BMC Bioinformatics

BioMed Central

Software

Open Access

A knowledge discovery object model API for Java

Scott D Zuyderduyn and Steven JM Jones*

Address: Canada's Michael Smith Genome Sciences Centre, BC Cancer Agency, 600 West 10th Ave., Vancouver, Canada

Email: Scott D Zuyderduyn - scottz@bcgsc.ca; Steven JM Jones* - sjones@bcgsc.ca

* Corresponding author

Published: 28 October 2003

BMC Bioinformatics 2003, **4**:51

Received: 11 July 2003
Accepted: 28 October 2003

This article is available from: http://www.biomedcentral.com/1471-2105/4/51

© 2003 Zuyderduyn and Jones; licensee BioMed Central Ltd. This is an Open Access article: verbatim copying and redistribution of this article are permitted in all media for any purpose, provided this notice is preserved along with the article's original URL.

Abstract

Background: Biological data resources have become heterogeneous and derive from multiple sources. This introduces challenges in the management and utilization of this data in software development. Although efforts are underway to create a standard format for the transmission and storage of biological data, this objective has yet to be fully realized.

Results: This work describes an application programming interface (API) that provides a framework for developing an effective biological knowledge ontology for Java-based software projects. The API provides a robust framework for the data acquisition and management needs of an ontology implementation. In addition, the API contains classes to assist in creating GUIs to represent this data visually.

Conclusions: The Knowledge Discovery Object Model (KDOM) API is particularly useful for medium to large applications, or for a number of smaller software projects with common characteristics or objectives. KDOM can be coupled effectively with other biologically relevant APIs and classes. Source code, libraries, documentation and examples are available at http://www.bcgsc.ca/bioinfo/software.

Background

The development of bioinformatics software for effective analysis and interrogation of biological data, and indeed software in general, must include the creation of a data handling framework. Ideally, this framework must be accurate, robust, extensible, and technically feasible. The successful implementation of this framework has substantial implications for the ultimate success of software development. Further, the ability for such projects to be quickly utilized in other arenas of biology, improved by multiple developers, or to be evolved to handle changing requirements is directly affected by the initial choice of a core data model [1].

Most modern programming languages are well complemented with standard libraries and components that remove a great deal of necessary low-level computational tasks. For example, arrays, lists, and vectors all provide for easily implemented methods of managing and manipulating sets of data. Since there are characteristic operations and common manipulations of data lists, the use of standard constructs is an advantage to the developer. Developers who use these standards wherever possible will increase the speed of development and robustness of the result. Improvements to implementation are transparent and inheritable, mundane algorithms do not need to be developed or repeatedly utilized, and successful use can be repeated in future projects [2].

The Knowledge Discovery Object Model (KDOM) is an open source API written with Java 1.4 [3] that attempts to embrace this ideal for biological data. Characterizing and

standardizing commonalities in biological knowledge utilization can divert more development focus to novel creations. Although scientific literature holds many examples of how to approach the creation of an ontological system [4,5], KDOM provides a core API to decrease the time and effort needed to deploy such a system, including the means to allow a user to visualize and manipulate the data.

Results and Discussion

Biological knowledge

What are the commonalities in biological data? Take the example of a "gene". An *in silico* gene can have a sequence, an annotation, possibly a chromosomal location, functional motifs, or similarities to other genes. Not all of these properties can be assumed to be enduring. It is certain that new properties will be discovered. However, we are confident that the larger definition of a gene will remain accurate for the foreseeable future. Further, we know that data relationships have inference in and of themselves. In a microarray experiment, an oligonucleotide is spotted onto a slide, and washed with labelled cell RNA that will hybridize depending on the level of expression of genes containing that oligo's sequence. The oligonucleotide has a sequence, a position on the slide, and an observed colour when the experiment is performed. It is not the oligonucleotide itself, nor the colour, or even the corresponding gene that provide the inference of the experiment. It is the combination of the three that offer knowledge. This reality as it relates to building an effective ontology has been previously described [4].

Describing knowledge

KDOM incorporates the above philosophy in its architecture. Since the API is Java-based, object-oriented technique is a focus. The API contains almost 40 classes, but the modelling of information primarily involves three: all biological data is a subclass of *BiologicalData*; the storage, acquisition, and management of acquired knowledge is handled through the *BiologicalBrain*; and interactions between data are described with a *BiologicalLink* (Figures 1,2). This is intended to model a labelled graph (which can be specified as directed or undirected by the implementer), where each BiologicalData object is a vertex, and each BiologicalLink object is an edge.

At this basic level, KDOM offers several advantages:

First, data is managed such that once an instance of a unique object is created; it is guaranteed to be the only instance of that object within the system. Unrelated procedures within the application domain can freely create or call instances of objects, knowing that existing instances will be utilized. For example, the developer could define three types of *BiologicalData* called *Chromosome*, *Gene*, and *FunctionalDomain*. If the user of the application invokes a procedure where all *Gene*s from a single *Chromosome* are acquired for use, a subsequent procedure acquiring all *Genes* with a given *FunctionalDomain* will use the existing *Gene* objects if appropriate.

Second, the developer need only define the object itself and its relationship to other objects once. If another task (possibly undertaken by a different developer) requires the same object type (or the same relationship between objects) the existing KDOM infrastructure can be utilized.

Third, properties of the data are acquired only when needed, and need only be acquired once. If a property of the *Gene* is "annotation", the *BiologicalBrain* will retrieve it when first needed, and the system will automatically utilize it again on subsequent procedures.

This approach saves computational energy, physical memory, and development time. These benefits increase as the amount of data in the system increases. Further, unrelated analyses can become meaningfully connected, thus presenting the opportunity for hypothesis discovery.

Using KDOM does not preclude the effective use of other Java-based bioinformatics APIs, such as BioJava [6] or BTL [7]. In fact, the functionality of these packages would complement the goal of KDOM. Where KDOM would provide the logical framework for defining and managing data definitions and relationships, other packages can assist in providing methods to obtain or manipulate this information.

Acquiring knowledge

Methods used for data acquisition are numerous. Flat files, databases, or the World Wide Web are potential sources of data. This fact necessitated flexibility in the KDOM approach. Therefore, the *BiologicalBrain*, as the responsible component, utilizes developer-implemented *BiologicalNervousSystem*s to acquire data (Figure 1).

The advantage of this design is that "nervous systems" can be swapped or combined depending on the requirements of the system. This allows a developer to simultaneously utilize information from many different sources and formats using KDOM as a semantic layer. Since this aspect of the developer's KDOM implementation is centralized, it allows relatively easy migration to a different data storage system when and if required, or the inclusion of optimizations within the acquisition procedures that will benefit the entire system.

Standardized data description and delivery systems, some of which are ontological in design and already model data using the labelled graph approach, are an area of research

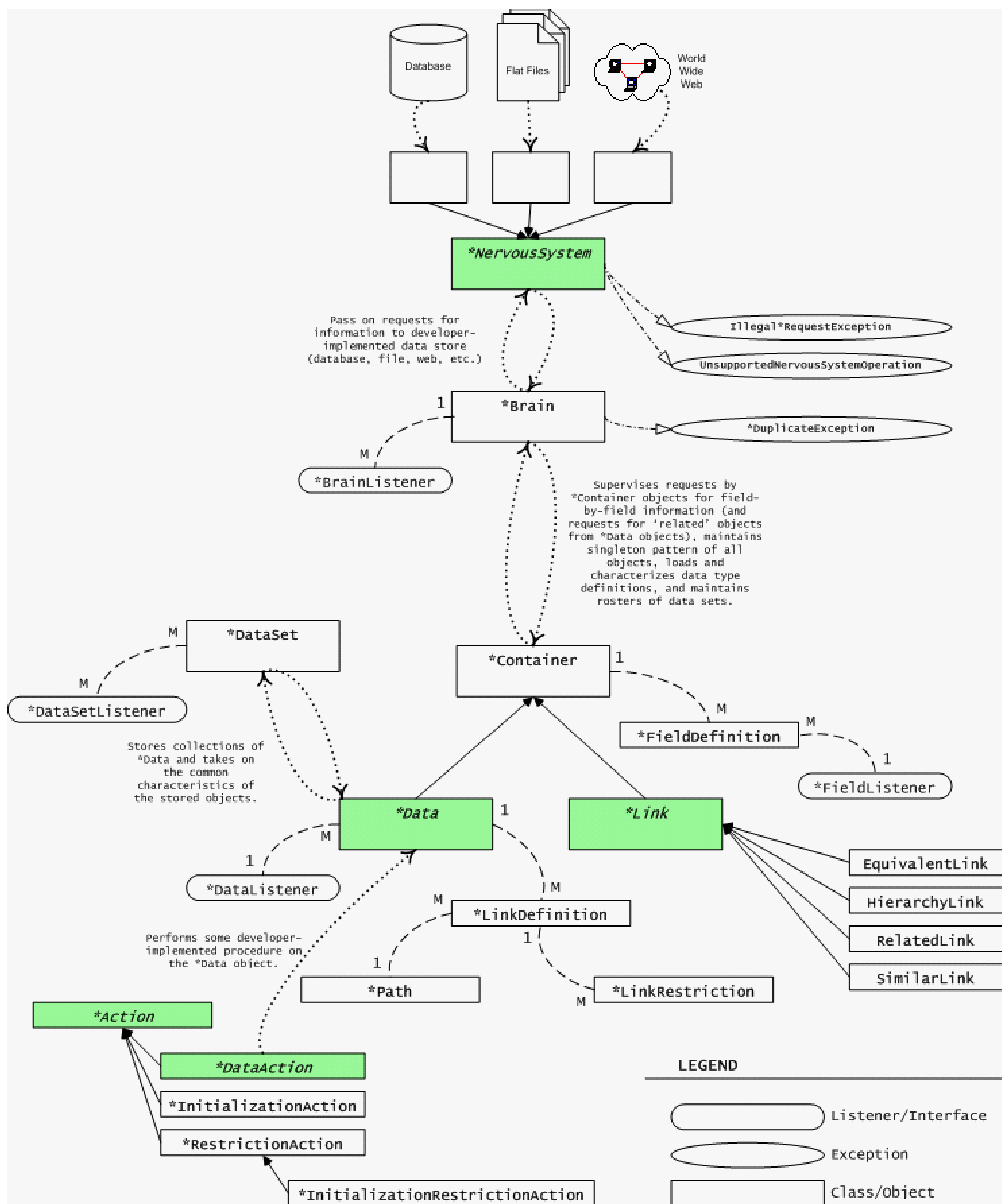

Figure 1
API architecture. An overview of the core classes that form the API. Abstract classes that require further implementation by a developer are shown in green and have italicized text. Most class names contain the word 'Biological', and so for brevity, this word has been replaced by an asterisk. Objects with a logical interaction are denoted by thick dotted lines, objects that are intrinsically related are denoted with a dashed line, object inheritance is shown using a solid line (with the arrow pointing to the superclass), and alternating dotted-dashed lines indicate an object that throws a specific exception (with the arrow pointing to the exception).

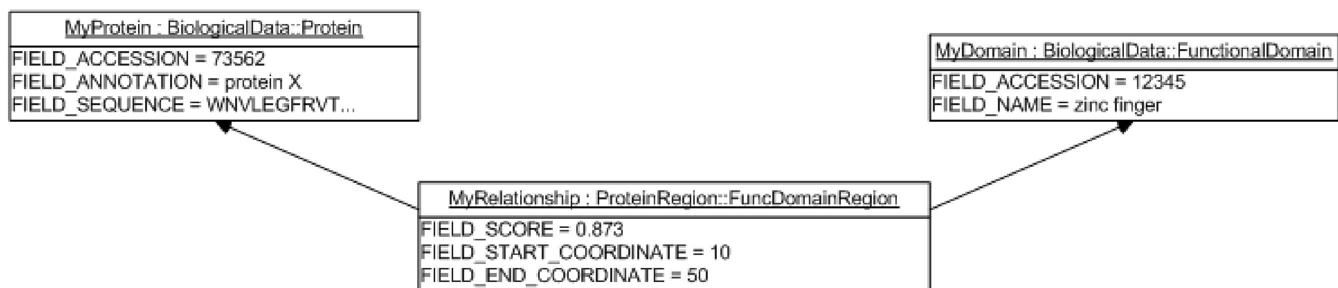

Figure 2
An example of data relationships represented within the API. A protein and a functional domain (both subclasses of BiologicalData) have a hierarchical relationship. The hierarchy relationship (a subclass of BiologicalLink) is extended to describe a protein region (with a start and end coordinate), and further extended to describe a functional domain region (which also includes a confidence score).

and development that could be particularly agreeable to a KDOM implementation. The distributed annotation system (DAS) [8], BioMOBY [9], and the Resource Description Framework (RDF) [10] are recent examples.

The separation of "what" the information is and "how" it is obtained is a fundamental approach in ontology building [4]. It is also a particular advantage in larger projects, as it makes these two needs individually transferable and manageable.

Data relationships
The relationship between data and the context in which a relationship exists is of fundamental importance. In the simplest system, relationships might be stored as an internal list inside a data object. However, the relationship would be unidirectional and the meaning of the relationship itself is not explicit.

KDOM utilizes a *BiologicalLink* class to describe the context of the relationship and provides a bidirectional association. Several *BiologicalLink* subclasses (*RelatedLink*, *EquivalentLink*, *HierarchyLink*, and *SimilarityLink*) are provided with the API to define the most common data relationships. These can be further subclassed to provide more specific context, and to define properties specific to the relationship. The API also supports multiple relationship types between the same two classes of data.

For example, a functional domain and a protein share a hierarchical (parent-child) relationship. The relationship itself may be associated with a mathematical score describing the confidence that a particular domain is truly present, and the coordinates of the putative domain in the protein sequence itself (Figure 2).

Graphical user interfaces
Typically, Java-based applications are highly GUI-oriented. The KDOM system provides abstract methods and some limited default implementations for providing context-specific interface components. For example, the display of a "gene" as it relates to a "functional domain" will differ from a "gene" as it relates to a "chromosome" (Figure 3). These take the form of extensions of common Swing components (TableCellRenderer, ListCellRenderer, and so on), making implementation straightforward (details in Figure 4).

The API also features support for displaying individual object properties in a context-sensitive manner. For example, an "annotation" property for a gene would be displayed to the user differently than the "image" property for a chromosome.

This provides a high level of GUI component sharing between separate deployments of the KDOM implementation. This is particularly valuable when a consistent look and feel across many projects is advantageous or desirable.

Other features
The API also contains a number of other features too numerous to list in full. Among them: a type-safe container class to store sets of BiologicalData and includes methods to facilitate threaded batch processing and set operations; internal row and column management, and cell and list renderers so sets of data can be quickly displayed and manipulated in tabular or list format; a multi-threaded, internal data request queue, which allows for large amounts of data to be retrieved without disrupting user interaction with the application; robust data serialization in XML, which provides a portable and efficient

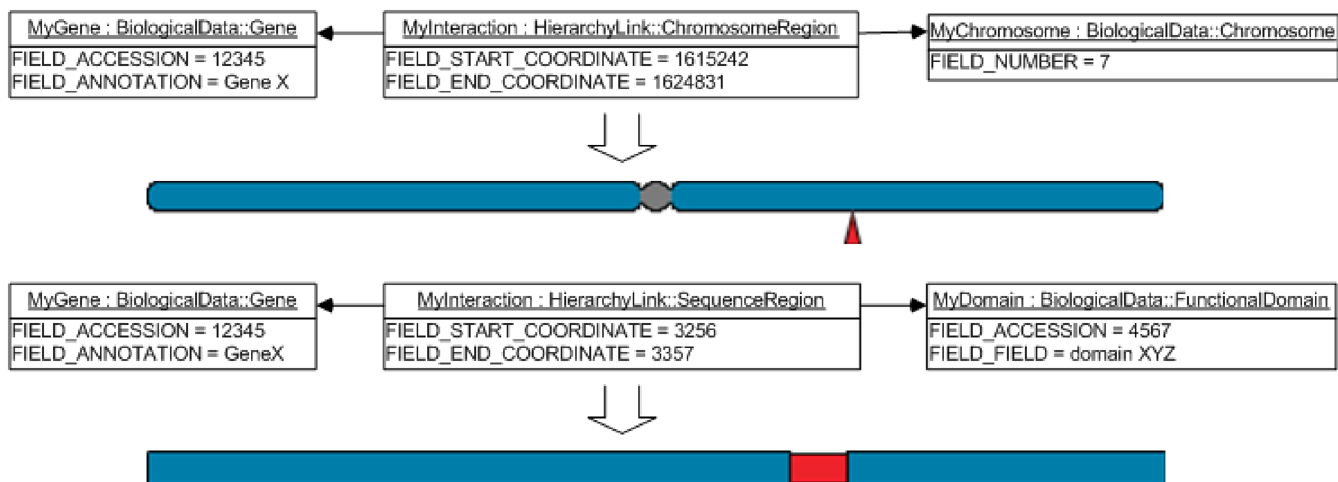

Figure 3
Graphical components implemented by the developer called using KDOM methods. The intrinsic display of a "gene" object linked to a "chromosome" or "functional domain" differs depending on the context.

method to store the results of data analyses; a centralized drag-and-drop extension for user-driven manipulation of KDOM objects; data and dataset listeners for creating responsive application components; and custom Class-Loaders that allow data definitions to be found at run-time without specifying a CLASSPATH directive.

A KDOM example

Consider the previously described example of a microarray experiment, and we have a developer that wants to implement a system where the expression of a particular gene can be visualized. For this system, we can create four objects, representing: a sample, an individual array spot, the slide, and the genes that correspond to each spot (Figure 5). We can further define the relationships between the sample and a spot, a spot and the slide, and the spot and a gene (Figure 5). Figures 6 and 7 show partial implementations of these objects in Java code. It is worth noting that in these particular implementations, the relationship is directed; and so, the relationships themselves are subclasses of a HierarchyLink, and the parent and child identities are constant (i.e. it is always the spot that is "washed with" the sample).

Now that we've defined the "what" of the biological data, we can define "how" to acquire it by implementing a *BiologicalNervousSystem* (Figure 8). Once this has been accomplished, another task requiring the same information can and should utilize this implementation. Of course, the implementation can be optimized or extended to include additional information in the future, without interfering with code that already utilizes a particular *BiologicalData* object. If one now wants to display all the genes expressed above a certain level, very little coding is required (Figure 9), and future tasks using the same information will be free of the initial overhead of defining new data types and data acquisition routines.

Ongoing development and future enhancements

The API continues to undergo active development. In particular, we intend to develop and integrate a sophisticated memory-management and serialization model so that very large-sized data collections can be utilized. Serialization of information could adopt one or more of the emerging standards (e.g. RDF) to facilitate further code reuse and utilization of information with existing packages.

The next generation of the API will include a mature "action-defining" concept (the ability is loosely implemented in the current version). This improvement would provide a developer with the ability to define task-specific modifications or interrogations of biological data within the data definition themselves. Thus, the component-sharing philosophy of KDOM would be extended to object-manipulation tasks, as well as data definitions and relationships.

For example, if one has defined a relationship between genes via homology, it may be desirable to generate an algorithm that determines whether a given similarity is important for the current task. Consider a task where an investigator requires a list of genes that have greater than 80% similarity to a particular target, and are found in the

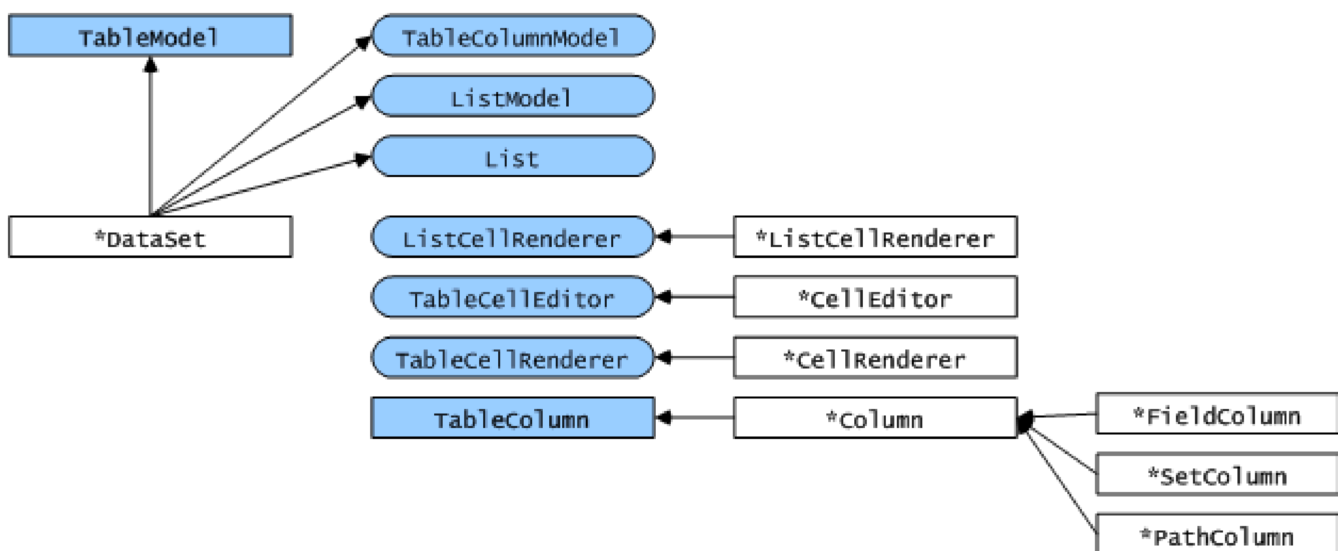

Figure 4
Classes useful for GUI development. An overview of classes useful for GUI development. Boxes denote classes, and rounded boxes denote interfaces. Inheritance and interface implementation is denoted with a connecting line (where the arrowhead denotes the superclass or interface). Classes and interfaces that are part of standard Java/Swing are coloured blue. The bottom screenshot is an example of a user interface from the DISCOVERYspace application (Zuyderduyn S et al., *in preparation*), created using the API.

mouse or human genome. The action model would define the implementation of this need (via, perhaps, a "numerical cutoff" action), and efficiently apply it to the current KDOM system. Further, these actions could be combined or linked together to create reusable analysis pipelines.

A developer repository is under construction, as of this writing, of KDOM data definitions. This repository would provide structures for common biological concepts. A developer could obtain the required objects relevant to a particular project, and would inherit their functionality, including GUI components for display of particular properties or data relationships. The developer would merely have to implement a *BiologicalNervousSystem* that conforms to their storage platform or data acquisition system to obtain the defined information. This collection of data types will consist of two general layers: an abstract layer of definitions of common biological concepts, and a further supplementary layer of extended definitions specific to the contents of common biological databases.

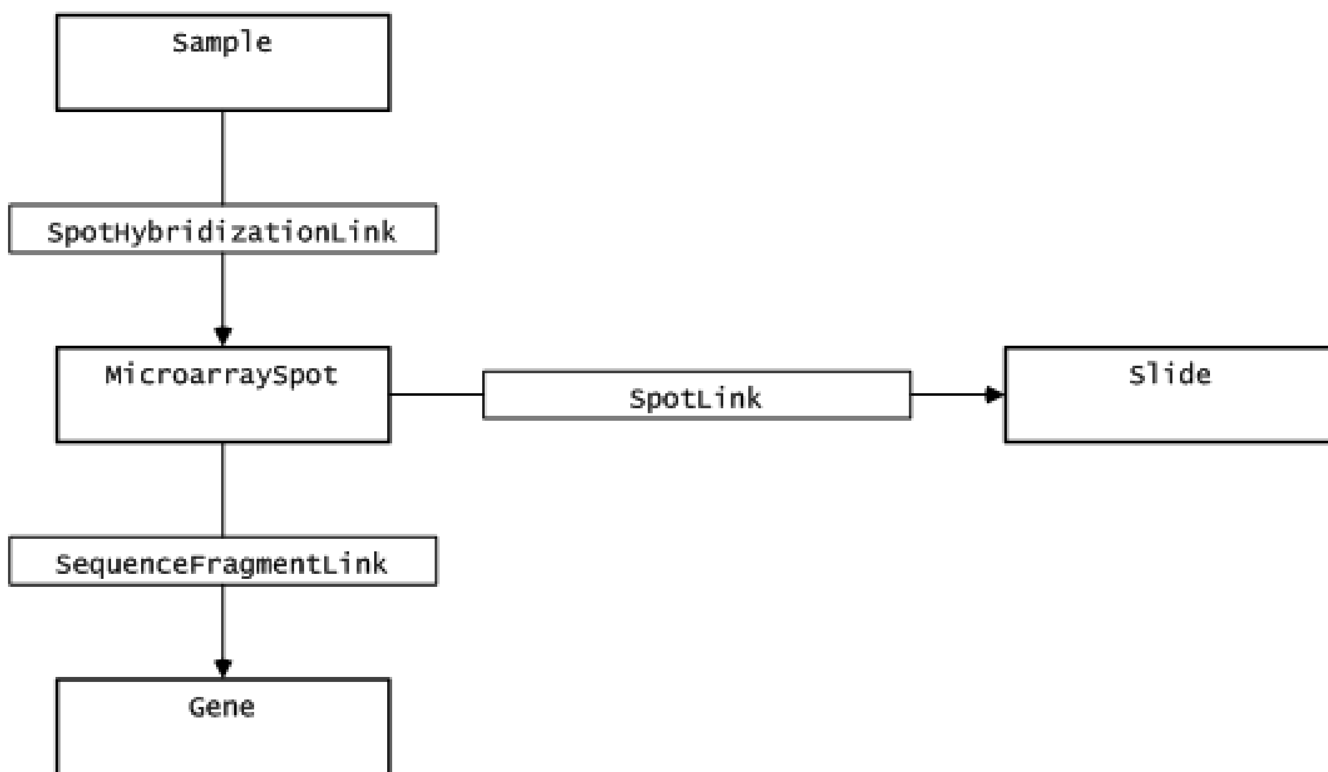

Figure 5
An example object definition set for microarray experiments. This is a theoretical set of objects that one might implement to describe a microarray experiment. BiologicalData subclasses are shown in with bold outlines, and BiologicalLink subclasses are shown with thin outlines.

Two successful implementations of KDOM

The KDOM API has been used as the foundation in an application called DISCOVERYspace (S. Zuyderduyn et al.,*in preparation*). This application uses a MySQL backend, populated using a set of recently developed parsing tools (R. Varhol et al., *in preparation*), to supply data with a focus on gene expression analysis and visualization. The application provides a flexible framework for exploring publicly available data and utilizing stored analyses (such those generated with BLAST [11] or HMMR [12]). A partial class diagram of the KDOM implementation is shown in Figure 10.

Another application called SAGEsoma (P. Ruzanov et al., *in preparation*) provides the ability to visualize gene expression data on a karyotype.

The development of SAGEsoma occurred almost completely independently of DISCOVERYspace, yet the two applications were able to share and benefit from development of the same KDOM implementation. Further, integration of SAGEsoma into DISCOVERYspace as a plugin was seamless, whilst allowing both to retain their standalone capabilities.

Conclusions

The Knowledge Discovery Object Model (KDOM) API provides an application framework for bioinformatics software development in Java. The model provides a well-defined system for knowledge management and utilization, and facilitates efficient development of medium to large-scale software projects with multiple developers, or an easily managed system for creating smaller single-developer projects with minimum overlap and overhead. KDOM complements well with other bioinformatics Java APIs.

The API provides a foundation for a logical, structured framework for data modelling, and can provide insights that result in novel hypotheses.

The API is open-source, with conditions, and can be obtained from http://www.bcgsc.ca/bioinfo/software. Documentation and code examples are also available.

```
public class MicroarraySpot extends BiologicalData {
  public MicroarraySpot( Object accession ) throws BiologicalDuplicateException  {
    super( accession );
  }

  //---------------------------------------------------------------------------//
  // Field/Property definitions                                                //
  // BiologicalField.createField( String English_name, Class data_type,       //
  //                              String description, boolean is_core_value )  //
  //---------------------------------------------------------------------------//
  public static final BiologicalField FIELD_OLIGO_SEQUENCE =
       BiologicalField.createField( "Oligonucleotide Sequence", String.class,
                                    "The oligo sequence at this spot.", true );

  public static final BiologicalLinkDefinition LINK_SLIDE =
       BiologicalLinkDefinition.createDefinition( "SpotLink", "Slide" );
  public static final BiologicalLinkDefinition LINK_GENE =
       BiologicalLinkDefinition.createDefinition( "SequenceFragmentLink", "Gene" );

  public String identity() { return "Microarray Spot"; }
  public BiologicalField getPrimaryKeyName() { return FIELD_OLIGO_SEQUENCE; }
  // Columnar definitions and GUI components would be defined here.
}

public class Gene extends BiologicalData {
  public Gene( Object accession ) throws BiologicalDuplicateException  {
    super( accession );
  }

  public static final BiologicalField FIELD_ACCESSION =
       BiologicalField.createField( "Accession", String.class,
                                    "The unique accession for this gene.", true );
  public static final BiologicalField FIELD_ANNOTATION =
       BiologicalField.createField( "Annotation", String.class,
                                    "A description of the gene.",true );
  public static final BiologicalField FIELD_SEQUENCE =
       BiologicalField.createField( "Sequence", String.class,
                                    "The nucleotide sequence of the gene.", false );

  public String identity() { return "Gene"; }
  public BiologicalField getPrimaryKeyName() { return FIELD_ACCESSION; }
  // Columnar definitions and GUI components would be defined here.
}

public class Sample extends BiologicalData {

  public Sample( Object accession ) throws BiologicalDuplicateException  {
    super( accession );
  }

  public static final BiologicalField FIELD_IDENTIFIER =
       BiologicalField.createField( "Identifier", String.class,
                                    "A unique identification code.", true );
  public static final BiologicalField FIELD_PREPARER =
       BiologicalField.createField( "Preparer", String.class,
                                    "The name of the slide preparer.", false );

  public static final BiologicalLinkDefinition LINK_SPOT =
       BiologicalLinkDefinition.createDefinition( "SpotHybridizationLink", "Spot" );

  public String identity() { return "RNA Sample"; }
  public BiologicalField getPrimaryKeyName() { return FIELD_IDENTIFIER; }
  // Columnar definitions and GUI components would be defined here.
}
```

Figure 6
Several partial implementations of BiologicalData. Partial Java code is shown for implementations of a microarray spot, RNA sample, and a Gene.

```
public class SpotHybridizationLink extends HierarchyLink {
  // Constructors defined here.

  //---------------------------------------------------------------------------//
  // Field/Property definitions                                                //
  // BiologicalField.createField( String English_name, Class data_type,        //
  //                              String description, boolean core_value )     //
  //---------------------------------------------------------------------------//
  public static final BiologicalField FIELD_EXPRESSION =
      BiologicalField.createField( "Expression", Number.class,
                                   "The level of expression.", true );
  static {
    FIELD_EXPRESSION.setRequired( true ); // the expression level must exist
  }

  // always maintain direction of relationship
  public BiologicalData getInitiator() {
    if( super.getInitiator() instanceof MicroarraySpot ) { return super.getInitiator(); }
    return super.getTarget();
  }
  public BiologicalData getTarget() {
    if( super.getTarget() instanceof Sample ) { return super.getTarget(); }
    return super.getInitiator();
  }

  public String getVerb() { return "washed with"; }

  // Columnar definitions and GUI components would be defined here.
}

public class SequenceFragmentLink extends HierarchyLink {
  // Constructors defined here.

  //---------------------------------------------------------------------------//
  // Field/Property definitions                                                //
  // BiologicalField.createField( String English_name, Class data_type,        //
  //                              String description, boolean core_value )     //
  //---------------------------------------------------------------------------//
  public static final BiologicalField FIELD_START_COORDINATE =
      BiologicalField.createField( "Start Coordinate", Integer.class,
                                   "The start coordinate of the sequence.", true );
  public static final BiologicalField FIELD_END_COORDINATE =
      BiologicalField.createField( "End Coordinate", Integer.class,
                                   "The end coordinate of the sequence.", true );
  static {
    FIELD_START_COORDINATE.setRequired( true );
    FIELD_END_COORDINATE.setRequired( true );
  }

  // always maintain direction of relationship
  public BiologicalData getInitiator() {
    if( super.getInitiator() instanceof MicroarraySpot ) { return super.getInitiator(); }
    return super.getTarget();
  }
  public BiologicalData getTarget() {
    if( super.getTarget() instanceof Gene ) { return super.getTarget(); }
    return super.getInitiator();
  }

  public String getVerb() { return "is a fragment of"; }

  // Columnar definitions and GUI components would be defined here.
}
```

Figure 7

Several partial implementations of BiologicalLink. Partial Java code is shown for implementations of the relationship between an RNA sample and a microarray spot (SpotHybridizationLink), and between a microarray spot and a gene sequence (SequenceFragmentLink).

```
class MyNervousSystem extends BiologicalNervousSystem {

  public boolean createConnection( BiologicalData requestor,
                                   BiologicalLinkDefinition link_def )
       throws UnsupportedNervousSystemOperationException, IOException {
    switch( getRequestorCode( requestor ) ) {
      case MyConstants.MICROARRAY_SPOT:
        return MySpotLinkHandler.link( (MicroarraySpot)requestor, link_def );
      case MyConstants.GENE:
        return MyGeneLinkHandler.link( (Gene)requestor, link_def );
      default:
        throw new UnsupportedNervousSystemOperationException(
            "Can't find handler for " + requestor.getClass().getName() );
    } // switch
  }

  private class MySpotLinkHandler {
    public boolean link( MicroarraySpot spot, BiologicalLinkDefinition link_def )
         throws UnsupportedNervousSystemOperationException {
      // Will create a relationship between a microarray spot and
      // and a gene, where the property of the relationship is the
      // coordinates of the spot oligo sequence within the gene sequence.
      if( link_def.getName().equals( "LINK_GENE" ) ) {

        Integer start_coord;
        Integer end_coord;
        String gene_accession;

        // Look up information from a file supplied by array manufacturer.
        // (Code omitted for brevity.)
        ...

        // Get the instance of the gene (created automatically it if it doesn't already
        // exist in the system).
        Gene gene = (Gene)BiologicalData.instance( Gene.class, gene_accession );
        BiologicalLink.instance( link_def, spot, gene,
                                 new BiologicalPropertyValue[] {
                                    BiologicalPropertyValue.instance(
                                      SequenceFragmentLink.FIELD_START_COORD,
                                      start_coord ),
                                    BiologicalPropertyValue.instance(
                                      SequenceFragmentLink.FIELD_END_COORD,
                                      end_coord ) } );
        return true;

      } // if
      throw new UnsupportedNervousSystemOperationException(
          "No method defined to relate Gene to " + link_class.getName() );
    } // method link( MicroarraySpot, BiologicalLinkDefinition )
  }

  public Object requestInformation( BiologicalData requestor, BiologicalField key )
       throws IOException {
    // Implementation here reads in requested information from a file.
    // This method is called internally by the API whenever a particular is not
    // available.
    // For example:
    if( key.equals( Gene.FIELD_ANNOTATION ) ) {
      String annotation;
      // Look up annotation for gene (Code omitted for brevity).
      requestor.setProperty( key, annotation );
      // We could also set additional properties here at the same time to
      // circumvent a future request if this is more efficient.
      return annotation;
    }
    ...
  }
  // Other components of a functioning nervous system would go here.
}
```

Figure 8

A partial *BiologicalNervousSystem*. The focus in this example is in supplying a relationship between a Gene and a SequenceFeature. This particular example is illustrative and therefore very explicit. A typical nervous system implementation is generally more dynamic.

```
// Assume a microarray slide object has been obtained in another operation
void printExpressedGenes( final Slide slide, final Number min_expression ) {

  // Define a restriction to return only highly expressed genes
  BiologicalLinkRestriction[] restriction = new BiologicalLinkRestriction[] {
    BiologicalLinkRestriction.createRangeRestriction(
                                               SpotHybridizationLink.FIELD_EXPRESSION,
                                               min_expression, Double.MAX_VALUE ) };

  for( Iterator iter = slide.getInteractions( Spot.LINK_SLIDE ); iter.hasNext(); ) {
    SpotLink link = (SpotLink)iter.next();
    String slide_coord = link.getProperty( SpotLink.PROPERTY_COORDINATE ).toString();

    MicroarraySpot spot = (MicroarraySpot)link.getLinkedData( slide );

    // Get the expression at the spot for the sample, and restrict the results to
    // those with the specified level of expression.
    for( Iterator iter2 = spot.getInteractions( Sample.LINK_SPOT, restriction );
                                                           iter2.hasNext(); ) {
      SpotHybridizationLink obs = (SpotHybridizationLink)iter2.next();
      Gene gene = (Gene)spot.getInteractionData( MicroarraySpot.LINK_GENE ).next();

      System.out.println( new StringBuffer().
                   append( gene.getProperty( Gene.FIELD_ACCESSION ) ).
                   append( " " ).
                   append( gene.getProperty( Gene.FIELD_ANNOTATION ) ).
                   append( " " ).
                   append( link.getProperty( SpotHybridizationLink.FIELD_EXPRESSION ) ).
                   append( " (" ).
                   append( SpotLink.FIELD_COORDINATE ).
                   append( ")" ).toString() );

    }
  }
}

Execute:

  printExpressedGenes( slide, new Integer( 30000 ) );

Output:

  12345 Gene X 32432.2 (A3)
  54838 Gene A 84732.1 (C5)
…
```

Figure 9

Two example procedures using a KDOM implementation. The first displays a list of supplied genes and their sequence features. The second obtains a list of genes with a particular feature of interest.

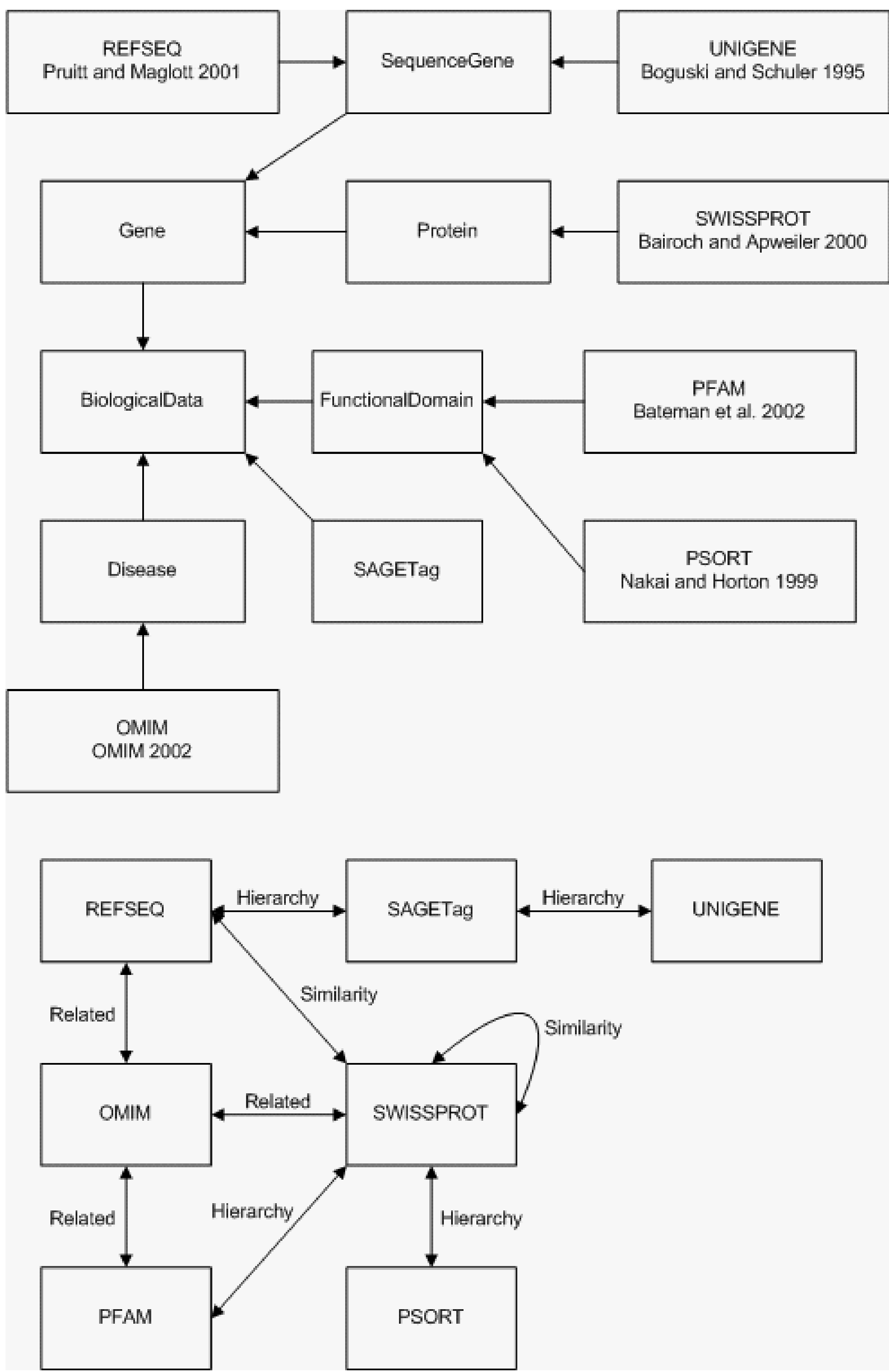

Figure 10
A partial diagram of our own implementation of KDOM. The top portion shows object inheritance, and the bottom portion shows data relationships.

Authors' Contributions

SZ conceived of, designed and coded the API. SJ provided high-level, practical considerations, resources, and scientific direction for the project.

Acknowledgements

SZ would like to acknowledge the valuable criticisms and comments of Chris Fjell, Mehrdad Oveisi, Shawn Rusaw, and particularily Neil Robertson. SZ would also like to thank Peter Ruzanov for implementing the API in the SAGEsoma project. This work was supported by funding from Genome Canada and the British Columbia Cancer Foundation.

References

1. Uschold M and Gruninger M: **Ontologies: Principles, methods and applications.** *Knowledge Engineering Review* 1996, **11(2):**.
2. Jazayeri M: **Component programming – a fresh look at software components.** In *Proceedings of the 5th European Software Engineering Conference: September 25–28 1995; Sitges, Spain* 1995:457-478.
3. Sun Microsystems: **Java Development Kit 1.4.0.** *901 San Antonio Road, Palo Alto, CA 94303* 2002 [http://www.java.sun.com/j2se/1.4].
4. Stevens R, Goble CA and Bechhofer S: **Ontology-based Knowledge Representation for Bioinformatics.** *Brief Bioinform* 2000, **1:**398-414.
5. Wiechert W, Joksch B, Wittig R, Hartbrich A, Honer T and Mollney M: **Object-oriented programming for the biosciences.** *Bioinformatics* 1995, **11:**517-534.
6. **BioJava** [http://www.biojava.org]
7. Pitt WR, Williams MA, Steven M, Sweeney B, Bleasby AJ and Moss DS: **The Bioinformatics Template Library – generic components for biocomputing.** *Bioinformatics* 2001, **17:**729-737.
8. Dowell RD, Jokerst RM, Day A, Eddy SR and Stein L: **The Distributed Annotation System.** *BMC Bioinformatics* 2001, **2:**7.
9. Wilkinson MD and Links M: **BioMOBY: an open source biological web services proposal.** *Brief Bioinform* 2002, **3:**331-341.
10. **Resource Description Framework** [http://www.w3.org/RDF]
11. Altschul SF, Gish W, Miller W, Myers EW and Lipman DJ: **Basic local alignment search tool.** *J Mol Biol* 1990, **215:**403-10.
12. Eddy SR: **Profile hidden Markov models.** *Bioinformatics* 1998, **14:**755-763.
13. Pruitt KD and Maglott DR: **Refseq and LocusLink: NCBI gene-centered resources.** *Nucleic Acids Res* 2001, **29:**137-140.
14. Boguski MS and Schuler GD: **ESTablishing a Human Transcript Map.** *Nat Genet* 1995, **10:**369-371.
15. Bairoch A and Apweiler R: **The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000.** *Nucleic Acid Res* 2000, **28:**45-48.
16. Bateman A, Birney E, Cerruti L, Durbin R, Etwiller L, Eddy SR, Griffiths-Jones S, Howe KL, Marshall M and Sonnhammer ELL: **The Pfam protein families database.** *Nucleic Acids Res* 2002, **30:**276-280.
17. Nakai K and Horton P: **PSORT: a program for detecting the sorting signals of proteins and predicting their subcellular localization.** *Trends Biochem Sci* 1999, **24:**34-36.
18. **Online Mendelian Inheritance in Man OMIM™** [http://www.ncbi.nlm.nih.gov/omim]

Publish with **BioMed Central** and every scientist can read your work free of charge

"BioMed Central will be the most significant development for disseminating the results of biomedical research in our lifetime."

Sir Paul Nurse, Cancer Research UK

Your research papers will be:

- available free of charge to the entire biomedical community
- peer reviewed and published immediately upon acceptance
- cited in PubMed and archived on PubMed Central
- yours — you keep the copyright

Submit your manuscript here:
http://www.biomedcentral.com/info/publishing_adv.asp