Hacking KiNG

This document is intended to provide a field guide to the code for KiNG and its associated libraries, Driftwood and Chiropraxis. For more information, please see the relevant Javadocs, and ultimately, the code itself. In particular, most of the references for the various published algorithms are somewhere in the Javadocs. Some parts of the code are better commented than others, but it's mostly pretty comprehensible. If not, you can often use the incomprehensible parts without having to understand how they work.

The first part of this discussion will provide an overview of all the code, starting with `driftwood`, building `chiropraxis` on top of that, and finally building `king` on top of them both. The second part will cover the organizational principles behind the layout of code and resources, compiling and building, maintaining documentation, and distributing finished applications.

KiNG is always changing. This document was written in the early spring of 2004 (roughly version 1.22), and within a few months, several major changes have already occurred. So, take this document with a grain of salt; the final authority is the Javadocs and the code.

In brief, here are some of the major changes. The drawing system has been reworked to use a `Painter` that basically replaces a `java.awt.Graphics` object. This allows us to do other types of non-Java rendering, such as OpenGL (via the JOGL library). The plugin system has been totally overhauled to follow a service-provider model, which is partially reflected in this document; some of the changes may not have made it in, though. All the structural biology-specific tools that used to be in the `king.tool.*` packages have been moved into `chiropraxis.kingtools`. This effectively inverts the dependency relationship – KiNG no longer depends on Chiropraxis to compile, but compiling Chiropraxis now requires that KiNG be present.

Last updated 6 June 2004 by IWD for KiNG 1.28.

Overview of the code

Packages driftwood. and Jama.**

This group of classes defines a lot of the low-level, highly-reusable functions I needed to build KiNG and the various macromolecular modeling tools. Most of the classes have few dependencies, and fall into small clusters of related functionalties. Some of the sub-packages do depend on the others (*e.g.* `driftwood.moldb2` uses the data structures from `driftwood.data`), but I try to keep these to a minimum. In no case will any class in this group of packages depend on anything outside the group, except for the standard Java libraries. Thus, it should be easy to reuse these classes, in whole or in part, in other projects.

OK, I lied a little. I realized I was going to need some standard linear algebra operations, and I didn't want to write them. So I incorporated the JAMA Java Matrix Libraries into the driftwood tree, and some of the things in `driftwood.r3` depend on them. JAMA is public domain code released by the U.S. government (http://math.nist.gov/javanumerics/jama/) and has been stable since 9/11/2000 (version 1.0.1). Should a new version be released, the code can be dropped into the tree as-is; I've made no modifications to it.

Driftwood got its name because it's one of the few words that contains my initials in order (IWD), and because its an apt metaphor for these little bits of floating code.

driftwood.data

This package defines my own implementation of a hash table plus doubly linked list. Java provides this in the form of `java.util.LinkedHashMap`, but only as of version 1.4. Also, the standard implemention lacks any means of "walking" the linked list from an arbitrary starting point, which is basically what makes this data structure useful (in my opinion). The algorithms are adapted from "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein.

`UberMap` is the hash table implementation, and `UberSet` is a lightweight implementation backed by an `UberMap` instance. This is the same relationship `java.util.HashMap` and `java.util.HashSet` have.

driftwood.gui

This package contains a multitude of utility classes for constructing graphical user interfaces (GUIs), usually using the Swing toolkit that is part of Java. Many of them are fairly simple and won't be mentioned here, but several are worth elaborating on.

`ReflectiveAction` is used extensively throughout KiNG. In the standard Java event model, most GUI widgets generate an `ActionEvent` when something happens to them (*e.g.* a button is pressed). Each component is monitored by a class implementing `ActionListener`, which then responds appropriately. Unfortunately, this typically means creating one small class for every "event" that can happen in the UI, which for large programs is very expensive both in terms of disk space and run-time memory. By using the Java reflection API, `ReflectiveAction` allows one to redirect `ActionEvents` from a

specific component to a named *function* in some arbitrary class (usually `this`). This is much more efficient, although you do lose some compile-time safety because you could name a function that doesn't really exist.

`AngleDial` is a custom widget that allows the user to graphically select an angle by dragging a pointer around the circular dial. It's used in several of the tools in KiNG, and is fairly configurable.

`FoldingBox` and its ancestors (`IndentBox`, `AlignBox`, and `SwapBox`) allow for "fold-out" sections of the GUI that are (de)activated by a checkbox. This is how the "recessiveon" property is implemented in KiNG.

`TablePane` makes the process of hand-coding a GUI much like using TABLE tags in HTML. It's actually just a wrapper on top of the standard GridBagLayout, but I find it makes layout code much more compact and comprehensible.

driftwood.isosurface

This package contains all the code for loading and contouring electron density. It's named "isosurface" because in principle it is capable of generating isosurfaces for any function sampled in three dimensions. However, the focus so far has been on electron density.

The core functionality is in the `MarchingCubes` class, which implements the *marching cubes* algorithm. The basic idea is that you have a grid of data samples, and you see that grid as lots of little cubes that have one data sample at each corner. Classifying each corner (or *vertex*) as above or below the contour level (a.k.a. isosurface threshold) yields 256 different patterns, which can be reduced to 15 unique cases by rotation, etc. The algorithm marches through the cubes one at a time, and "contours" each one individually. The remarkable thing is that these pieces almost always fit together to make the correct surface. One occasionally gets holes, but the truly topologically correct algorithms are much harder to implement. My code is capable of generating either wireframe (mesh) or triangle (solid) surfaces.

Data samples are identified by three integer indices. Thus, the `MarchingCubes` class needs two kinds of input: a class that implements `VertexLocator` to translates the indices into a point in Cartesian coordinates, and a class that implements `VertexEvaluator` to look up the actual data value at the specified indices. In practice, a single data source usually implements both interfaces.

Output of the marching cubes algorithm can be directed to anything that implements the `EdgePlotter` interface. `MarchingCubes` feeds the `EdgePlotter` a series of kinemage-style move-draw commands that define either a wireframe mesh or a series of triangle strips, depending on how `MarchingCubes` was configured.

Three input formats are currently supported: O maps (DSN6 or Brix), XPLOR maps, and CCP4 maps. Each map type is read by its own class, all of which are descended from `CrystalVertexSource`. Only primitive output capabilities are provided in this package: `KinfileEdgePlotter` writes a kinemage file from the mesh output of `MarchingCubes`. (KiNG implements its own `EdgePlotter` that constructs the contours directly in memory, rather than first writing a kinemage file and then parsing it.)

driftwood.moldb2

This package implements data structures for working with macromolecules. `ModelGroup` represents the contents of a PDB file, and contains one or more `Models`. These in turn contain `Residues`, which contain `Atoms`. The information in this tree is *only* the naming and relatedness (parent / child) information.

To store coordinates, B-factors, occupancies, etc, each `Atom` has one or more `AtomStates`. The `AtomStates` for a particular `Model` are collected together into one or more `ModelStates`, each of which represents e.g. a particular alternate conformation. Each `ModelState` contains either zero or one `AtomStates` for each `Atom` in the `Model`, and thus represents a single, unique, unambiguous conformation (with the possibility that some `Atoms` are undefined, usually leading to an `AtomException`). `ModelStates` inherit from one another to avoid duplication of information, so that (for instance) the "B" conformation only needs to define positions that differ from "A". This also makes it easy to model a change to one residue as a small "mask" that rides on top of the original conformation. `ModelStates` are implemented using hash tables (often `driftwood.data.UberMap`, in fact) so that look-up of the state for a particular `Atom` is a fast (constant-time) operation.

`AminoAcid` provides some static functions for evaluating `Residues` that happen to be amino acids. You can measure ϕ, ψ, and τ; determine if the residue is pre-Pro; etc. Things like Ramachandran and rotamer evaluations are done in the `chiropraxis.rotarama` package.

I'm still not entirely satisfied with this library, although it's the best I've managed so far. AtomException really should have been a checked exception, because uncaught AtomExceptions are always crashing my code somewhere. Still, it would put a fairly substantial burden on clients of the library, because a lot of fundamental operations can throw this exception. I don't particularly like the way all the original conformations read in from the PDB file are bundled with the Model, but I don't have a good alternative yet. And I don't have any support yet for bonds (CONECT records and traversing a network of bonded atoms) or for ANISOU records.

driftwood.r3

The `r3` package provides basic support for 3-D geometric calculations; *i.e.* for calculations on points in $\mathbf{R}^3$. There are two interfaces (`Tuple3` and `MutableTuple3`) and one concrete implementation (`Triple`) of points in 3-D. `Triple` represents either points or vectors, depending on context, and provides functions for dot product, cross product, etc. `Transform` uses a 4x4 matrix to define rotations, translations, etc; `Transform`s can be pre- or post-multiplied together. Finally, three utility classes. `Builder` encodes some basic geometrical constructions, like Dave's `construct4` in Mage. `SuperPoser` implements an algorithm for least-squares superpositioning of two sets of points. `LsqPlane` uses the singular value decomposition from JAMA to find the least-squares plane through a cloud of points.

driftwood.util

This package contains miscellaneous utility classes for string manipulation and I/O. `Strings` provides several common parsing and formatting operations. `Props` is an enhanced version of `java.util.Preferences`, with automated parsing of numeric strings and fail-fast behavior by default. `OutputStreamTee`, `SoftOutputStream` and `SoftLog` together provide memory-sensitive caching of messages written to standard output and standard error. `StreamTank` collects output like an `OutputStream`, then lets you read it back out as an `InputStream`. `ProcessTank` leverages that to collect output from some external process.

*Packages chiropraxis.**

The classes in the `chiropraxis` group encapsulate a lot of higher-level molecular modeling operations that provide a lot of the core functionality of the modeling tools in KiNG. They are intermediate in complexity and in number of dependencies between those of `driftwood` and those of `king`. In particular, they depend heavily on `driftwood.r3` and `driftwood.moldb2`, so using `chiropraxis` in another project will mandate including the `driftwood` code as well. However, nothing in this package depends on anything in KiNG.

Chiropraxis is a general name for remodeling the backbone of macromolecular structures, and was dreamed up by Laura Weston Murray and David Richardson.

chiropraxis.mc

This package implements functions for (re)modeling protein mainchains. If we develop tools to refit RNA backbone in the future, they would likely go here. `CaRotation` implements the "Backrub" motion — a simple rigid-body rotation around an imaginary axis between two alpha carbons. It can handle either a single rotation for two Cα's that are any distance apart, or all the peptide rotations between those two Cα's. The math is straightforward, but tedious and somewhat error-prone, so it's nice to have it all packaged up here.

The package also includes two utility programs. `HingeFit` tries to iteratively superimpose two loop conformations using only Backrub-like moves, by selecting the hinge motion that most improves the Cα RMSD at each iteration. `Suppose` is a utility for doing least-squares superpositions. It was a test bed for me to play with Cα difference-distance plots, vector and unsigned-sum methods of "collapsing" the difference-distance information, and Lesk's method of choosing an optimal subset of atoms to superimpose on. It's pretty limited in usability, because it requires two PDBs with the same number of alpha carbons in them. It seems relatively straightforward to enhance it with the ability to select ranges, align on something other than Cα's, etc. However, I wrote it in a hurry so the code's kind of a mess, and the design would probably require substantial reworking. So far, I don't see the payoff in reinventing the wheel (ProFit and LSQMAN). I just edit my PDBs to be the same length, then run the alignment that way!

chiropraxis.minimize

This is sort of a toy package written so I could learn about steepest-descent and conjugate-gradient minimizers. `GradientMinimizer` supports both modes of operation, but the implementation is far from optimal, and may not even be correct! If you want to use this for a serious application, you should study something like the Numerical Recipes books or the GNU Scientific Library to learn the ins & outs of doing conjugate gradients robustly. I don't see how I could mess up steepest-descent that badly, but I may have.

The minimizer accepts a `PotentialFunction`. `SequenceSpacer` uses a `SimpleHarmonicPotential` in the minimizer to do a 3-D graph layout of a bunch of amino acid sequences. We used this tool to look at the "evolutionary" relationships among a lot of sequences spit out by Dezymer for scaffolds that had been reshaped with Backrub in different ways.

chiropraxis.rotarama

This package contains all the code for rotamer and Ramachandran evaluations. There's not much code, but the lookup tables take up many megabytes, so I wanted to make this stuff easy to separate out — that would be a lot of dead weight to carry in a project that didn't use these functions.

`NDFloatTable` is a generic implementation of an N-dimensional array of floating-point numbers. This is the sort of thing you need for doing a four-dimensional histogram of arginine rotamers, for instance. Each dimension can "wrap" so that 0 is the same as 360, etc. It also has functions for doing density traces in N dimensions, using either Gaussian or cosine smoothing functions. This means you can "write" a point at some arbitrary location and it will be spread around to the nearby bins; you can "read" the density at an arbitrary point and it will be estimated via linear interpolation from the nearest neighbors. The code for this is highly recursive and quite hairy. You've been warned.

`NDFloatTables` can be saved to disk in a binary format and loaded into memory again; this is how the Ramachandran and rotamer data is stored. The `silk` smoothing package contains a more sophisticated version of this code that can read & write a text-based format, too; it also supports double-precision floating point numbers (which is overkill for this application).

The `Ramachandran` and `Rotamer` classes are bare-bones evaluators. KiNG uses these to score the results of interactive modeling. However, the evaluations and kinemages produced on MolProbity come from completely separate code in my old `hless` package (so named because models didn't need Hs to be processed). That code uses a very different set of classes to represent macromolecular models, and since it works just fine I haven't bothered to port it all over to the "modern" `driftwood.moldb2` system.

Finally, `TauByPhiPsi` uses data from a 1996 paper by P. A. Karplus to predict the "ideal" (or at least average) tau angle for a given backbone conformation.

chiropraxis.sc

This package contains all the functions for working with protein sidechain conformations. `RotamerDef` is a simple data structure for storing rotamer definitions from the

Penultimate Rotamer Library (name, frequency of occurance, canonical χ angles). `SidechainAngles2` is the class that allows measuring and adjusting all the sidechain dihedrals, both χ angles and rotatable methyls, etc. It's somewhat fragile in that it relies on atom names being absolutely uniform, rather than trying to make decisions based on spatial data too. Both `RotamerDef` and `SidechainAngles2` use the `angle.props` resource file that defines all the rotamers and all the angles.

`SidechainIdealizer` contains code both for idealizing whole sidechains and for idealizing just the Cβ position (and Hα's). It uses `singlesc.pdb` and `singleres.pdb`, which are slightly adapted versions of the ideal geometry data hard-coded into Prekin. The algorithm for Cβ idealization is basically the one in Prekin, which constructs ideal positions from both directions and averages them. However, I treat all amino acids like alanine, whereas in fact the ideals are slightly different for the branched betas, etc. This is due to sheer laziness and could be fixed.

`RotamerSampler` is just a quick hack to generate the sampled sidechain conformation libraries that Homme wanted for Dezymer.

*Packages king.**

The code for KiNG is larger and more interconnected than that for `driftwood` or `chiropraxis`. Note that most of this code depends on the `driftwood` libraries, as well as on some third-party libraries (regular expressions, PostScript export). However, only the `king.tool.model` package, which contains the rebuilding functions, depends on `chiropraxis`. Thus, it's relatively easy to eliminate all the molecular modeling stuff if you just want a 3-D graphics program.

Also, a lot of the math in KiNG was done before I had written `driftwood.r3`, so there's some duplication of effort (dot products, cross products, matrix multiplication, etc.). I've slowly been trying to replace the original code with calls to standard stuff in `driftwood.r3`, but the transition is incomplete.

Incidentally, the name "KiNG" isn't me being conceited; it's an acronym for "Kinemage, Next Generation." The word "kinemage" is itself a contraction of "kinetic image."

king.core

This package comprises the minimal subset of classes to parse kinemage format files, store and manipulate them in memory, and write them back out. For just creating kinemages from scratch, it's usually easier to write the file yourself, directly. However, this package could conceivably be used by another kinemage editing or display program to handle the input and output.

A note: I sometimes use the term "group" to refer specifically to a kinemage group, and sometimes I use it generically to mean group, subgroup, or list. The intended meaning should be fairly clear from the context.

The data structure is the object-oriented, hierarchical tree that you would expect; it mirrors the structure of kinemage files. Thus a `Kinemage` object is at the top, it contains `KGroups`, they contain `KSubgroups`, they contain `KLists`, and they contain various subclasses of `KPoint`. Note that all the kinds of lists are identical internally; it's the kind

of points in them that determines what kind of list you have. This means that technically, you can have all different kinds of points in the same list if you wanted to (although there's no support for writing this out to a file). I'm not sure I would do it this way again if I were to start over—I might have multiple kinds of lists and only one kind of point instead—but that's water under the bridge now.

Two abstract classes, `AHE` and `AGE`, provide much of the shared functionality for the data structure classes. `AHE` (Abstract Hierarchy Element) is the parent of all the `KPoints` and of `AGE` and its descendants (see below). It provides the concepts of having a name (point ID or list/subgroup/group name), of belonging to a parent (*e.g.* the parent of a list is a subgroup) and belonging to a kinemage, and of being on (*i.e.*, its checkbox is checked) and being visible (*i.e.*, all its parents are also on). `AHE` also provides the basic hooks for being rotated and drawn to the screen, because it implements `TransformSignalSubscriber`. This mechanism will be discussed further in the section on KiNG's rendering engine.

`AGE` (Abstract Grouping Element) extends `AHE` to provide the shared functionality for everything that contains "child" nodes: `Kinemages`, `KGroups`, `KSubgroups`, and `KLists`. This is the level where masters, lens, dominant, and similar properties are implemented. All of these "grouping" or "container" classes are part of a doubly-linked tree rooted in the kinemage itself; *i.e.* every node knows both its parent and its children. Unfortunately, this isn't really coordinated very well, so to (for example) add a new subgroup to an existing group, `add()` has to be called on the group *and* `setOwner()` has to be called on the subgroup. I've done a better job coordinating maintenance of the same sort of tree structure in `driftwood.moldb2`, but by now this awkward system is fairly well enshrined in the KiNG code.

There are at least two functionalities that are implemented directly as part of `AGE` that really should not have been part of these classes: the checkboxes for turning groups on and off, and the `TreeNode` interface. The problem with the buttons was that the button had to be renamed whenever the group/subgroup/list was renamed. The better solution would have been to recreate all the buttons every time a renaming occurred, so that the button itself didn't have to be part of the `AGE` class. As a result, we are limited to one and only one checkbox controlling any particular `AGE`, which totally rules out multiple views of the same kinemage (for better or worse). Implementing `TreeNode` directly in `AGE` means that groups/subgroups/lists can be used directly in the `JTree` component of the hierarchy editor, and it simplifies implementing the cut/copy/paste operations. The downside is that it makes `AGE` unnecessarily complicated and may limit us to only one `JTree` at a time (I don't know). A better solution would have been to create `MutableTreeNode` wrappers around the `AGE`s.

Rounding out the container classes are `MasterGroup` and `AnimationGroup`, which are and were originally (respectively) `AGE`s too. `MasterGroup` implements a master button (and pointmasters), but doesn't actually keep track of its children anymore. Masters are implemented by having each `AGE` keep a list of master names that affect it; when a master is turned on/off, the whole kinemage tree is traversed and groups are flipped on/off as necessary. `AnimationGroup` is the remains of my grand scheme to enhance the kinemage format with an arbitrary number of highly configurable animations, of which the traditional `animate` and `2animate` would merely be a subset. Unfortunately, it was rather

awkward and relied on the original non-standard behavior of the masters. I still haven't decided whether I'm going to remove it entirely along with the Animations menu, or whether I want to take another crack at creating more complex animations.

Now we come to `KPoint` and its subclasses: `BallPoint` (also responsible for spheres), `DotPoint`, `LabelPoint`, `MarkerPoint`, `TrianglePoint` (also responsible for ribbons), and `VectorPoint`. They implement the standard point properties like color, aspects (via the `Aspect` class), unpickable-ness, and width/radius that they don't get from `AHE`, but their main function is to store an <x,y,z> coordinate. This is another place where I went tragically wrong. Points store both their original location *and* their transformed location (after rotation, scaling, etc.). Among other things, this means that implementing stereo is nearly impossible, because you need to track *two* transformed locations for each point. (I got around this by doing two transform-draw cycles; the downside is that you can then only pick points on whichever side was transformed last.) It also makes it very hard to use the nice linear algebra functions from `driftwood.r3`, because sometimes you want them to apply to the original coordinates and sometimes to the transformed ones. The upside and original motivation for this decision is that it's an efficiency gain, because storing the transformed coordinates outside of the points themselves requires frequently creating and then discarding thousands of short-lived objects, which can be expensive. My understanding is that modern Java VMs have been extensively optimized so that this isn't much of a concern any more. It would be very difficult to accomplish this change because it's so fundamental, but if it didn't degrade performance much it would really improve the design.

Closely related is the fact that each specialized point type contains all the code for actually drawing it to the screen. This was fine when there was only one way to draw each point, but as I introduced different quality settings for the rendering I realized this was a bad idea. A better arrangement would be to have pluggable drawing engines that know how to draw each type of point, so that you could even have (for example) an OpenGL option without having to change the point code, which should be primarily a very long-lived and slow-changing data structure. It's also worth mentioning that the `VectorPoint` and `TrianglePoint` classes maintain a singly linked list running through themselves, with each point knowing who it's predecessor is. (Other types of points know which list they belong to but not what order they're in.) The vectors and triangles use this information to draw themselves — if they have a predecessor, they draw from that point to themselves (and are thus "L" points); if their predecessor is `null`, they draw nothing (and are thus "P" points). The advantage to this scheme, relative to having each point know its index in an array, is that it requires little or no updating to insert and remove points from the list. The downside is that traversing the list for other purposes is more difficult; you have to retrieve the full list of children from the list, search for the starting point, and then explore up and down the list from there.

I'm almost ready to discuss the rendering engine, but first need to introduce a few utility classes. A `KPaint` object represents each named kinemage color, like "green" or "lilactint". It keeps the `java.awt.Color` objects used for painting points with various depth-cueing levels on white and black backgrounds, in full color and in monochrome. It also does all the calculations for how colors are depth-cued and how lighting interacts with the triangle and ribbon normals. `KPalette` actually defines all the different colors in

terms of a hue-saturation-value (HSV) model, and maintains the various "pens" used for depth-cueing line thickness.

`KingView` defines a view in the kinemage sense (center, rotation matrix, zoom or span) and is also used to track the current viewpoint as the kinemage is moved with the mouse. It's all straightforward matrix multiplication and so forth, but it was done way before `driftwood.r3.Transform` and is quite quirky. It's made even worse by the fact that it's thread safe, so it worries about synchronization. I originally anticipated that I wanted a background thread to be able to update the view, so I could get automatic rocking for presentations. However, I ended up using a `Timer` and doing the actual updates on the main thread anyway, so it's all for naught. The drawing process now just pulls the information out and builds a `driftwood.r3.Transform` to actually work with. I should also mention that the zoom can be stored as either a zoom value or a span value, because that's the way it is in the kinemage format. However, the first time a `KingView` is used for drawing, it converts everything to a span, based on the current size of the bounding box of the kinemage. I find spans to be more robust, because a view defined in terms of span doesn't change if you change the size of the kinemage. For instance, if you define a view of your model's active site and then merge in a larger tetramer of the same structure for comparison, you don't want the active site view to get zoomed out just because the kinemage now occupies a larger volume.

Points are actually rendered in two passes — in the first pass they are transformed (rotated, scaled, *etc*.) and register themselves to be painted, and in the second pass they are actually painted to the screen. `TransformSignal` drives that first pass with a publish/subscribe model. Various `TransformSignalSubscribers` register with the `TransformSignal` to be notified when a new rendering is initiated. When someone then publishes a rendering request to the `TransformSignal`, all subscribers are given the opportunity to transform themselves (via a matrix multiplication) and/or to register to be drawn in the next pass. As mentioned above, all `AHE`s implement `TransformSignalSubscriber`. However, typically only the kinemage is registered as a subscriber with `TransformSignal` because the `AGE`s (kinemage, groups, subgroups, lists) respond to a signal by simply passing it on to all their children. Thus, the signal trickles down from the kinemage down through its groups to all its points, which are then actually rotated and registered to be drawn. In addition to kinemages, many tools subscribe to a transform signal so that they can render objects that are not actually part of the current kinemage, such as markers or electron density contours. Finally, `TransformSignal` is itself a `TransformSignalSubscriber`, so that you could hook up several of them to create a sort of signaling cascade. I guess that means I've taken to many molecular biology courses, huh?

The `Engine` class coordinates most of the second pass of the rendering process. During the first pass, in response to a `TransformSignal`, all of the points are rotated and scaled and registered with the Engine to be drawn via the `addPaintable()` function. Balls and spheres are also recorded in a hashtable via the `addShortener()` function, so that lines whose endpoints match with the center of the ball can be shortened to make the balls look more realistically three-dimensional. As the rotated points are registered to be drawn, the ones that fall inside the visible slab (clipping planes) are sorted into one of 1000 bins based on their z-coordinate. This allows me to draw them in roughly back-to-front order.

(I call the bins a z-buffer, which I think is not quite the correct name for it.) All the points are drawn, regardless of whether they'll be covered up later; there are no provisions for hidden surface removal. In a figure consisting mostly of dots and vectors, I doubt it's worth the effort. The z-buffer is wiped clean before every new rendering operation, but between renderings the information is cached and used for picking. Thus the picking algorithm can traverse the z-buffer front to back, looking for points that approximately match the mouse click position. `Engine` also contains a plethora of public variables that control various aspects of the rendering process (depth-cueing mode, background color, *etc*.); they mostly match the options in KiNG's Display menu. These values can be adjusted to give different kinds of renderings.

The last major functionality in `king.core` is reading and writing kinemages. `KinfileParser` is responsible for reading the kinemages, though it delegates the low-level parsing (tokenizing) to `KinfileTokenizer`. The parser is rather large but is pretty simple. It's recursive-decent, except without the recursive part, because kinemages don't have any recursive (parenthesized) sorts of expressions in them. You feed the parser a stream, and it returns to you (1) a `Collection` of `Kinemage` objects and (2) all of the text and caption sections of the kinemage, concatenated together. Note that before using the kinemages you must call `initAll()` on them, which does things like calculate the bounding box and initialize the views, masters, and animations. `KinWriter` is also very straightforward; you feed it a `Collection` of `Kinemages`, some text, and a stream.

All that's left in this package are a few loose ends. `KinemageSignal` and `KinemageSignalSubscriber` are modeled after `TransformSignal` and `TransformSignalSubscriber`, but to inform listeners about changes to the kinemage (creating and deleting groups, *etc*.). Unfortunately, I built it rather late so it's not used by much of anything; most of KiNG relies on a more primitive `notifyChange()` method in `king.KingMain`. `RecursivePointIterator` is a utility class for visiting all of the `KPoints` in a kinemage; it gets used by the Find command.

king

The `king` package itself contains all the code that makes up KiNG the program — largely the graphical user interface (GUI) and its trappings. It also contains the foundations of the tool/plugin system (discussed below), but most of the actual tools and plugins are sequestered in `king.tool` and its subpackages; `king` does not depend on them and can easily be compiled without them.

`KingMain` provides the entry point for KiNG running as an application (it contains the `main()` method), and it serves as a control center throughout the lifetime of a run. `KingMain` parses the command line, instantiates the major subsystems and lets them find each other (via the `get___()` methods), coordinates the event notification system via `notifyChange()`, and manages multiple instances. Most message/event passing is done by clients calling `KingMain.notifyChange()`, which then trickles down into the `notifyChange()` methods of all the subsystems. When multiple instances are created in the same Java VM (using the File | New command), a new `KingMain` instance is created and a static counter is incremented. When a KiNG window and its associated `KingMain` die (File | Exit or close the window), the counter is decremented. When it reaches zero, `System.exit()` is called to ensure clean termination of the program.

Kinglet provides the entry point for KiNG running as a web page applet. It does very little, except to decide which of three modes its running in (embedded, free-floating window, or launcher button) and instantiate a KingMain object. All remaining tasks are delegated to KingMain, even the processing of applet <PARAM> tags.

The first task for KingMain is configuration, including parsing the command line or applet <PARAM> tags. These tasks are performed in the KingMain constructor, whereas the other subsystems are not created until later, in the Main() method (an instance method called after the constructor, not to be confused with the static function main(), where execution starts for the application). It first creates a KingPrefs object to hold all the configuration data. KingPrefs is a souped-up java.util.Properties class that holds lots of key-value pairs (*e.g.* fontMagnification = 1.0). The default values for all the keys are loaded from king_prefs in the JAR file, but may be overridden by entries the user's .king_prefs file (located in his/her home directory as defined by the underlying OS) or by command-line switches. The allowed configuration keys are documented in the king_prefs file and will not be repeated here. KingPrefs is also responsible for loading other resources from the JAR file (*e.g.* icons), for identifying the location of other programs distributed with KiNG (*e.g.* Probe), and for querying kinemage.biochem.duke.edu to find out if a newer version of the software is available. The query is just an HTTP GET and can be disabled by the user; it reveals no information about the user except their IP address and that they're running KiNG. The PrefsEditor class provides a GUI for modifying some of these configurable properties at runtime.

As mentioned above, the major subsystems are instantiated in KingMain.Main(), called after the constructor. They include KinStable, MainWindow and ContentPane, KinfileIO, KinCanvas, UIMenus, UIText, and KinTree. These subsystem heads all have references back to the KingMain object that created them, allowing them to retrieve references to each other as needed. At the same time, KingMain sets the font magnification (useful for those who can't see well and for giving presentations), captures standard output and error for logging purposes, and installs drag-and-drop handlers. The drag-and-drop handlers (FileDropHandler for the graphics window, MacDropTarget for the Finder and the Dock) are created using the Java Reflection API, so that the attempt will fail gracefully on machines that don't support it (VMs before 1.4 and non-Macs, respectively). This is a technique I use in several other places, most notably with the tool/plugin architecture. Only after all of this infrastructure is in place are files from the command line or <PARAM> tags actually loaded.

KinStable is a container for all the currently open kinemages (think horses' stable, not solid-as-a-rock stable). It provides mechanisms to add (open/append) and remove (close) kinemage, and owns the GUI kinemage chooser box. To actually open or save a kinemage, clients call methods in KinfileIO, which may either present a dialog box or not depending on the method chosen. It tracks the name of the last file opened and provides automatic versioned naming (foo.kin, foo.1.kin, foo.2.kin, …). It also displays the progress dialog. KinfileIO implements KinLoadListener so that it can use a KinfileLoader to do the actual parsing in a background thread (this is what makes the progress dialog possible). Clients that don't want to use KinfileIO directly but still want background-thread file loading can implement KinLoadListener themselves and use their own KinfileLoader.

Several other classes are also involved in I/O operations. `ImageExport` saves the current graphics display to disk in bitmap form as a PNG or a JPEG. `PdfExport` saves the display in Adobe's Portable Document Format for 2-D vector graphics. `XknWriter` tries to save the currently open kinemages as XML, and `Vrml97Writer` tries to save them as VRML2/VRML97. Both `XknWriter` and `Vrml97Writer` are incomplete, out of date, and not maintained; they are kept on with the thought that they may one day be fleshed out. All of these classes except `Vrml97Writer` use libraries that may not be present on all systems (`javax.imageio`, the iText PDF library, and `javax.xml`). Thus, they are instantiated by reflection so that systems with Java 1.3 or without iText can still run KiNG.

`ContentPane` is the root of the GUI hierarchy. It divides the main GUI space into graphics display, kinemage checkboxes, and user controls (zoom, clip, animate, *etc*.). When it receives a `notifyChange()` message from `KingMain` that the kinemage structure has changed, it refreshes the list of checkboxes with help from the current `Kinemage` object. `MainWindow` used to perform all of these functions, but they were factored out into `ContentPane` so that the GUI could be placed into either a free-floating `JFrame` (like `MainWindow`) or into a web-page-embedded `JApplet` (like `Kinglet`). `MainWindow` now is an empty shell that does very little except set its title.

Obviously, the graphics space dominates the GUI, and it is embodied in the `KinCanvas` class. It doesn't really do much in and of itself, but it coordinates a lot of things. The one thing that is wholly contained within `KinCanvas` is the zoom and clipping slider controls, and the math that makes them "exponential" rather than linear. Otherwise, `KinCanvas` delegates. It contains a `king.core.Engine` that it uses to do the primary drawing, both for painting the screen and for printing. It also contains a `ToolBox` instance, which is the top of the tool/plugin hierarchy. `KinCanvas` calls the `ToolBox` to do extra painting, like point ID and distance readouts. The whole drawing hierarchy is implemented using the `SignalTransform` system described above for `king.core`. `KinCanvas` also receives all the mouse and keyboard events for the graphics area, but passes them to the `ToolBox`, which in turn passes them to the current active tool.

`ToolBox` is a central part of KiNG's model for user interaction, because it manages all the plugins that appear in the Tools menu. At any given time, there is one (and only one) active *tool* (a specific kind of plugin), which receives all the mouse and keyboard events from the `ToolBox`, which in turn has received them from `KinCanvas`. In fact, this is the distinction between tools and plugins: tools receive user input from the graphics area (clicking, dragging, *etc*.) and (ordinary) plugins do not. Thus, there can be only one tool-type plugin active at a time, but any number of other plugins may be in use. `ToolBoxMW` is a subclass that adds support for mouse wheel events, which were added in Java 1.4. `KinCanvas` uses reflection to first try creating a `ToolBoxMW`, and if that fails, it creates an ordinary `ToolBox` instead. That way, Java 1.4 users get mouse wheel support but KiNG still runs with Java 1.3.

The `ToolBox` loads the plugins and tools via reflection so that KiNG won't crash if they can't be loaded. This is important both because some tools reside in other JAR files that could be missing, and because some tools require Java 1.4 and will not work with earlier versions of Java. The loading procedure uses a service-provider model like the `javax.imageio` package. Thus, in every JAR the file

`/META-INF/services/king.Plugin` contains the fully qualified class names of all plugins that should be loaded from that JAR, one per line. Every JAR file on the current classpath can contain such a file, so plugins can be compiled into their own JAR files and be detected by KiNG at runtime, rather than at compile time. In order for KiNG to detect plugin JARs, they must either be in the special `plugins/` folder or on the Java classpath. The `plugins/` folder is found in the same place as `king.jar` and will be searched for JAR files containing plugins when the `ToolBox` is loaded. The `plugins/` folder is the preferred method, and the easiest, but JARs may instead be placed directly on the classpath by using the `-cp` switch for java (or by referencing them in the manifest of the `king.jar` file, but this is strongly discouraged). `ToolBox` also contains the code to build the Tools menu and the tool-related items in the Help menu. The name of the submenu that each plugin should appear in is determined by an item in the `king_prefs` file (`/king/king_prefs` in the JAR or `.king_prefs` in the user's home directory) named *`fully.qualified.PluginClassName`*`.menuName`. The special values `<main menu>` and `<not shown>` either place the plugin directly into the Tools menu or suppress its menu item altogether, respectively. The user can customize these settings, so the ones provided in the JAR just serve as a default.

`ToolBox` also contains a `ToolServices` instance, which has the code for picking, measuring, markers (crosshair and Mage-measures), and the common mouse actions — rotate, translate, clip, and zoom. Many of these could have actually been included in the base tool class, but instead the tools just call *e.g.* `rotate()` or `zoom()` in `ToolServices` when they get a mouse event. However, measures and markers do have to be separate from the tool implementations, because as a user you don't want the markers/measures state to be tied to which tool is active. If that were true, your markers would jump around just because you switched from Edit Properties to Move Point, or a Mage-like construct4 tool wouldn't be able to "see" the measurements you just made with the Navigate tool. Thus, markers and measures reside in `ToolServices`.

Since they don't have to deal with input from the graphics area, plugins are easier to write than tools, so I'll cover the `Plugin` class first. `Plugin` is an abstract base class that all plugins and tools extend, either directly or indirectly. Most plugins should extend `Plugin` directly. It defines several useful variables for accessing other parts of the drawing/event system: references to the `ToolBox`, its `ToolServices` instance, the `KinCanvas` that owns them, and of course to `KingMain`. There are only two methods a `Plugin` must implement. The first is `getToolsMenuItem()`, which returns a single `JMenuItem` to appear in KiNG's Tools menu. If you need multiple menu items for your plugin, put them in a `JMenu` and return that instead. The second is `getHelpMenuItem()`, which returns a `JMenuItem` for KiNG's Help menu that provides help on using this plugin. Either of these functions may return null to indicate that they don't have an associated menu item. `getToolsMenuItem()` must be implemented from scratch for each plugin, but there is already an infrastructure in place for the default Help behavior. If the plugin's documentation resides in the KiNG manual, you need only return an HTML anchor from `getHelpAnchor()`. If the help is in a different HTML file, you can instead override `getHelpURL()` to return that address. Note that the Java browser only supports HTML 3.2. If you're going to use this default help system instead of overriding `getHelpMenuItem()`, you should override `toString()` to return a reasonable name for

your plugin. Since it's so easy to add plugin documentation, please provide help pages for your plugins!

More complex plugins may want to implement some additional methods. Overriding `isAppletSafe()` to return `false` will ensure that KiNG does not try to load that plugin when running as an applet. This is useful for plugins that access the file system, use native libraries, and so on. The `getDependencies()` method returns a list of the fully-qualified class names of other tools or plugins on which this plugin depends. KiNG will only load this plugin if it can successfully load all of its dependencies first. Complex chains can be resolved, but be careful not to introduce circular dependencies, as this will prevent all the plugins involved from being loaded.

Tools must be derived from the more complex `BasicTool` class, but the good news is that `BasicTool` has no abstract methods that must be overridden — it's completely functional out of the box. In fact, it's actually the implementation of the default Navigate tool that provides Mage-like navigation through the kinemage. Tools have the same menu item requirements as plugins, and the help system is identical. Thus it's usually sufficient to override `toString()` and either `getHelpAnchor()` or `getHelpURL()`; with that information, `BasicTool` will create appropriate items in the Tools and Help menu for you.

As mentioned above, only one Tool can be active at a time. Thus, tools receive `start()` and `stop()` messages when they are (de)activated. A tool will receive a `reset()` when the active kinemage changes. Tools can implement `MouseListener` and `MouseMotionListener` methods directly, but it's usually more convenient to allow `BasicTool` to parse those into clicks, drags, and mouse wheel events. Each type of mouse event belongs to one of four classes, which are determined by which mouse button is used (for multi-button mice) and/or by which keyboard modifiers were held down. The classes are normal (left mouse button), Shift (right mouse button), Ctrl (middle mouse button), and Shift+Ctrl (left and right mouse buttons together, for drag but not click). Three events and four states lead to twelve listener functions; see the `BasicTool` code for their default implementations. The `click()` functions receive the `KPoint` picked by the mouse (or `null` if none), the coordinates of the click, and the original `MouseEvent`. The `drag()` functions don't get coordinates directly, but instead get the number of pixels the mouse has moved in x and y since the previous call to `drag()`. The variable `mouseDragMode` is set to `MODE_HORIZONTAL` or `MODE_VERTICAL` if the drag started as primarily horizontal or vertical, which simplifies implementing behaviors like Navigate's zoom vs. clip. The variables `isNearTop` and `isNearBottom` allow for switching on behaviors like the pinwheel rotation that occurs when you drag near the top of the graphics area.

I anticipated that many tools would want to display a non-modal dialog box when they were active, like the Draw New tool does. Thus, if a tool returns a non-null `JComponent` from `getToolPanel()`, it will automatically be placed in a dialog that pops up when `start()` is called (via the `show()` method) and disappears when `stop()` is called (via the `hide()` method). The `initDialog()` method was added later as a place to do things like add a menu bar to the dialog; because the `dialog` variable is still `null` during the constructor. The whole thing is far more trouble and confusion than it's worth, but I'm too lazy to take it out now and force other tools to implement this behavior for

themselves. If you don't want to deal with it, just let `getToolPanel()` continue to return `null` and create your own dialog box from scratch.

There are two more important methods in `BasicTool`: `signalTransform()` and `overpaintCanvas()`. Yes, just like the active kinemage, the `ToolBox`, `ToolServices`, and the active tool are part of the `TransformSignal`-driven drawing hierarchy. (Plugins must register themselves explicitly to be included, but they can if they want to.) By having the tool's `signalTransform()` call the `signalTransform()` method of a group, subgroup, or list that it owns, that graphics element can be rendered to the screen or printer without being saved as part of the kinemage. This is how `ToolServices` manages markers, and is analogous to how electron density is displayed. The `overpaintCanvas()` method allows the tool to draw directly on the canvas after all the 3-D graphics are rendered; this is how `ToolServices` paints point ID labels, measurement distances, *etc*.

The best way to learn about writing tools and plugins is to explore the existing code in the `king.tool` package and its subpackages. See below for more description of the existing tool set.

There is actually one plugin that lives in the `king` package, which is the electron density displayer `EDMapPlugin`. It's a simple shell that allows the user to select a file and then opens a `EDMapWindow` for it. As mentioned above, in its constructor the window subscribes to the `ToolBox`'s `TransformSignal`, so that it can display the maps without them necessarily being part of the kinemage. `EDMapPlotter` implements `driftwood.isosurface.EdgePlotter` and is used by `MarchingCubes` to create the mesh or translucent triangle surface directly in memory from `Vector-` or `TrianglePoints`. `EDMapPlugin` was included in the main `king` package because it's not optional; the `Kinglet` applet may try to create `EDMapWindows` directly in response to certain `<PARAM>` tags.

All that's left is a run-down of the rest of the GUI code. Keep in mind that all of the GUI makes heavy use of `driftwood.gui.ReflectiveAction` for event handling. Without it, I would have to write hundreds of small anonymous inner classes. That way gives you more compile-time type safety and is the Java standard, but it becomes very costly to store that many classes in the JAR file and in memory (even though they're small). My way is much more efficient and (I think) much easier to read, at the expense of some compile-time safety checks.

`KinTree` implements the hierarchy editor using a `JTree` (remember that all the classes descended from `AGE` implement `TreeNode`). It's large because of all the tree manipulation functions; Cut, Copy, and Paste are particularly lengthy. None of it's terribly complicated though. Some of those functions use the `GroupEditor` and `PointEditor` classes, which are also used by the Edit Properties tool to modify groups, subgroups, lists, and points. The editors use instances of `ColorPicker` to graphically set the color of lists and points.

All of the menus are defined and either handled directly or delegated elsewhere in `UIMenus`, which is another monster file. It also holds the timer for auto-animation, and owns instances of `PointFinder` (for the Edit | Find dialog) and `ViewEditor` (for editing views). `PointFinder` uses `gnu.regexp` regular expressions to drive the search, because Java didn't come with regular expression packages of its own until version 1.4. Note that `UIMenus` has mechanisms for rebuilding the Views and Animations menus, which need

updating when kinemages are (un)loaded. If I weren't so lazy, I would split this into one file per major menu to increase maintainability, which is actually what I did already with the Display menu (`UIDisplayMenu`). The Display menu items mostly act by changing variables in the `KinCanvas`'s `Engine` instance.

Now for a few loose ends. The `UIText` class manages the kinemage text window and allows the user to edit that text. `HTMLHelp` is an ultra-simple HTML browser intended for displaying help information, like the user manual. Note that the Java HTML viewer component only supports the HTML 3.2 standard, so you can't do anything too fancy. `GridBagPanel` is the predecessor to `driftwood.gui.TablePane`, but harder to use. I would remove it entirely except that some parts of the GUI still haven't been upgraded to `TablePane`. `ReflectiveRunnable` works just like `ReflectiveAction`, except that it lets any arbitrary function stand in for the `Runnable` interface. This is sometimes useful when you're trying to do multi-threaded things (such as file loading).

king.tool

The `king.tool` package and its subpackages contain a fair collection of different tools and plugins for doing various things, some related to our research, some not. This is the part of KiNG undergoing the most active development, so this section is almost certainly out of date already. The operation of all these tools and plugins is well described in the user manual, so this overview will seek to highlight features of their implementation that may be of interest to other developers. After all, the existing tools are the best reference on how to write new ones. (See above for a general description of how to create new tools and plugins.) The `king.tool.model` package for tools that work with PDB structure files is more complex than most of the others, so it will be treated separately in the next section.

`MovePointTool` and `EditPropertiesTool` are two of the basic kinemage-editing tools. `MovePoint` tool uses `KingView.translateRotated()` to convert a movement in the scaled, rotated graphics space back to a change of coordinates in the kinemage model space. It also overrides `mousePressed()` to find the picked point at the start of a drag, which is something most drag operations don't need. `EditProperties` is a minimally different from `BasicTool`; it just uses `king.GroupEditor` and `king.PointEditor` to alter the kinemage.

`Dock3On3Tool` is a more sophisticated tool. It overrides `BasicTool.getToolPanel()` to provide an accessory dialog box, and it has its own data structure for storing picked points. It also overrides `signalTransform()` to display the numbered markers that describe the alignment to be performed. Its `transformAllVisible()` method demonstrates how to go about changing the coordinates of kinemage objects using a `driftwood.r3.Transform`. Finally, it uses `driftwood.r3.Builder` to do the geometric calculations necessary for the superposition. `DockLsqTool` is quite similar, except that is uses `driftwood.r3.SuperPoser` to do the least-squares superposition calculation.

`ViewpointPlugin` is fairly boring, except that it illustrates how to modify the current view directly rather than through the `ToolServices.rotate()`, `zoom()`, *etc.* functions. `CrossWindowPickTool` shows how two or more KiNG instances in the same Java VM can communicate. (Multiple instances are created with the File | New command.) Each

new KiNG instance has its own instance of `CrossWindowPickTool`, and these all share a static class list of all the `CrossWindowPickTool` instances. The list is implemented using `SoftReferences` to avoid memory leaks. Thus, each instance of the tool is able to find out about and communicate with all the others.

`SolidObjPlugin` is interesting in that it does some non-trivial geometric calculations and creates fairly intricate surfaces from `TrianglePoints`. It also does the cute trick of traversing the kinemage and finding all of the currently visible vector lists.

`DrawingTool` has a whole host of interesting tricks. Its undo mechanism saves the state of a `KList` by making a copy of its children variable, with the additional complication of saving connectedness information at the end of `Vector-` and `TriangleLists`. The undo system also uses `SoftReferences` to make the undo cache memory sensitive — it's cleared out in response to memory demand by the Java VM. `DrawingTool` uses `driftwood.gui.FoldingBox` with a `JRadioButton` for a slight twist on the GUI segment that "drops down" in response to a checkbox. It demonstrates the use of a `Kinemage`'s `metadata` field for storing kinemage-specific information with the kinemage rather than within the tool. This helps avoid memory leaks when kinemages are closed. (The other solution is to use a `SoftReference` again.) The prune and punch commands demonstrate traversing a `KList` and making modifications to a line segment or polyline. The auger command shows how to use `Engine` to pick all the points in a region, and the spherical crop demonstrates the use of `king.RecursivePointIterator`. Auger also does some interesting work in `mouseMoved()` to do XOR drawing of its beach ball of death rather than trying to install a large custom mouse cursor.

The RNA tools in `king.tool.xtal` were developed by Vincent Chen during his rotation in the Richardson lab in the fall of 2003. They were derived from `EDMapPlugin` and friends, but modified so that the user can pick one isolated polygon in the density (like a phosphate peak in RNA) and the tool will group all those vectors together and highlight them. I'm sure the mechanics of rearranging the vectorlist connectivity are very interesting here. It also searches the electron density map for the highest-density node with that polygon's bounding box, and marks it as a suggested atom location.

king.tool.model

This package contains a set of highly interdependent tools and plugins for modifying protein structure, as represented in a Protein Data Bank (PDB) coordinate file. The system is centered around the `ModelManager2` plugin, which other tools can access via the `ToolBox`'s `getPluginList()` method. `ModelManager2` requires both a kinemage and a PDB file, unlike Mage: the kinemage must already be loaded in KiNG as usual, and the user is prompted to load a PDB into the model manager as soon as any tool makes a request that requires structural information. It makes correlations between the graphics display and the invisible structural data typically by comparing Cartesian coordinates, but sometimes based on kinemage point ID and atom or residue names. The manager uses the `driftwood.moldb2` system to load, store, manipulate, and save structural data.

`ModelManager2` thinks about structure in terms of a static, "frozen" conformation that represents the molecule as it currently stands; and zero or more dynamic, "molten" sections that are currently being manipulated by the modeling tools. The molten sections,

with their gaps filled in with coordinates from the frozen conformation, constitute a complete "molten" conformation for the whole structure (even though many parts of it are not actively being remodeled). Each of these states can be retrieved as either a `ModelState` object or a (temporary) PDB file.

Tools (or plugins) that wish to interact with the model manager must implement the `Remodeler` interface and then call `ModelManager2.registerTool()` to register themselves. For convenience, new tools may extend the abstract `ModelingTool` class, which automates the process of finding a reference to the model mananger. Whenever a tool wants to change the model conformation, it calls `ModelManager2.requestStateRefresh()`. Each registered `Remodeler` is then queried in turn — fed a starting `ModelState` and prompted to either return it unchanged, or to create a new `ModelState` descended from the input, modify it, and return that. `Remodelers` must not modify the input state directly, but because of the hierarchical/layered design of `ModelStates`, the cost of making a change is only linear in the size of the change, *not* linear in the size of the entire model. In this way, each `Remodeler` composites its own contributions onto the transformation from frozen state to molten state. Thus, one tool might move the backbone, another might rotate a sidechain riding on that backbone, and a third might idealize the covalent geometry of that sidechain.

To make a permanent change to the frozen model, a `Remodeler` calls `requestStateChange()`. That one tool is then asked to calculate a modified conformation for the frozen state via the same process used to compute the molten state above. The tool is then unregistered (so that it won't apply the same modification again to the molten state) and the molten state recomputed as above. The tool must re-register itself to begin a new round of remodeling; of course, tools can abort a modeling operation by unregistering themselves directly.

So far, all the remodeling has changed only the coordinates of atoms. If a tool needs to add, remove, or replace atoms or residues (*e.g.* mutate a sidechain), it needs to change both the `Model` and the `ModelState`. To do this, one calls `replaceModelAndState()`. This requires that nothing in the model be currently molten, because if the `Model` is changed while other tools are reshaping it, the atoms they thought they were moving could disappear out from under them! All of the changes to the `Model` and the frozen `ModelState` are tracked by `ModelManager2`'s undo system and are fully reversible (provided that the client cloned the original `Model` and modified the clone instead of modifying the original). The changes to the molten `ModelState` are of course transient by nature, so there's really nothing to undo for them.

In addition to the frozen and molten models and the undo system, `ModelManager2` also drives all the interactive model visualizations. A better design of would be to factor the visualizations out of this class and make them operate as registered clients of the model manager instead, more like the tools do. However, I haven't gotten to this yet. So at the moment, the manager uses `ModelPlotter` to make kinemage drawings of the entire frozen state (in sky and yellowtint) and the molten parts of the molten state (in shades of orange). `ModelPlotter` operates only on proteins and goes strictly by atom names, so its pretty easy to break it. Still, it works most of the time. Its connectivity database is stored in the `sc-connect.props` resource file. `ModelManager2` also calls the command-line C

programs `probe` and `noe-display` to generate more visualizations, although most of the `noe-display` logic and interface has been isolated in the `NoePanel` and `ExpectedNoePanel` classes.

The calls to `probe` and `noe-display` are done by `BgKinRunner`. These programs can take several seconds to run, and then KiNG still has to parse the textual kinemage data they output. Furthermore, running external programs from Java seems to be a little dicey — sometimes they hang indefinitely for no apparent reason in a random and irreproducible fashion. For all these reasons, `BgKinRunner` launches such programs in a background thread and kills them off if they run for too long. It then parses the kinemage data and adds it to the display, replacing the program's previous output (if any). Thus, the `probe` and `noe-display` visualizations often lag a few seconds behind the moving molten model, but the UI remains responsive and crashes don't seem to have any adverse, destabilizing effect on KiNG.

Finally, to the tools, of which there are three. `HingeTool` implements the generalized version of the Backrub backbone motion, and uses the `PeptideTwister2` component to manage peptide rotations. (Both use `chiropraxis.mc.CaRotation` to do the actual rotation calculations.) `ScMutTool` is entirely self-contained, and is the only one that actually changes the `Model` as well as the `ModelState`. `ScRotTool` spawns lots of `SidechainRotator` windows that do the actual work of reconfiguring sidechains.

Organizational principles

Source code

All of the Java source code is stored in the `src/` directory of each top-level package's distribution (`driftwood`, `chiropraxis`, `king`). Subdirectories are named by (sub)package name according to the Java convention.

When editing code, I prefer to use the jEdit text editor (http://www.jedit.org). One of its features is explicit "folding" of the source code, driven by the triple curly brace markers (`{{{` and `}}}`). I try to keep these folded sections to about 40 lines, so that one of them fits comfortably on the screen when expanded. Thus, whenever possible, I work hard to keep my function definitions to about 40 lines as well. Multiple functions may go in the same fold if they're short. There's a fair bit of lore in the CS world that comprehensibility of any chunk suffers badly as soon as you can't see the whole thing at once, and my own experience bears this out.

Since I leave a line of space between folded segments for readability, that puts an upper limit of about 20 segments * 40 lines = 800 lines of code in any one file. Beyond this point, you can't even fit the folded overview of a single class on screen at once. Indeed, I find it hard to keep track of all the things going on in a class with 1000 lines of code and sometimes end of stepping on my own toes. With few exceptions, I believe that classes this big are trying to do too much, and should be split into several more focused and modular components.

The same philosophy extends to package organization: 10 to 20 classes is a nice number. Finer divisions seem silly unless you expect future growth (as I do for many of the packages documented here), but beyond about 20 classes I find that it's hard to get a handle on what a package does and how its parts interrelate. Libraries with hundreds of (public) classes in a package are just plain poorly organized, in my opinion. The main `king` package is pushing 40 classes now, and I think that's a little unwieldy. On the other hand, `king.core` is about 30 classes, and I can't see any good way to segment it further, so I won't.

I make a real effort to document my source code using Javadoc comments (`/** … */`) for all the functions. I don't worry so much about variables, though I should pay more attention to them. One thing that's currently lacking is package-overview documentation in the Javadocs, but that's basically the purpose of this document, so I may not do much more there.

Resource data and user documentation

The `resource/` directory for each top-level package has a structure parallel to that of `src/`, again named by (sub)package name. The `resource/` directory is incorporated directly into the JAR file, so that the compiled Java class files and their necessary resources end up in the same directory of the JAR. (The code can retrieve this resource data with `Class.getResource()` and similar functions.) For example, the smoothed rotamer and Ramachandran distributions are stored this way in `chiropraxis`. KiNG

keeps its icons, configuration files, and HTML help in the `resource/` tree. Most of the command-line utilities also keep their help information in plain text `.help` files this way.

It is my goal to provide all end-user documentation in HTML format always, and in PDF format when it makes sense. As mentioned, command-line utilities often contain their own plain-text help summaries as well. There are no really great tools that I've ever run across for producing clean and attractive HTML and PDFs from the same source document. Microsoft Word is a possibility, but it writes atrocious HTML that the Java viewer could never handle and it doesn't run on Linux. The best alternative I found is LyX (http://www.lyx.org), a "what-you-see-is-what-you-*mean*" cross-platform document processor that uses LaTeX as its underlying formatting engine. It produces pretty clean HTML and nice PDFs, and it's really good about helping maintain a hierarchical structure (chapters, sections, subsections) and cross references.

Every project also has a `CHANGELOG.html` file, which is really just an HTML wrapper around a plain text file. It records an overview of the changes made for each version, along with notes and ideas for future features, to-dos, and bug reports that are still outstanding. I take care to make sure the nominal current version number from the changelog tracks with the actual version number, stored in the `resource/` tree as `version.props`.

Build system

The build system is driven by Apache Ant (http://ant.apache.org/), which is the Java standard. Ant stores its build instructions in `build.xml` files, one per top-level project. The major targets are `compile` (run `javac` to compile the source), `build` (construct the JAR file), `dist` (build source and executable distribution bundles), and `clean` (remove products of the other three). The compiled `.class` files are placed into the `build/` directory, and the distribution bundles are built up in the `dist/` directory. Both directories are removed by `clean` and recreated as needed. There are also other targets, like `backup` and `deploy-`*`xxx`*, that exist only for my convenience. You can ignore these or rewrite them for your own use.

The build process does make use of version information in `resource/`*`packagename`*`/version.props`. In fact, it updates the build number every time a build occurs. This makes it easier to track the exact version and build date for a distributed JAR file.

I take dependency management seriously, because I'm serious about wanting other to reuse my code. I know from the experience of trying to borrow other people's libraries that unless I can cut out just the small subset of functionality I'm interested in, I'm not going to use the library — who wants to drag around hundreds of kilobytes of dead weight? Thus, as I outlined below, `driftwood` has no dependencies, `chiropraxis` depends only on `driftwood` (and for some functions, other third-party libraries), and `king` is allowed to depend on both `driftwood` and `chiropraxis`. I also try to minimize or eliminate dependencies among the subpackages whenever possible. A great free tool for querying and visualizing these dependencies is Dependency Finder (http://depfind.sourceforge.net/). I use it frequently for dissecting and refactoring my own code, as well as for trying to understand other people's code.

Distribution and installers

For those familiar with running Java programs, the JAR files or distribution bundles created by Ant are all that's needed. However, for KiNG users I wanted a more novice-friendly distribution method. I found the registered-but-free basic version of Install Anywhere (http://www.zerog.com/) does a very nice job of distributing Java applications bundled with a private copy of the Java VM and runtime libraries. This makes for some large (ten megabytes and up) downloads, but great convenience. I use this method for Windows and Linux, where a Java VM is not part of the standard installation. For the Macintosh, I used the included OS X developer tools to make an application bundle for my JARs. Once you've created that once, you can easily update it just by replacing the JARs inside, without having to go through the whole song and dance again.