MESHI: a new library of Java classes for molecular modeling

Supplementary Material

The supplementary material includes two sections:

1. A detailed description of selected MESHI classes and packages.
2. A comment about the performance of the MESHI application - Beautify - in the CASP6 experiment.

1. A detailed description of selected MESHI classes and packages

General: We demonstrate the modular and object-oriented design of MESHI through a detailed description of two central packages and their sub-packages. Each of these packages is presented from two angles: structural and functional. The structural point of view emphasizes the inheritance patterns of generic classes and the specialized classes that extend them. The functional view, on the other hand, emphasizes the interactions between different classes.
For clarity sake, not all the classes in the packages are presented and only few of the fields and methods of each class are mentioned. For a more complete picture see the MESHI API (http://www.cs.bgu.ac.il/~meshi/API).

Figure S1 introduces the graphical representation of the program components and their relations. This figure serves as legend for figures S2-5.

Figure S2 provides the structural view of the **molecularElements** package and its sub packages. The **molecularElements** package is built around three classes: **Atom**, **Residue** and **Protein**. In addition to the classes presented in the figure, the package also includes several specialized container classes such as **AtomList**.

Figure S3 demonstrates some of the interactions between objects in the **molecularElement** package. It emphasizes the important role of lists in organizing the low level elements (e.g. atoms) within high-level elements (e.g. proteins).

Figure S4 presents the structural view of the **energy** package and its sub-packages. In MESHI we made a considerable effort to standardize the energy terms in order to accelerate their design and implementation. The extensive use of abstract classes in the **energy** package serves two purposes:

1. It promotes code reuse. Generic procedures are written once, and are inherited and used by all the extending classes. For example, the "test" method of the **EnergyElement** class allows any energy term to verify its correct derivation.
2. It provides a standard interface for term-specific methods. For example, the abstract method "evaluate" in the **AbstractEnergy** class is implemented differently by different energy terms. Other classes, however, needs to know nothing about the specific implementation.

Figure S5 demonstrates how information flows from the lowest level, the X,Y,Z coordinates of atoms to the top level minimization algorithm LBFGS [Liu and Nocedal (1989)]. The **LBFGS** class implements this powerful minimization technique without "knowing" anything about either atoms or specific energy terms. It should be noted that the rather complex data structure presented in the figure is practically created with five code lines.

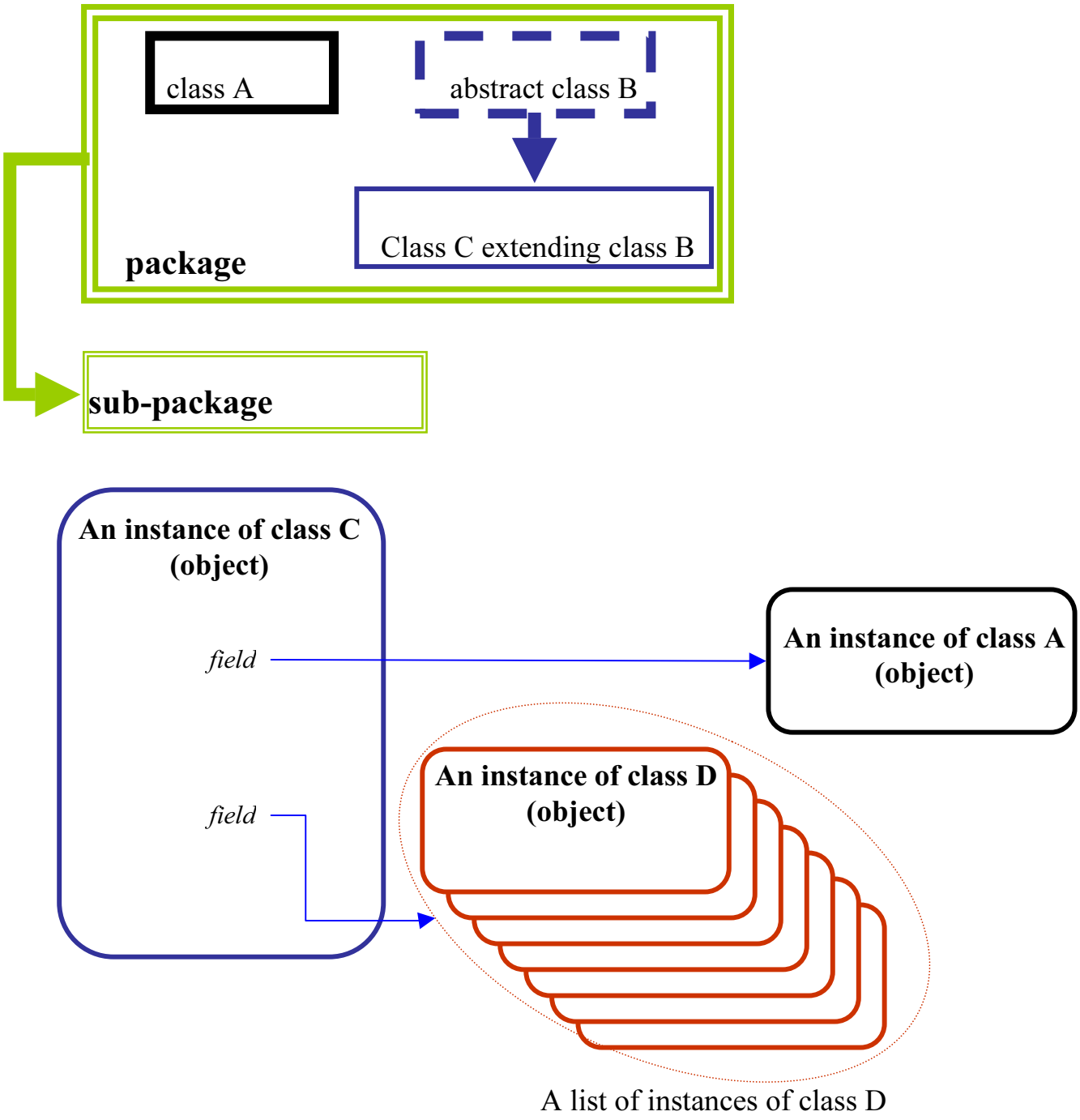

Figure S1: The graphical representation of the program components. Note the difference between classes and their instances (objects). Note also that wide arrows represent inheritance relation between classes, while thin arrows implies that a field of some object refers to (points at) another object.

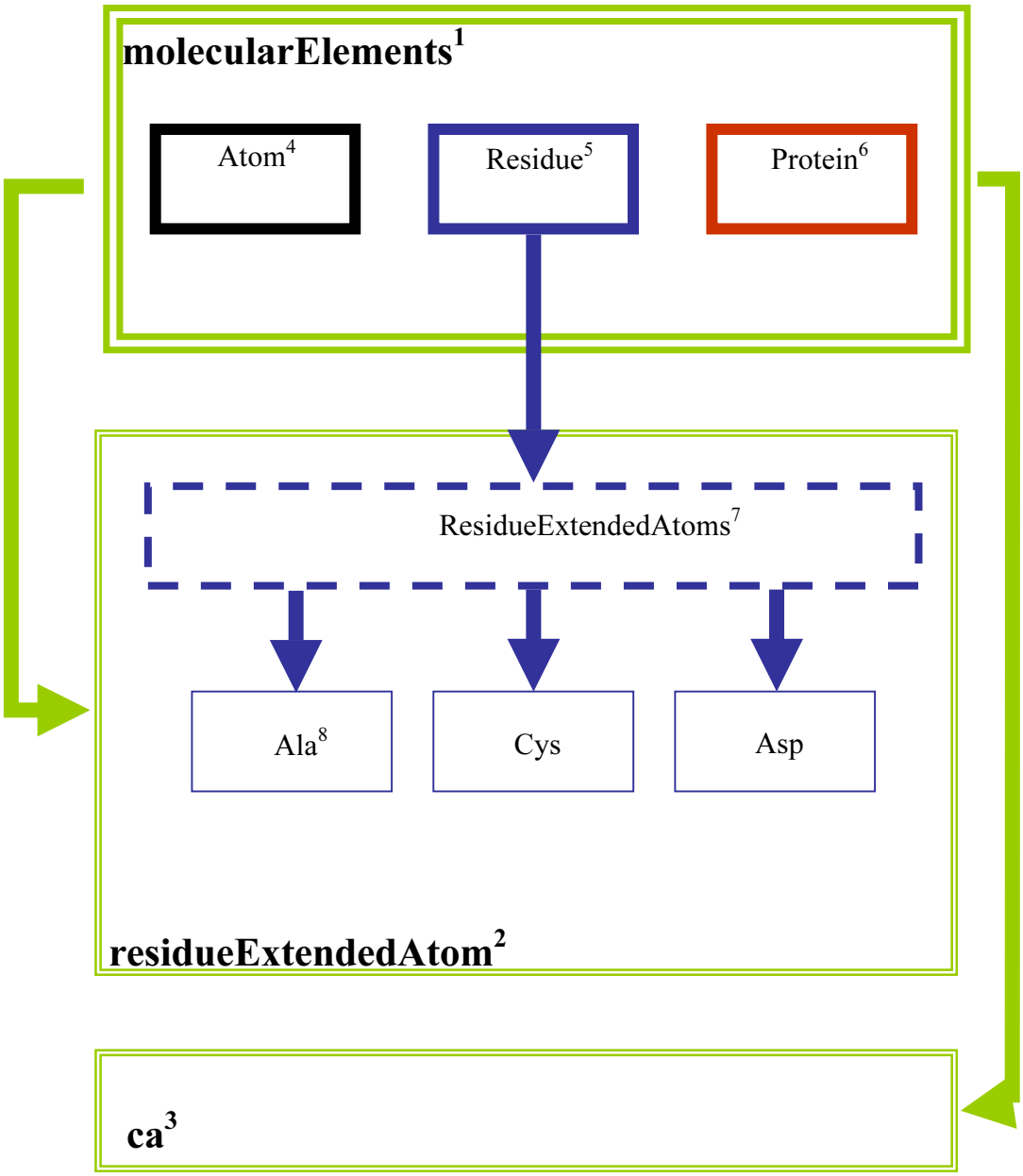

Figure S2: A structural view of the molecularElements Package and its sub-packages.
The **molecularElments** package includes generic classes, while its sub-packages include their model-specific extensions.

1. A generic package for molecular elements.
2. A specialized package for the extended atom molecular model. This model explicitly considers all heavy atoms and some of the polar hydrogen atoms (the ones that are well-localized). Other hydrogen atoms are swallowed by their heavy neighbors.
3. A specialized package for the Cα model of proteins.
4. The **Atom** class represents a general atom. Each Atom instance "knows" its position in space, its type, its name etc. That is, it has fields that refer to and methods that manipulate this information.
5. The **Residue** class represents a residue in a protein. Each Residue instance "knows" the list of its atoms, its name and position in the chain etc.
6. The **Protein** class represents a polypeptide chain. A Protein instance "knows" the lists of its residues, atoms, bonds etc.
7. The abstract class **ResidueExtendedAtoms** represents a model of amino acid residue. Thus, it has fields for the backbone atoms and the beta carbon and a list of the bonds that connect them.
8. The **Ala** class represents a specific residue type within the extended atoms model. It extends the **ResidueExtendedAtoms** adding some more information like the name and the fact that no additional atoms are required.

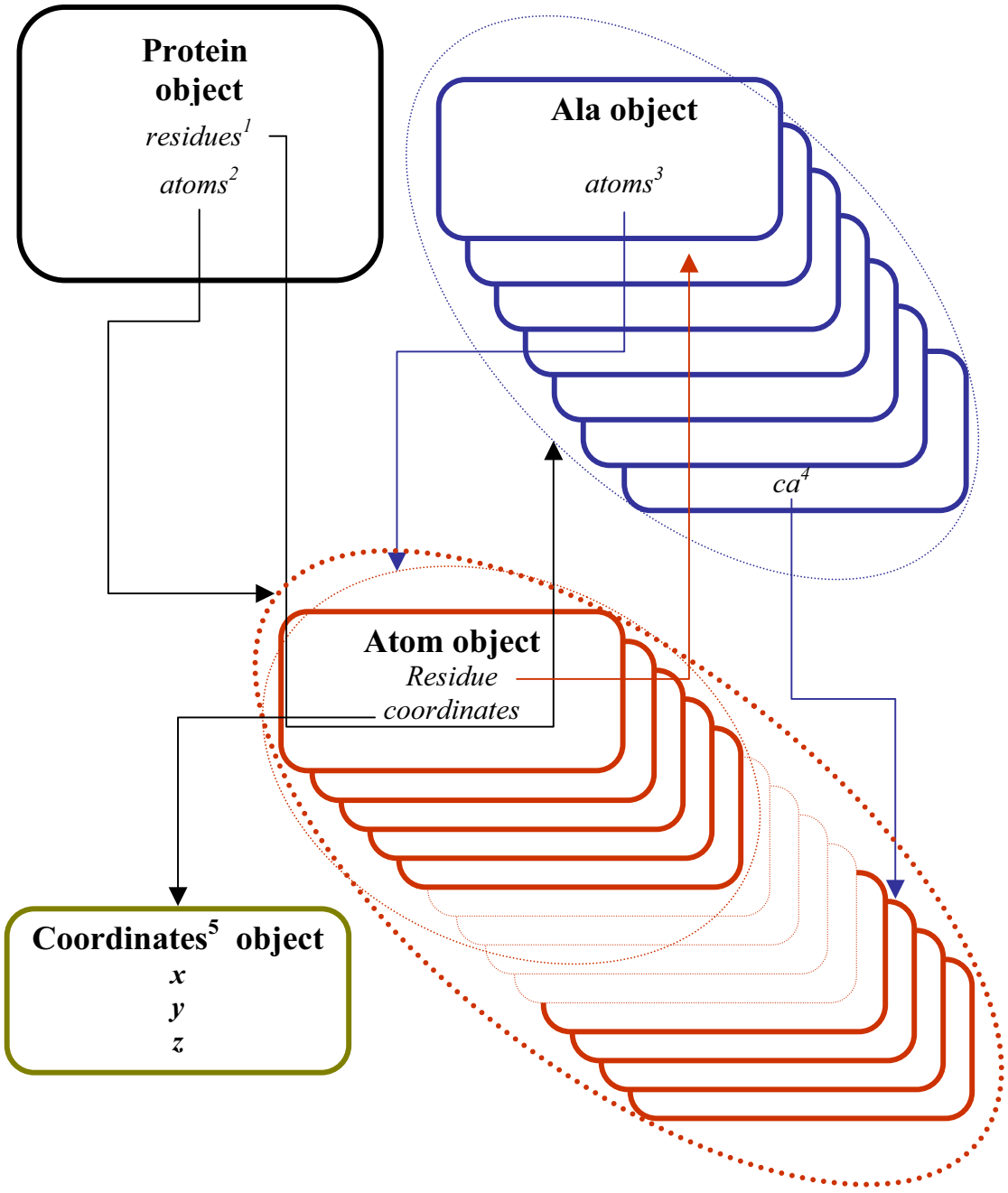

Figure S3: A functional view of the molecularElements Package and its sub-packages.
The various molecular elements constitute a tightly connected graph of references. For clarity sake, only representative objects and references (the nodes and edges of the graph respectively) are presented. Note, that every object may be referenced by quite a few other objects. The scheme is modular and the **Protein** object, for example, need not know about the implemented molecular model. Within an application, this complex data-structure may be created by a single line of code.

1. A reference to the list of the protein's residues.
2. A reference to the list of the protein's atoms.

3. A reference to the list of the residue's atoms. This field was inherited from the Residue class that Ala extends.
4. A reference to the Cα atom of this residue. Note that this field is meaningful only in certain molecular models and is not part of the generic Residue class.
5. The Coordinates class is defined in the geometry package. For each axis it stores and manipulates both the position and the force.

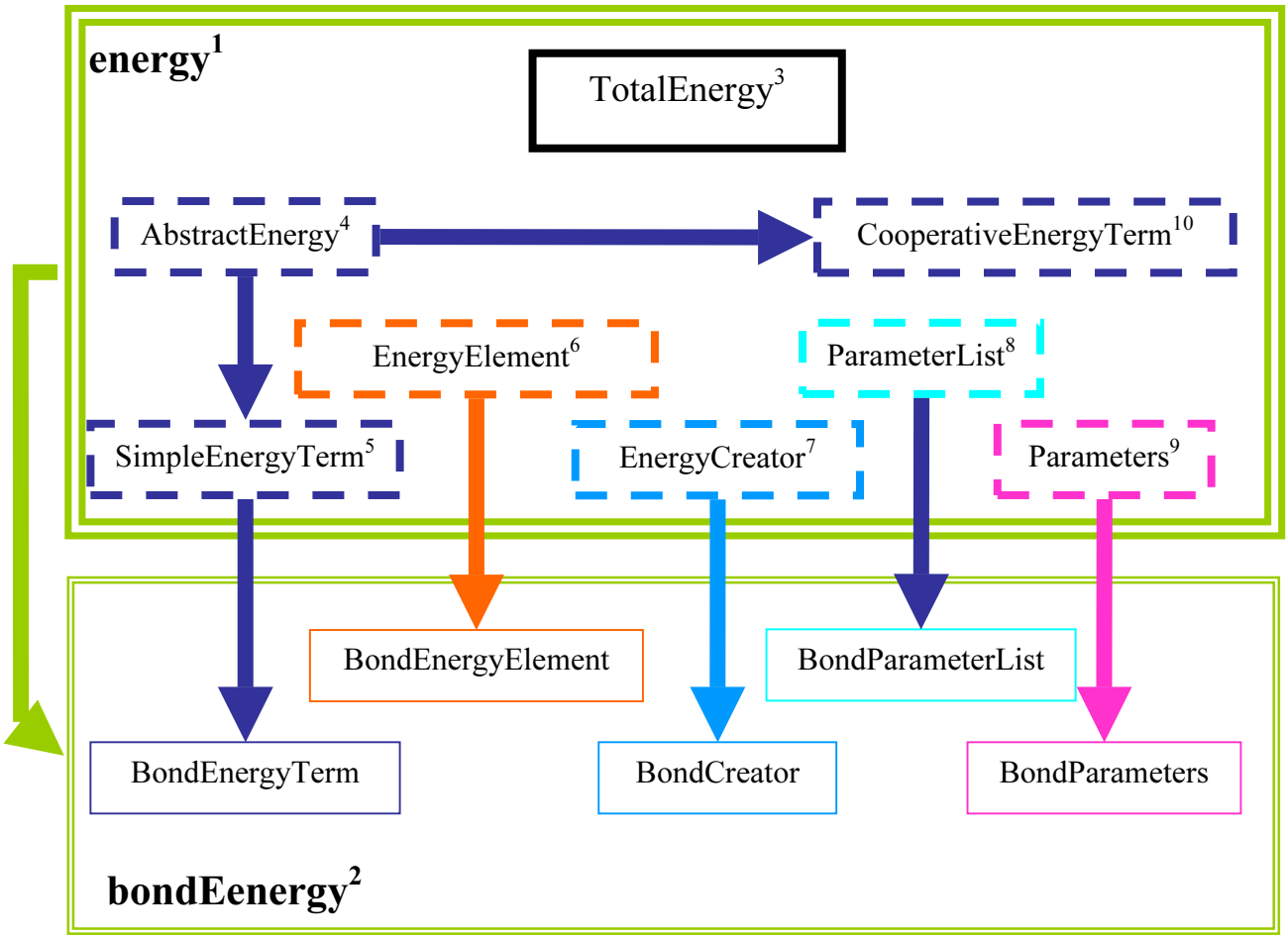

Figure S4: A structural view of the energy package and one of its sub-packages.

1. A package containing all the classes that are relevant to energy functions and are not term-specific.
2. A specific energy term (e.g. bond energy) is implemented within a designated package (in this case bond-energy). Term-specific classes extend the energy package classes.
3. The **TotalEnergy** class provides the optimization methods with a simple interface to the energy terms. It handles the creation, evaluation and summation of all the different energy terms.
4. The **AbstractEnergy** class handles all the generic aspects of the energy terms, such as activation/disactivation, report printing etc. Its abstract methods enforce standard interface for term-specific functions such as the energy term evaluation.
5. **SimpleEnergyTerm** is a more specific extension of **AbstractEnergy.** It handles energy terms that can be decomposed into a sum of multiple independent elementary energy terms (e.g. covalent bonds).
6. **AbstractEnergyElement** handles all aspects of an energy element (e.g. a single bond or a Lennard-Jones contact) that are not term-specific. Its abstract methods enforce standard interface for term-specific functions.
7. The **EnergyCreator** class provides a simple interface that hides the complexities of energy term formation. It encapsulates all the details of how to build an instance of an energy term. These details may include: searching for a parameters file, setting the term weight in the total energy, and the handling of any other required data. The **TotalEnergy** class uses an array of **EnergyCreator** instances to build its list of energy term without actually "knowing" anything about each specific term.
8. The **ParameterList** class is used to read multi-line files that hold a large set of parameters.
9. The **Parameters** class instances contains specific parameters for a single energy element.
10. **CooperativeEnergyTerm** is yet another extension **AbstractEnergy.** It handles those terms that cannot be decomposed to a sum of simple elements.

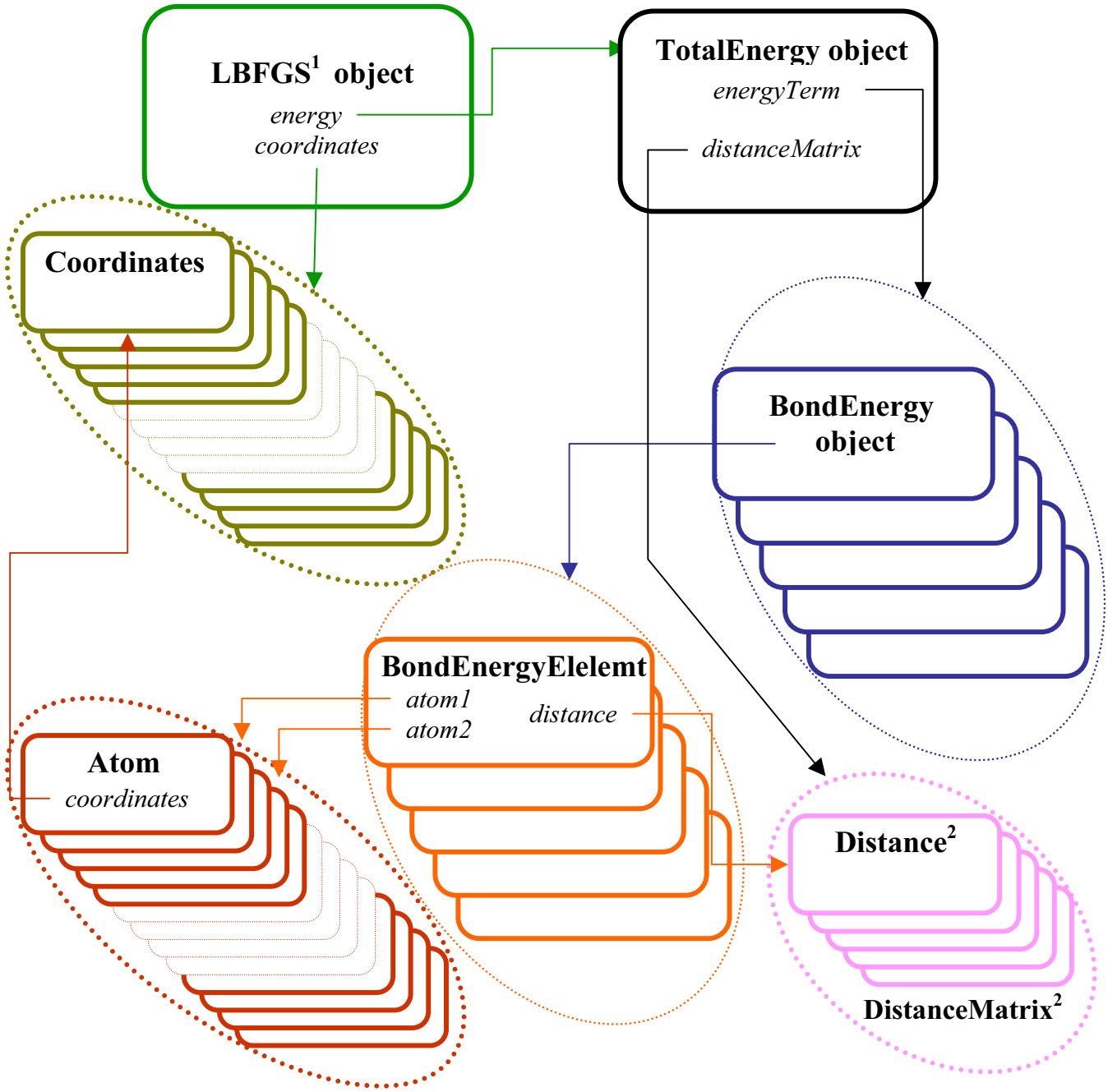

Figure S5: Modularity and abstraction in energy minimization.
The **LBFGS** class implements a powerful minimization algorithm. It knows nothing about the molecular system accept that it is represented by a derivable energy function and an array of coordinates. Both the energy values and the array of coordinates are provided to **LBFGS** by the **TotalEnergy** object.

The **TotalEnergy** itself also knows very little. When **LBFGS** asks it for an energy value, it triggers the **DistanceMatrix** to update. Then, **TotalEnergy** iterates over the **energyTerms** list and sums the values returned by each of them.

Simple energy term objects, like the **BondEnergy** presented in the figure, are a sum of elementary terms. Thus they only need to know the list of these elements. Only the **EnergyElement** object actually knows about both the atoms and the functional form. Some energy terms, however, are hard to decompose and thus their implementation is a bit less straightforward. Note that distances between atoms are not calculated by the energy term/element but provided by the **DistanceMatrix**. Thus each interatomic distance is calculated at most once.

1. The **LBFGS** class is part of the **optimizers** package.
2. The **Distance** and **DistanceMatrix** classes are part of the **geometry** package.

2. **A comment about the performance of the MESHI application Beautify in the CASP6 experiment.**

The usability of the MESHI package was demonstrated in the CASP6 experiment [http://predictioncenter.org/casp6/Casp6.html]. We have used a preliminary version of Beautify to refine homology and fold-recognition models that were downloaded from the CAFASP4 site. Selected Cα predictions of CASP6 targets were downloaded manually from the CAFASP4 site, and refined by Beautify. Some of the refined models were ranked among the best submitted.

Figure S6 shows the prediction made with Beautify on CASP6 target T0249_2 (#1 model of group #160) compared with the submissions of other groups. The graph generated by the CASP6 site according to the method of Hubbard (1999), shows that except for very small distance cutoffs, Beautify was indeed able to significantly improve over the original Cα model.

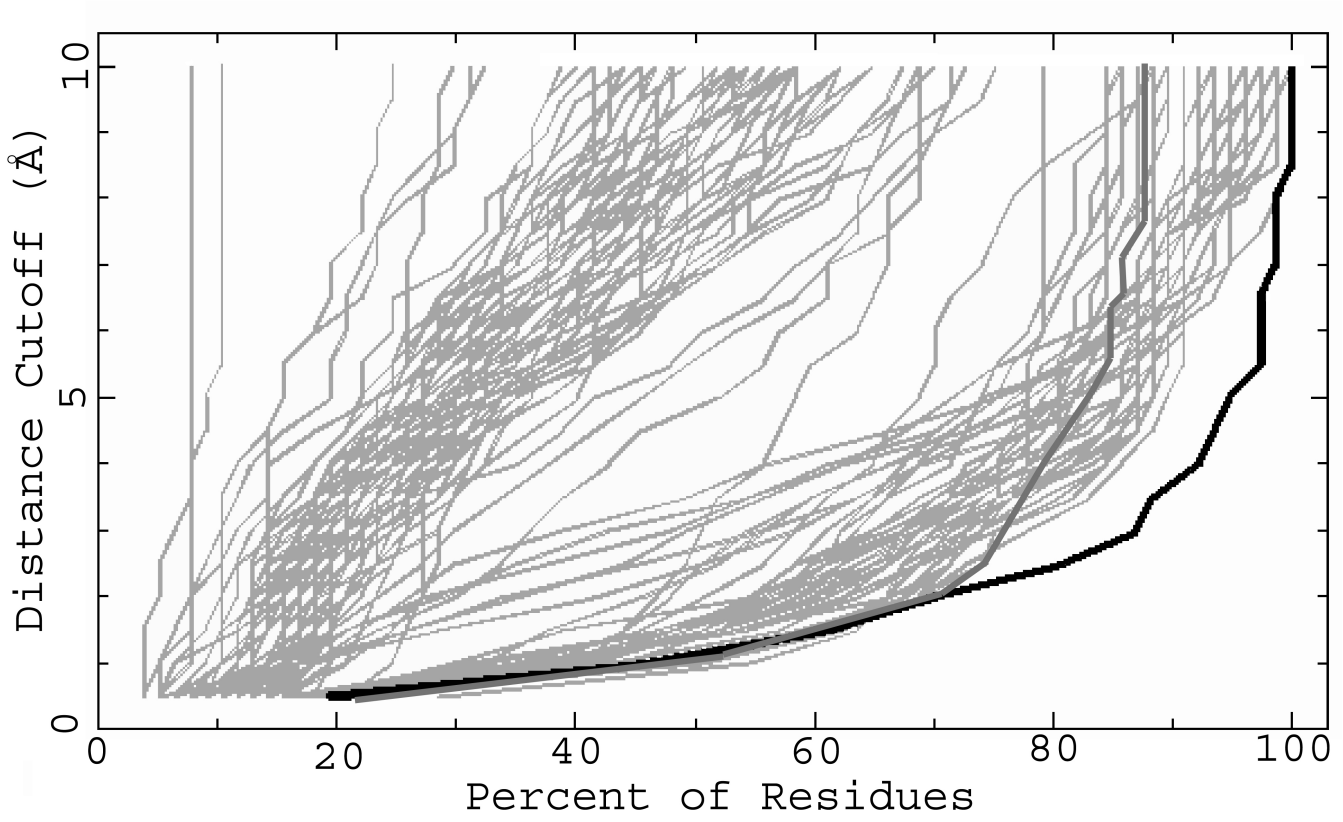

Figure S6. An RMS/Coverage graph of the prediction made with Beautify (black) compared with predictions of other groups (light gray) and the original FR model (dark gray) on CASP6 target T0249_2. Each line represents a single prediction, and shows the minimal RMSD from native of all Cα subset sizes (measured in percentage of the total number of residues in the target). The graph was downloaded from the CASP6 site, and was overlaid with the original FR model data curve.