasonable situations just explodes. To remedy this problem, we use an Agent design construct.

An Agent is an object that holds information particular to an instance of a class. For the most part, it is just a data construct, which does not itself perform any non-trivial operations and has no interface. As an object though, it can be designed to hold any desired set of data. The task of identifying (providing a handle to) the instance of the Agent that is associated with the corresponding instance of the object is supplements is performed by an AgentManager. Client objects that need to use an AgentManager will implement an interface that returns a new Agent (an object of the client’s design), which the AgentManager will use to associate a unique Agent instance with each instance of the target class. For example, a class that implements a molecular dynamics integration could define an Agent class with fields to hold the force and previous-step position of an atom. It can then access this information as it processes the atom by passing the Atom instance to the AgentManager, which will return the Agent instance that holds the corresponding information for that atom. The AgentManager is needed to ensure this all works in the presence of complicating circumstances, such as when the number of atoms changes (as in a grand-canonical simulation). The overall design is summarized in Figure 4, and does not follow an established design pattern.
Different clients each define their own Agent classes, and instantiate their own AgentManager to associate the information important to them with each instance of the target (e.g. each Atom). Different clients that need to share the same supplemental information can do so by sharing (each holding a handle to) the same AgentManager instance.

2.3 Events and Listeners

Agents provide a means to meet the data needs of an *ad hoc* assembly of simulation components – it permits objects to associate data with other objects that were otherwise not designed to hold that data. Likewise, we need a means for such an assembly of components to do their tasks in a coordinated fashion. For example, any time a MC trial or MD time step is completed, objects that are accumulating averages of properties may want to increment their sums; if a molecule is added or removed from a box, an object responsible for maintaining neighbor lists will need to update its data. The number of average-accumulators, or the presence of a neighbor-listing object, are both highly variable from one type of simulation to another. We need a means to enable these interactions for any type of collection of objects. The approach we adopt to enable this
capability is based on an event model, which follows the Observer design pattern. Events are usually considered in the context of a GUI, e.g., clicking a button is an event that the program must handle, but events are useful also as a means to coordinate the behavior of simulation components.

An event source can be any object that might do something that requires a response or action of another object, such as the integrator that moves the atoms around, or the box that holds and oversees addition and removal of atoms. Such objects have methods that allow other objects to register (or de-register) themselves as listeners to the event-source object. Listener objects must implement an interface that includes a method that is called by the event source when it fires an event. In simple terms, this means that the event source iterates through its list of registered listeners, and calls the method that each is required to implement as a listener. The method takes an Event object, which contains information particular to that type of event, e.g., if an atom is added to the box, the Event instance will have a field specifying which atom it is. The listener responds to the event in whatever manner is appropriate, e.g., updating its list of atom neighbors. The event model is summarized in Figure 5.
3 \hspace{1em} \textbf{Top-Level Components of Etomica}

In the following subsections we present the key components of \textit{etomica}.

3.1 \hspace{1em} \textbf{Simulation}

The Simulation class provides methods for accessing instances of the other components used to construct a molecular simulation. As such, it forms a common point of reference for all the components. A molecular simulation is represented in its entirety by a single instance of Simulation, and there is no need or ability for different Simulation instances to interact with each other.

Typically one defines a simulation by writing a class that implements Simulation, and codes in it the instantiation and assembly of the other components described below. For example, we have a class named HSMD2D that conducts molecular dynamics of the hard-sphere model in a 2D space (Fig. 3). To run this simulation, the user creates an instance of this class and invokes a method in the Controller (discussed below) that launches the simulation process. The instance could be created and launched at the command line, or instead it could be embedded in a GUI that adds capabilities to launch, monitor and interact with the simulation on the fly. Alternatively, within this framework one could develop a GUI that permits the user to construct a simulation interactively by adding desired components to a Simulation instance, and launch it via a button click.

In an object-oriented framework, there should be no need to define an auxiliary scripting language that specifies the sequence of activities performed by a simulation. Such a language will, necessarily, provide only a subset of the capabilities enabled by the programming interface, and it must be modified if it is to recognize any extension of
the interface. Instead, object-oriented code that serves the purpose of the scripts seen in some molecular simulation programming frameworks can be implemented in several places, including the constructor of a Simulation subclass [GibbsEnsemble, HSMD2D], as a reusable Action or Activity class, or in the main method of the class that launches the simulation. Such an approach is much more robust than a scripting language, because it can access the entirety of the object library, as well as native constructs offered by the programming language itself (*i.e.*, loops, conditionals, *etc*.).

### 3.2 Space

The most elementary feature of a simulation is the physical space on which the simulated system is defined—for example, whether it is 2-D or 3-D. Particle movements are performed, and interactions and properties are computed, via manipulation of particle coordinates defined by the space. Consequently, this basic data structure permeates the simulation. To avoid having to write versions of the *etomica* components for each type of space, the coordinate manipulations are encapsulated in Vector and related objects having interfaces that implement the manipulations. Thus we can add, subtract, scale, normalize, *etc*. coordinate vectors via general method calls, so that most of the *etomica* components can be written without needing to include space-specific code. The Space interface defines a factory with methods that return new instances of Vector *etc*. as needed to define particle coordinates (as well as other space-dependent quantities such as the boundaries of the simulation volume). Then in many cases one can select whether a simulation is performed in 2D or 3D simply by constructing it with an instance of a different Space class, and changing nothing else.
3.3 Atom, Molecule, and Species

It is natural to define an Atom class to represent each of the elementary particles that are manipulated in performing the simulation. While the “Atom” moniker may not be appropriate for all applications (e.g., the Atom instance might actually represent a colloid particle, or a magnetic spin), the interface does not preclude classes from representing such a variety of physical systems. The Atom interface presents a method to obtain the particle’s coordinate, which is a vector constructed by the Space. The coordinate can be interpreted as needed, so it could for example represent a spatial position, or a spin state, or some other degree of freedom appearing in the Hamiltonian.

Subclasses of Atom are defined to introduce dynamical and/or orientational coordinates; this yields four variants in all. Alternatively, we could have a single Atom class with a coordinate of arbitrary form that is defined via the Agent construct, but we felt that these cases were important enough to define them specifically in the interface.

While an Atom’s coordinate will be unique to each instance, often there is information about the Atom that is common to it and many other instances. This might include information such the atom’s mass, moment of inertia, chemical identity, etc., as appropriate to the model and the type of simulation. Rather than store this redundant information with each Atom instance, it is instead held by a single instance of an AtomType class, and a handle to this instance is held by each Atom of that type (similar to the Flyweight design pattern).

It is also natural to aggregate Atom instances into groups that are held by objects implementing the Molecule interface. This is useful because there is often need to reference molecules in a simulation, such as to rotate them, to compute their interaction energies, or insert and delete them in a grand-canonical simulation. Thus the interfaces for Atom and Molecule include methods that return references to each other.
One could go further and define other hierarchical elements, aggregating atoms into amino acids, for example, which are then assembled to form larger molecules, but we have not found this to be an essential part of the core programming interface. Such classes could be added to the library, and this would be useful if there were other simulation components developed to operate on them specifically.

Molecules are constructed by objects that implement the Species interface, following the Factory Method design pattern.\(^2\) To do this, the Species object constructs an appropriate number of Atoms for the Molecule, and arranges them in a nominal conformation. The Species defines the type of coordinate associated with each atom when it constructs the Atom. Molecule instances contain a reference to the Species that constructed them, which is needed when determining how they interact with other molecules. Likewise the AtomType referenced by each Atom instance can be used to identify the Potential (see below) that governs its interaction with other Atom instances. We note that information about bonding is not stored with the Atom or Molecule; instead the effect of bonding is realized via the Potential, and/or by the type of moves that are applied to the molecule (\textit{e.g.}, rigid bonds are implemented by performing only rigid rotations of the molecule and no independent displacements of its atoms).

AgentManagers are defined for Atom and Molecule classes, as described in Sec. 2.2, to allow additional information to be associated with instances of each. The Atom and Molecule classes each include index fields that can be accessed by AgentManagers to facilitate rapid identification of the instance of the Agent associated with a given Atom or Molecule instance.

3.4 \textit{Box and Boundary}

Molecules are collected within a Box object, which serves three functions. First, it provides a point of reference for Molecule instances that are subject to interaction with
each other. There may be multiple Box instances within a single simulation (e.g., to conduct Gibbs-ensemble\(^7\) [GibbsEnsemble/69,84] or Gibbs-Duhem integration\(^8\) simulations, or to perform parallel tempering\(^9,10\), but only those Molecule instances in the same instance of a Box will be interacting. Thus, requests to calculate forces, energies, \textit{etc.} make reference to the Box, to specify which molecules or atoms are the basis for the calculation.

Second, the Box is responsible for defining the size and shape of the simulation volume, and specifying what happens at its boundaries (e.g., central imaging and nearest imaging for periodic boundaries). These responsibilities are delegated to a Boundary object, which has methods to perform all of these functions. Thus, different boundary conditions or simulation shapes can be implemented by constructing an appropriate Boundary object, and giving it to the Box.

Finally, the Box provides the infrastructure to permit Molecule instances to be added to or removed from the simulation volume. It manages the indexes assigned to the Molecules and Atoms, and it holds an EventManager that allows interested listeners to be notified whenever the number of molecules in the box changes.

Additional information can be associated with each Box instance through the use of Agents that are managed by a BoxAgentManager. These are employed, for example, when implementing neighbor lists for more efficient calculation of energies and forces—an object that manages the neighbor lists is constructed for each Box and associated with it via the BoxAgentManager.

\subsection{3.5 Potentials and Iteration}

The potential quantifies the interactions of the atoms and the molecules, and it is the most challenging element to design and implement in a way that applies across the broad range of molecular simulations that we aim to include in this development. It
must handle calculations of interest to both hard and soft potentials, pairwise or higher-order interactions, Monte Carlo or molecular dynamics, and intra- and inter-molecular contributions.

The approach we take to this is superficially quite simple to use, but it achieves this simplicity of use at the cost of some complexity in implementation. The root of all calculations related to the potentials is the PotentialMaster class. During the construction of the simulation, instances of other Potential classes that define atomic or molecular potentials are added to the PotentialMaster, with a specification of the atoms to which they apply [HSMD2D/74-76, GibbsEnsemble/114-115]. The Potential instances can indicate whether they are 1-body, 2-body, etc. forms, which the PotentialMaster uses to determine how many Atom or Molecule instances to give it when performing a calculation. The Potential can represent the actual implementation of the model potential, or it may hold a group of potentials that define, for example, all the intramolecular interactions in a molecule. All Potential classes have a method that returns the energy, and beyond this there are different sub-interfaces that can be combined to supplement the Potential interface for the various cases that arise. For example, there is a soft-potential interface that requires the Potential to have methods for computing the gradient, enabling evaluation of forces and the virial; alternatively, there is a hard-potential interface that specifies methods for the time to collision assuming a ballistic trajectory, and that implements impulsive collision dynamics.

The key method in PotentialMaster for doing calculations involving potentials takes three pieces of information: (1) a Box, thus specifying which set of molecules are of interest; (2) an IteratorDirective, which encapsulates the specification of which atoms in the Box are subject to the calculation; and (3) a PotentialCalculation, which is a callback that encodes the calculation to be performed on the specified atoms (following
the Visitor design pattern\(^2\), and which holds the result of the calculation when the
method returns. Details follow. We have a few different versions of PotentialMaster
which implement this method in different ways, allowing for the use of more efficient
algorithms for some common special cases (such as monatomic molecules), and for
implementing neighbor lists.

The IteratorDirective may specify that the calculation is to be performed on all
Molecule instances in the Box; alternatively it may specify a target molecule on which
intramolecular calculations are performed, or for which intermolecular interactions are
summed over other molecules (needed, for example, to process a Monte Carlo molecule
displacement). Iteration can be implemented within the PotentialMaster via one of
several iterator classes, or via a hard-coded loop. In either case, the IteratorDirective
specifies which Molecule and Atom instances (or pairs, or high-order groups) are to be
generated during the iteration and passed in sequence to the PotentialCalculation. A
PotentialMaster class might iterate over atoms based on neighbor or cell lists, but only it
needs to know about this detail, thus different neighbor listing schemes can be
introduced while not affecting the rest of the code.

The PotentialCalculation interface defines a single method that takes a group of
Atom instances, and a Potential instance. From here, an implementation can do anything
that is required by the simulation: add to an energy and/or force sum, compute a
collision time, \textit{etc}. The method is called by the PotentialMaster as it loops through atom
and molecule groups in accord with the IteratorDirective. Sums computed over the
iterates can be accumulated in and obtained from the PotentialCalculation upon
completion; alternatively, the PotentialCalculation can hold an AtomAgentManager that
it uses to store appropriate information (\textit{e.g.} force sums) with each Atom instance (see
Figure 4).
We have not attempted a comprehensive implementation of potentials in \textit{etomica}, and in particular we have not yet attempted implementation of popular transferable potentials such as CHARMM, AMBER, and TraPPE. Most of the potentials in \textit{etomica} have been developed as needed, and encompass simple hard and soft potentials, and a few realistic potentials of small inorganics and alkanes, including polarizable models of water.

3.6 \textit{Integrator and Controller, Action and Activity}

We separate the driver of the simulation process from the algorithm that generates the configurations. The former is managed by a Controller object, and the latter is coded in an Integrator. The Controller holds a set of Action objects, each of which performs a task required of the overall simulation. An Action completes some elementary function, such as placing all the atoms on a lattice, or zeroing out accumulators. Action classes that perform lengthy processes are implemented in an Activity, which is a subclass of Action. An Activity includes methods that accommodate user interaction, meaning that it can be paused, restarted, or stopped before completing naturally. An ActivityIntegrate class, for example, is defined to perform a loop for a given number of iterations, repeatedly invoking a method in an Integrator instance to move the atoms according to its integration algorithm.

To allow operation in an interactive environment, careful attention must be paid to issues of threading. A thread is a process managed by the operating system, and different threads can operate in (what appears to be) a simultaneous fashion to complete tasks in parallel. The activities of a typical simulation naturally run on a single thread, but to accommodate user interactivity, each simulation process (\textit{i.e.}, Activity) has to run on a dedicated thread, so that the Controller remains unoccupied and able to respond to user requests while it waits for the activity to complete. In contrast, an Action runs on
the Controller’s thread, and thus cannot be interrupted before completing. Thread behaviors must be carefully synchronized to ensure that external requests to pause an Activity do not interrupt it at an awkward point (e.g., halfway through the Integrator’s process of moving all the atoms to their new time-step positions); thus a pause is always “requested”, rather than imposed, and the Activity heeds the request once it can safely do so. The Controller keeps a queue of pending Actions, and when it completes them it exits, but it may be restarted if Actions are added (e.g., via the GUI). It is possible also to invoke an Action directly by the GUI, if the Action is tied to a GUI button or other interface widget. However, it is not a good practice to do this while the simulation is running, and instead such actions are implemented by passing them to the Controller, and requesting it to do them as soon as it is safe to interrupt the integration activity.

Movement of atoms according to a stochastic or dynamical algorithm is performed by the Integrator. The usefulness of polymorphism to implement different behaviors is very clear in this context. Different Integrator subclasses are developed to perform specific integration algorithms; examples\textsuperscript{11,12} include Metropolis MC sampling, velocity Verlet continuous MD, hard-particle discontinuous MD, Brownian dynamics, dissipative particle dynamics, and so on. Implementation of each algorithm in a simulation is accomplished simply by instantiating the desired type of Integrator, and adding it to and ActivityIntegrate instance, which is then at some point run by the Controller.

The MC Integrator (named IntegratorMC) deserves special mention. One of the most appealing features of the MC method is the ease with which it can be applied to sample any ensemble, and incorporate many types of trial moves. It is not feasible to implement all these variations via polymorphism, \textit{i.e.} by subclassing IntegratorMC. Instead, we use composition (as it is called in object-oriented parlance): we add
MCMove objects to IntegratorMC, each of which performs a particular type of trial. Isobaric simulations are implemented, for example, by adding a MCMove instance that performs perturbation of the volume. Fields appropriate to an ensemble are introduced with the MCMove that enables it (e.g., a volume-changing MCMove will include a field that specifies the pressure). Moves can be introduced also to improve sampling, such as configurational bias when simulating chain molecules.

The IntegratorMC and its MCMove instances have a close interaction. IntegratorMC selects which move to perform (randomly, but with weight to give a desired distribution of trials), tells it to perform its trial, and receives information that it uses to determine whether to accept the trial. It then notifies the MCMove of the outcome (so it can, for example, restore the original state if the trial is rejected). Each MCMove tracks its acceptance rate, and can adjust its trial steps sizes if permitted (e.g., production trials are not yet started).

Many different objects will be interested in the progress of the Controller and the Integrator, and we use the event model to keep them updated. The Controller will fire an event at various milestones in its processing of actions: when it begins its processing, then at the start and end of each action, and when it is either halted or it empties its queue of pending actions. The Integrator fires an event when it starts and completes a MC trial or MD time step, while specific subclasses of Integrator can be designed to fire additional events as appropriate.

3.7 Data flows

The last major element to define is the framework for taking and processing property data as configurations are generated during the simulation. We have established a pipeline-based system to permit data processing to be done in a flexible and extensible manner. This is illustrated schematically in Figure 6. All data are encapsulated in Data
objects, and the flow of data is realized by the passing of an instance of Data from one
element in the pipeline to the next. Data objects vary in the type of data that they hold,
and include methods to perform simple mathematical operations on them. Encapsulated
data may be something as simple as a scalar value, or an array of values, a set of arrays,
or some other compound structure of data elements. Metadata can be associated with
the Data in a DataInfo object, characterizing the data with descriptive text labels, its
physical dimensions (e.g., length, mass, etc.), and so on, to aid in processing or display.
The DataInfo is passed down the pipeline before the data stream begins, and again any
time significant changes are made to the metadata (e.g., via some action by the user). It
also provides information about the type of data that will follow, such as whether it is a
scalar or an array, to permit the processing elements to initialize themselves
appropriately.

Data processing starts with a DataSource object, which codes the calculations needed to
calculate a property, such as the internal energy, diffusion constant, radial distribution
function, free energy, and so on. The impetus to perform the calculation is provided by
a DataPump, which accesses a method (getData) in the DataSource to extract a Data
instance from it. The DataPump takes this instance and passes it on to a DataSink,
which has a method (putData) to receive a Data instance. A DataProcessor is a
DataSink that is configured to perform some calculation on or using the data (e.g., to
accumulate averages and error statistics), before passing it (perhaps intermittently)
down to one or more other DataSink instances (which might process it further, or
display the data to screen, or write it to file, etc.). Retrieval of data by the DataPump is
typically triggered by an IntegratorEvent, via a listener registered with the integrator
and configured to invoke action from the DataPump. DataPump in fact implements the
Action interface, so alternatively it can be configured to respond to a button click on a
GUI, for example to take data at user-directed times.

Any number of data flows of the type described here can be configured in a single simulation. A great variety of data sources can be developed, and each data pipeline running from it can have any number of processing elements, and can forked as much as necessary. In fact, given an appropriate user interface, pipelines or pipeline elements can even be added or removed interactively, even as the simulation proceeds. Thus the overall structure provides a flexible and highly extensible framework for data generation and processing. An example is given in lines 92-103 of HSMD2D, Fig. 3.
4 Etomica Implementation

We present this molecular simulation object library without emphasizing issues related to its implementation, with the idea that the general structure described here could be implemented in whole or in part by other developers, perhaps in a language other than Java. The choice of simulation elements and the manner in which they interact are the key points of the design, and the details of how this is coded will be strongly dependent on the programming language used. The overall design can be realized in any object-oriented programming language, and could be approximated or used to guide the structure of a simulation framework programmed in a non-object languages. Still, although the focus here is not on such details, it is nevertheless of interest to discuss briefly our experience in implementing and applying the interface, if just to demonstrate its suitability for its designed purpose. We also discuss the two examples listed in Figs. 2 and 3, showing how etomica is used to construct different molecular simulations using a common set of library components.

Etomica\textsuperscript{13,14} has been used by us for over 15 years, in a wide variety of applications. In one direction, it was used as the basis for developing a dozen or so graphically-oriented molecular simulation modules appropriate for use in education.\textsuperscript{15,16} Each module was developed in collaboration with a member of the science and engineering community, and designed to convey understanding of the molecular basis for some macroscopic behavior important to engineering and science. Phenomena demonstrated include: heat and work, equations of state, reaction equilibrium, osmosis, polymerization kinetics, chemical potential, adsorption, surfactants, and more. The simulations are interactive, and most are based on discontinuous potentials (e.g. square well) to enable fast dynamics. The simulations provide quantitative data that can be analyzed using the formalism, theories, and models that undergraduates learn about in their courses. The
hope is that in applying techniques they learn in the classroom to the analysis of
simulation-based behavior, students will develop a better appreciation for the
underlying molecular origins of the behaviors being studied.

The etomica library has also been used as the basis for most of the research
activities performed in our group, and naturally the classes developed for it are oriented
toward our research applications. Much of our recent interest has been in calculation of
cluster integrals using Mayer sampling Monte Carlo methods. This approach has been
inspired by free energy calculations, but the structure of the simulation is very much
unlike the way a conventional molecular simulation is designed. Nevertheless, we have
found that the etomica codes, which were not originally designed for this application,
work quite well as a basis for them. In other directions, we have used etomica to
perform calculations on a variety of small-molecule models, including discontinuous
potentials, polarizable and non-polarizable water models, ab initio based 2- and 3-body
models, and large organic molecules. Experience has shown that it can serve as the
basis to implement a broad variety of simulation techniques, including MC in any
ensemble, NVE or NVT MD, Gibbs ensemble MC, free-energy methods, parallel
tempering, path integral methods, biasing techniques, and more.

To demonstrate, in Figures 2 and 3 we present annotated Java code that
constructs two different molecular simulations using our implementation of the MSPI.
The first is a Gibbs-ensemble Monte Carlo (GEMC) simulation of the Lennard-Jones
model. The GEMC method performs a simulation of two systems in such a way that
upon equilibration they describe the conditions found in coexisting thermodynamic
phases. This example demonstrates how simulations can be developed using multiple
boxes. The second example demonstrates the simulation of a mixture of 2-dimensional
hard spheres via discontinuous molecular dynamics. This example shows how multiple
species and interaction potentials can be specified, and it also demonstrates a data flow
and illustrates how easy it is to specify the dimension of the space in which the
simulation is conducted. Although the types of simulations demonstrated with these
examples are quite different from each other, there is very much overlap in the
components appearing in each implementation.

A more complete description of the etomica library can be found on the etomica
web site. Links found at these locations lead to documentation of many additional
components that we have implemented for specific applications.

5 Concluding Remarks

Even when using a comprehensive component library that follows a
programming interface, simulations written for new applications often require
development of new library components. Ideally, the coding focuses only on the
development of new classes to implement the new models or methods. What we want to
avoid in a new application is any need to rewrite existing code, or worse, redesign the
programming interface. In this respect, the etomica library has succeeded, at least in the
range of applications we have encountered, and more so in recent years, as it has
matured. Still, we feel that there is room for improvement, particularly with the design
related to Potentials. It can be somewhat tedious to program models using etomica for
applications to specific multiatomic molecular systems. Given the broad range of
application we set for the design of the Potential interface, perhaps such complexity is
unavoidable. As a possible remedy, it may be that we can build convenience classes that
present an easy-to-use interface for a specific category of molecules and/or simulation
algorithm (e.g., MD specifically), while encapsulating code that uses the etomica
elements to implement the model. We have not yet attempted this.
In summary, we have outlined an extensible programming interface for constructing molecular simulations from modular components. Such a framework is appealing because it facilitates re-use of coded elements, which simplifies the construction of new simulations while reducing programming errors. The framework also promotes the re-use of certain design patterns (e.g., Agent, Event), which can solve problems related to the interaction of the components. The structure provided by the etomica library also has pedagogical value, and can guide new users and new developers when learning the architecture and operation of a molecular simulation (even one of a different design). The goals of the framework described here put significant emphasis on generality of application, which steered the design toward a higher level of abstraction. Such generality will not be in the interest of all developers, but it is hoped that some of the design concepts outlined here find use, in at least a limited form, in new and existing molecular simulation code bases.

Acknowledgements

This work was conducted with support from the U.S. National Science Foundations, grants CHE-0626305 and DUE-0618521.
References

2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. (Addison-Wesley, New York, 1995).
Figure 1. Examples of object-oriented programming constructs as applied to components of a molecular simulation. The classes shown here demonstrate only part of the full set of fields and methods in the correspondingly-named classes in etomica. Dark blue and green lettering are code comments, maroon coloring indicates Java keywords; bright blue are field names.
import etomica.action.BoxImposePbc;

/**
 * This class demonstrates implementation of the Gibbs Ensemble method for evaluation
 * of phase equilibria by molecular simulation. It shows how Monte Carlo methods are
 * implemented and how multiple non-interacting Boxes can be used. The demonstration is
 * set up for a simple Lennard-Jones model system.
 */

// The Simulation superclass defines a few fields that organize the simulation elements.
public class GibbsEnsemble extends Simulation {

    // The constructor puts together the elements needed by the Simulation.
    // It takes a Space instance as an argument, allowing the simulation to
    // be constructed for use with an arbitrary dimensional space.
    public GibbsEnsemble(Space space) {
        super(space); // invoke the constructor of the superclass

        // We use a PotentialMaster that is specialized for monatomic molecules.
        IPotentialMaster potentialMaster = new PotentialMasterMonatomic(this);

        // We used a two-level set of Integrators. The one instantiated here handles
        // moves specific to the Gibbs ensemble (volume and molecule exchanges between
        // boxes). It controls an Integrator for each Box that determines the simulation
        // algorithm used to move molecules in the respective Box.
        IntegratorGEMC integrator = new IntegratorGEMC(getRandom(), space);

        // We use an Activity that is designed to advance the Integrator step by step
        ActivityIntegrate activityIntegrate = new ActivityIntegrate(integrator);

        // The Activity is added to the Controller, which in general oversees all Activity
        // instances (if more than one). The Controller has its own operating-system thread
        // on which it is run.
        getController().addAction(activityIntegrate);

        // We set a field that forces the activity to pause briefly between steps; this
        // is used for interactive simulations, so the GUI has time to catch up to the
        // simulation. It wouldn't be used for a production-type calculation whose only
        // aim is to generate results quickly.
        activityIntegrate.setSleepPeriod(1);

        // Now we specify the geometry of the molecules that are being simulated. Here it is
        // just monatomic spheres.
        SpeciesSpheresMono species = new SpeciesSpheresMono(this, space);
        addSpecies(species); // this is a method of the Simulation superclass

        // We create a Box for each of the phases being simulated.
        Box box1 = new Box(space);
        addBox(box1); // this is a method of the Simulation superclass
        box1.setNMolecules(species, 200); // create 200 monatomic spheres in this Box

        // We create a general Monte Carlo integrator to define the type of MC trials
        // performed in this Box
        IntegratorMC integratorMC1 = new IntegratorMC(this, potentialMaster); // construct
        integratorMC1.setBox(box1); // connect it to the Box
        integratorMC1.setTemperature(0.420); // set the temperature field for the Integrator

        // Here we add the moves that define the trials performed in this Box. There's only
        // one type of trial move, which performs simple molecule displacements
        integratorMC1.getMoveManager().addMCMove("random" is a field defined in Simulation
                      new MCMoveAtom(random, potentialMaster, space));
        integrator.addIntegrator(integratorMC1);
GibbsEnsemble.java

// We repeat all of the above, now for the second phase being simulated.
Box box2 = new Box(space);
addBox(box2);
box2.setNMolecules(species, 200);
IntegratorMC integratorMC2 = new IntegratorMC(this, potentialMaster);
integratorMC2.setBox(box2);
integratorMC2.setTemperature(0.420);
inegratorMC2.getMoveManager().addMCMove(
    new MCMoveAtom(random, potentialMaster, space));
inegrator.addIntegrator(integratorMC2);

// This sets up the initial configuration. It is "hard-wired" to handle only
// 2- or 3-dimensional systems, but this could be easily generalized by passing
// the configuration-generation class to the GibbsEnsemble constructor.
SpaceLattice lattice;
if (space.D() == 2) {
    lattice = new LatticeOrthorhombicHexagonal(space);
}
else {//expect this would be D = 3
    lattice = new LatticeCubicFcc(space);
}
ConfigurationLattice config = new ConfigurationLattice(lattice, space);
config.initializeCoordinates(box1);
config.initializeCoordinates(box2);

// Now we define the Potential that governs the interactions of our monatomic
// molecules.
P2LennardJones potential = new P2LennardJones(space);
potential.setSigma(1.2); //set LJ size parameter to something other than the default

// Specify which atom types the potential applies to, when adding to PotentialMaster
potentialMaster.addPotential(potential,
    new IAtomType[] {species.getLeafType(), species.getLeafType()});

// We specify the objects that want to be informed about progress with the
// Integrators. We would normally add objects needed for ensemble averaging,
// but we omit these for this example.

// These objects will enforce period boundary conditions in each box
integrateMC1.getEventManager().addListener(
    new IntegratorListenerAction(new BoxImposePbc(box1, space)));
integratorMC2.getEventManager().addListener(
    new IntegratorListenerAction(new BoxImposePbc(box2, space)));

// We initialize the density of the second box to make it more gas-like.
// This is done with an Action class that performs a volume change while
// scaling all the molecule positions accordingly. Another instance of this
// same Action class is used to implement volume-change moves in NPT MC simulations.
BoxInflate inflater = new BoxInflate(box2, space);
inflater.setTargetDensity(0.1);
inflater.actionPerformed();
}
public class HSMD2D extends Simulation {

    public HSMD2D() {
        super(Space2D.getInstance());

        PotentialMasterList potentialMaster = new PotentialMasterList(this, space);

        IntegratorHard integrator = new IntegratorHard(this, potentialMaster, space);
        integrator.setIsothermal(false);
        integrator.setTimeStep(0.01);

        double sigma = 1.0;
        potentialMaster.setRange(3.0*sigma);

        ActivityIntegrate activityIntegrate = new ActivityIntegrate(integrator);
        activityIntegrate.setSleepPeriod(1);
        getController().addAction(activityIntegrate);

        SpeciesSpheresMono species1 = new SpeciesSpheresMono(this, space);
        species1.setIsDynamic(true);//this is needed to ensure they have momentum fields
        SpeciesSpheresMono species2 = new SpeciesSpheresMono(this, space);
        species2.setIsDynamic(true);
        addSpecies(species1); //this makes the Simulation aware of the Species instance
        addSpecies(species2);

        P2SquareWell potential11 = new P2SquareWell(space);
        P2SquareWell potential22 = new P2SquareWell(space);
        P2HardSphere potential12 = new P2HardSphere(space);
        potential11.setCoreDiameter(sigma);
        potential22.setCoreDiameter(0.5*sigma);
        potential11.setLambda(1.5); //multipler for well diameter
        potential22.setLambda(1.5);
        potential12.setCollisionDiameter(0.75*sigma);

        IAtomType leafType1 = species1.getLeafType();
        IAtomType leafType2 = species2.getLeafType();
        potentialMaster.addPotential(potential11, new IAtomType[]{leafType1, leafType1});
        potentialMaster.addPotential(potential12, new IAtomType[]{leafType2, leafType2});
        potentialMaster.addPotential(potential22, new IAtomType[]{leafType1, leafType2});

        IBox box = new Box(space);
        box.setNMolecules(species1, 60); //number of species-1 molecules
box.setNMolecules(species2, 4); //number of species-2 molecules
integrator.getEventManager().addListner(potentialMaster.getNeighborManager(box));
new ConfigurationLattice(
   new LatticeOrthorhombicHexagonal(space), space).initializeCoordinates(box);
integrator.setBox(box);

//Set up a meter that measures the intermolecular potential energy, and
//put it in a data flow that records the average
//First define the meter
MeterPotentialEnergy energyMeter = new MeterPotentialEnergy(potentialMaster);
energyMeter.setBox(box);
//The accumulator receives data from the meter and averages it
AccumulatorAverage energyAccumulator = new AccumulatorAverageFixed();
//The data pump listens for events from the integrator and at fixed intervals
//will take a value from the meter and deliver it to the accumulator
DataPumpListener energyPump = new DataPumpListener(energyMeter, energyAccumulator);
energyAccumulator.setBlockSize(50); //this sets the interval between measurements
integrator.getEventManager().addListner(energyPump); //register pump as listener to integrator
//For this example we'll just route the accumulator average to the console
DataSinkConsole console = new DataSinkConsole();
energyAccumulator.addDataSink(console, new StatType[] {AccumulatorAverage.AVERAGE});

/**
 * Demonstrates how this class is used in the simplest way possible. The simulation
 * is instantiated and run in a batch (non-interactive) mode.
 */
public static void main(String[] args) {
    HSMD2D sim = new HSMD2D();
sim.getController().actionPerformed();
}

//final SimulationGraphic simGraphic = new SimulationGraphic(sim, SimulationGraphic.TABBED_PANE,
//   "TEST", sim.space, sim.getController());
//simGraphic.makeAndDisplayFrame();