Representation and Visualization of Genetic Regulatory Networks

by

Dacheng Zhao

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 16, 2004

[September 2004]

Author________

ing and Computer Science
August 16, 2004

Certified by____________ [illegible]
Prof. David Gifford
sor

Accepted by__________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Representation and Visualization of Genetic Regulatory Networks

by

Dacheng Zhao

Submitted to the
Department of Electrical Engineering and Computer Science
August 16, 2004
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present a new framework, Sonnet, for the interactive visualization of large, complex biological models that are represented as graphs. Sonnet provides a flexible representation framework and graphical user interface for filtering and layout, allowing users to rapidly visualize different aspects of a data set. Many previous approaches have required users to write customized software in order to achieve the same functionality. With Sonnet, once features of interest are identified, they can be captured as figures for offline presentation. We demonstrate the application of Sonnet to the visualization and manipulation of transcriptional regulatory networks in yeast. Sonnet is particularly well adapted to this application as native presentation of these networks yields dense and difficult to decipher results.

Thesis Supervisor: David Gifford
Title: Professor of Computer Science and Engineering

Acknowledgements

I would like to express my profound gratitude towards my advisor, Professor David Gifford, for his support, mentorship and guidance. Prof. Gifford's vision and wisdom are extraordinary, and his invaluable feedback kept me focused. I would also like to thank Georg Gerber, a graduate student in the laboratory of Prof. Gifford. Georg's work in computational biology was the original inspiration for the project. Moreover, Georg's advice during all stages of this project—design, implementation, and write-up—was incredibly helpful and had a tremendous impact on the project. I would also like to thank my parents, Li Chen and Wei Zhao, and my brother, Derek Zhao, for supporting and nurturing me my entire life. My parents fostered my curiosity and instilled in me a deep appreciation for science and learning. Last but not least, I would like to thank my fiancée, Bonny Lee, for her immeasurable support. Her devotion and care have always been a great source of inspiration for me.

Table of Contents

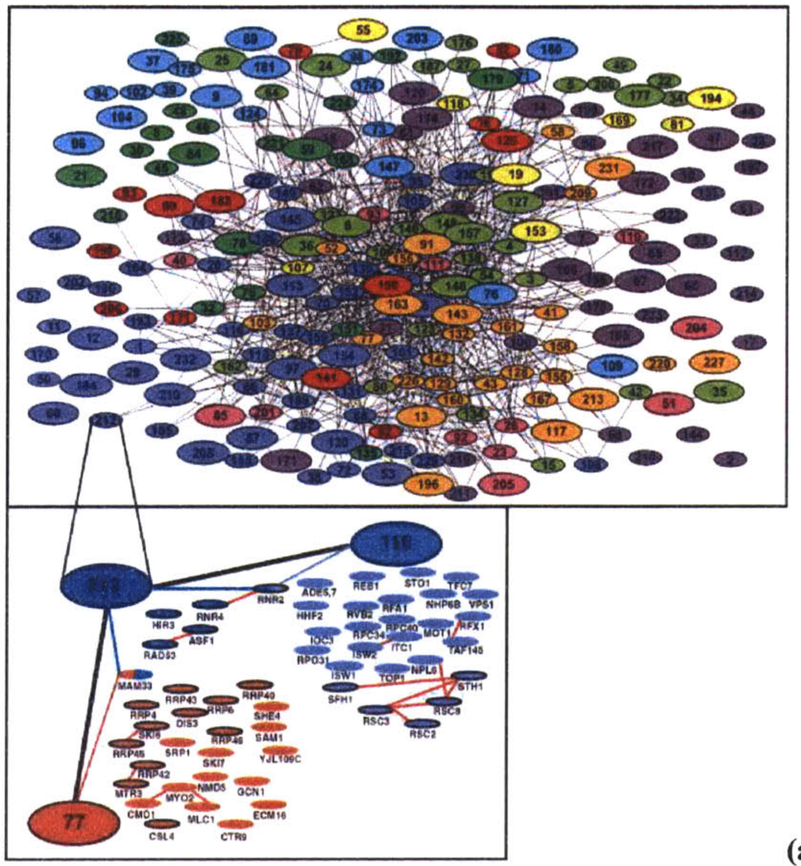
1.0 Introduction

With sequenced genomes as references, scientists now measure an assortment of genome-wide phenomena including expression of genes, binding of transcription factors to DNA, and protein-protein interactions. The enabling factor for these measurements has been the dramatic advancement of high throughput biological assays. Robotics and miniaturization have increased the throughput of assays spectacularly, while the natural progression of the science has increased the breadth of assays. Most importantly, the commoditization of assays has empowered scientist to begin probing cells under a wide variety of environmental conditions, allowing much more sensitive resolution of functional systems that lie dormant in the rich media conditions under which laboratory cells are typically maintained.

This boon in data has engendered a flurry of computational research geared at functionally annotating genes, discovering genetic pathways and understanding control mechanisms. Earlier work in this field focused on learning from homogenous data, such as protein interaction networks based on protein-protein interaction data and gene clusters based on gene expression [1-3]. Recent work has tried to elucidate higher level organization, such as genetic regulatory networks, from disparate data sources [4-6]. Regardless, visualization of the resulting interaction networks, whether simple and homogeneous or complex and heterogeneous, is of paramount importance. As assays become standardized, inferred networks are moving into the forefront of research; an inferred networks is not merely an incidental finding, but the central discovery in a research work, and graphs of networks are often the most important way of summarizing and visualizing the data.

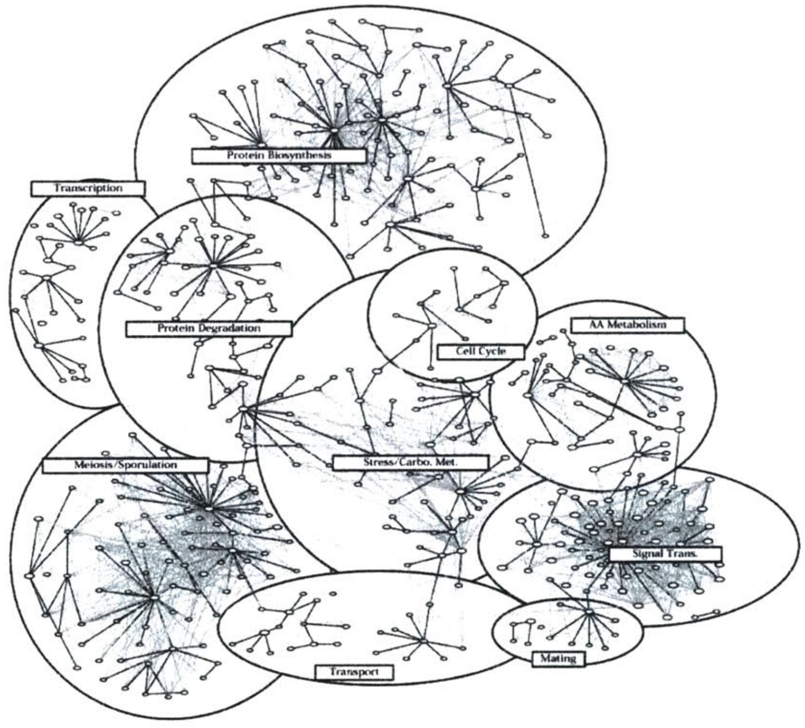
Unfortunately, the creation of informative and visually appealing static graphs from the increasing variety and quantity of high throughput biological data sources is becoming intractable with current approaches. Although layout algorithms and software can readily render large numbers of nodes and edges, the end result is often a dense, unintelligible, spaghetti-like graph (see Figure 1). The sheer number of edges and nodes overwhelm the capability of the reader to discern organization and to understand the significance of the data.

To avoid unintelligible graphs, researchers often carefully hand select relevant portions of the data to visualize, and then painstakingly layout the graph by hand. Alternatively, they can render the graph with an automatic layout program and try to adjust parameters until some satisfactory output is achieved, an arduous process at best. Unfortunately, even the best efforts are no match for the rapid growth in complexity of the models. For instance, a 2003 study on genetic regulatory interactions in *Saccharomyces cerevisiae* (baker's yeast) found over 1500 regulatory interactions [6]. These interactions were further reduced into approximately 100 co-regulated, co-expressed genetic modules, which were presented in a graph with roughly 100 nodes and 200 edges. However, this research only explained regulation for 600 of yeast's 6000 genes. Extrapolation of these figures and recent unpublished research, suggest that over 1000 such regulatory modules exist in yeast. When this analysis is applied to human data, with five times as many genes and untold-fold more complex regulation, one might expect to discover tens of thousands of modules. Even in the most conservative case, any graph like those in Figure 1 would be hopelessly clogged with information.



77
118
ADE5,7
REB1
STO1
TFC7
RNR2
RNR4
HIR3
ASF1
RAD53
HHF2
RVB2
RFA1
NHP6B
VPS1
IOC3
RPC34
RPC40
MOT1
RFX1
RPO31
ISW2
ITC1
TAF145
ISW1
TOP1
NPL6
STH1
SFH1
RSC3
RSC8
RSC2
MAM33
RRP43
RRP6
RRP40
RRP4
DIS3
SHE4
SKI6
RRP46
SAM1
RRP45
SRP1
SKI7
YJL109C
RRP42
MTR3
NMD5
GCN1
MYO2
CMD1
MLC1
ECM16
CSL4
CTR9

(a)



Protein Biosynthesis
Transcription
Protein Degradation
Cell Cycle
AA Metabolism
Stress/Carbo. Met.
Meiosis/Sporulation
Signal Trans.
Mating
Transport

(b)

Figure 1. Typical visualizations of yeast interaction networks

a) Interaction network describing one of the first large scale mass spectroscopy assay of yeast protein complexes [7]. Ovals represent protein complexes and the number and size of the ovals relate to the number of proteins. Edges between two ovals exist if the complexes share a protein. The ovals are color coded according to functional categories: red, cell cycle; dark green, signaling; etc. Although the publication is notable for the data itself, this diagram is only suitable for conveying vague notions that the protein interaction network contains somewhat large clusters (many with 100+ proteins), the network is distributed across many yeast functions, and the organization of this network is complex.

b) Recent work attempting to reveal the organization of the yeast molecular network by combining heterogeneous data [5]. Genes are organized into functionally related groups represented by small white ovals. An edge exists between two ovals if the two groups share more than 1/3 of their genes. Groups of genes are further organized into large gray ovals which are labeled by their GO classification. Although this study is noted for using a wide variety of genome wide data to create a reasonable model without any *a priori* knowledge about the networks, this particular interaction diagram presents only a "cartoon" of the data with a few placid observations. The reader garners that there are a lot of interactions, interactions might follow a power log distribution (star topology), and general biological systems (Transport, Mating, etc) are connected.

The rapidly growing magnitude of datasets is only part of the problem when visualizing models. As mentioned earlier, current models are also rapidly growing in dimensionality. High-throughput assays have become more accessible, and as a result, data is beginning to be gathered under a variety of environmental conditions. Even though these conditions are sometimes naively lumped together when creating models, in many cases, such presentation is misleading or even erroneous. In addition, models are often labeled with additional layers of supplementary or corroborating data. For instance, a common practice when building regulatory networks is to functionally categorize learned structures based on their MIPS or GO categories [8, 9]. Current approaches of using textures, colors and sizes to denote different attributes are cumbersome. Invariably, the graphical metaphor for how the representation relates to meaning confuses the reader. In the worst cases, visually hard to detect variations in color or size distort the meaning of the model.

Difficulties with large heterogeneous networks aside, one should also consider the requirements and practices of researchers who create the models. Model building occurs in an iterative "boot-strap" fashion. Researchers usually begin by qualifying the raw data. Then, exploratory analysis, based on previous research, might be performed to determine areas that merit further attention. Finally, new computational techniques are developed to analyze the dataset. After a model is resolved, it is often corroborated with other data sources or research. At each of these steps, intermediate models may be built and preliminary graphs created. These models provide great insight into how the analysis works, and may greatly aid a reader in understanding the results. However, for lack of a convenient way of representing data, preliminary models are usually not incorporated into the final visualization.

Fortunately, the inherent complexity of large heterogeneous models can be exploited to provide a solution for many of these problems. Instead of static and meticulously hand-perfected images, these models might be best described and understood by a series of automatically drawn graphics, interactively created by the end user. Using various features to filter the high dimension data, a reader could explore the data in a series of intuitively connected fragments. Specific layouts and filters could be programmatically defined for a given model so as to focus the reader's attention and aid them in their exploration. However, the bulk of the learning would be automated, based on interactions from the user. Moreover, interactions of the network could be synchronized with novel ways of summarizing attributes of nodes and edges (see Figure 2) to create a seamless environment for viewing complex models.

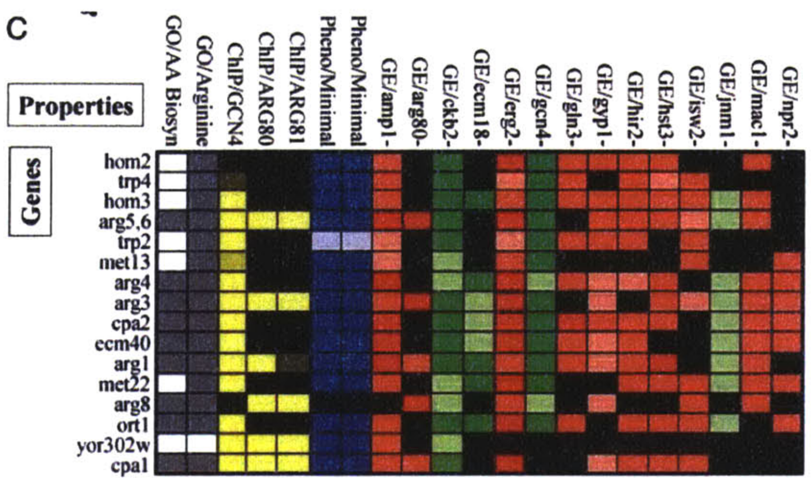


Figure 2. Example tabular data summary from Tanay et al[5]
This table represents one module of genes from Tanay et al. Building upon "Eisengrams", this table extends the concept of color density relating to the value of an attribute [3]. Genes are listed along the y-axis while different properties—GO category, protein binding (ChiP), phenotype sensitivity, transcription factors/conditions—are listed along the y-axis. Different properties have different colors as well as different interpretations for shadings. For example, red and green (and shades thereof) correspond to the usual down and up regulation of a gene with respect to the transcription factor gene. Meanwhile, yellow represents the strength of the binding of a transcription factor to the upstream region of a gene. This representation not only summarizes the data well, it allows the user to readily see which properties contributed most to the inclusion of a gene in the module.

1.1 Background

Even though the main mechanisms for cellular regulation have been known since the 1950s, many important aspects of regulation are still uncharacterized today [10]. The data and analysis in this section will deal with the components of the central dogma of biology; a very simplified view is presented (for a more realistic view see Alberts et al [11]). Protein transcription factors bind to the upstream region of genes, enabling (or obstructing) transcription. During transcription, DNA is transcribed into mRNA by various protein apparatuses. mRNA may undergo post-transcription modification such as intronic excision and alternate splicing in eukaryotes. However, the relatively unstable mRNA molecule is then immediately translated into proteins via ribosomes. Finally, proteins may be modified post-translationaly.

Modern molecular biology is essentially the story of the biological assay. Hundreds of assays have been developed to capture various snapshots of cellular state. Scientist can now measure the genome-wide levels of transcription and translation, as well as proteome-wide binding between protein and protein, and binding between protein and DNA (transcription factors and DNA). More importantly, advancements in technique and machinery have radically increased the throughout of assays. Unfortunately, not only does this windfall of data overwhelm analysis techniques, but also high-throughput data usually includes much more noise. Hence, new emphasis is being placed on computational analysis to learn relationships hidden inside noisy data.

1.1.1 Biological assays

Traditionally, molecular biologist elucidated the inner workings of cells one interaction at a time with thorough and often ingenious assays. Although traditional assays are often complex and time-intensive, molecular biologists have developed the finesse and experience to measure proteins and DNA accurately. In general, high-throughput assays represent a distinct shift in philosophy from traditional assays. Assays are miniaturized, and hundreds to thousands of the same assay can be performed in parallel with the aide of robotics. Although high-throughput assays might reduce certain forms of human error, more insidious forms of noise arise. For example, in single experiments, the parameters of the protocol are usually fine-tuned to best resolve the particular interaction in question. In the case of high-throughput assays, parameters may not be ideal for any individual assay, but rather satisfy global constraints.

Although not an assay in the conventional sense, the Sanger method for sequencing DNA is the forefather of most modern biological assays [12]. The Sanger method, along with shot-gun sequencing techniques developed in the 1990s, allowed the rapid sequencing of the genomes of many organisms. Notably, the yeast genome was completed in 1996 and a rough draft of the human genome was completed in 2001 [13-15]. At the time of writing, hundreds of genomes have been completed, including those of 35 eukaryotes[1]. Sequenced genomes are often referred to as parts lists; however, a more accurate description would be a parts list where all annotations—spaces, punctuation, etc.—have been removed. Nonetheless, even without any annotation, sequenced genomes are an extremely critical reference for many assays.

With accurate sequenced genomes in place, scientists were able to develop assays to learn state information about transcribed genes. Before the early 1990s, DNA and RNA products could be verified via Northern and Southern blot assays [16]. Unfortunately, these assays could only accurately resolve a handful of DNA or RNA products. The breakthrough came when scientists at Affymetrix were able to attach short DNA bait sequences to substrates at high density [17-19]. Then, the entire RNA detritus from a collection of cells was washed over the slide. When, segments of RNA bound to the baits, antibodies tagged with fluorescence would bind to the RNA and reveal their presence, and hence, which genes were being expressed. Many similar microarray technologies are based on these basic principles, and the latest versions of microarrays can resolve well over one hundred thousand unique sequences per slide [20, 21].

---

[1] http://www.ebi.ac.uk/genomes/

Currently, reliable and accurate protein microarrays, where antibodies specific to various proteins serve as baits, are still being developed.

Assays have also been developed to determine the binding of transcriptional regulatory proteins to genetic promoter regions on a genome-wide scale. This is particularly important because it provides direct physical evidence for which transcription factors regulate which genes. In the Chromatin Immunoprecipitation (ChIP) assay, a chemical treatment causes covalent bonds to form between all protein-DNA complexes in the cell [22]. Then, antibodies specific for a transcription factor are used to co-precipitate the transcription factor and all DNA fragments bound to it. Finally, the covalent DNA-protein bonds are reversed, and the DNA fragments originally bound to the transcription factor are assayed using a microarray to determine which gene promoter they bound to.

Proteins often bind together into complexes that form the machinery for many cellular processes, including genetic regulation. There are several common methods for measuring protein-protein interactions and protein complexes. One method is two-hybrid assays, in which the genes of the proteins of interest are inserted into two special genetic constructs [1, 2]. If the two proteins products interact, a reporter gene[2], whose protein product can be detected, is expressed. Another popular method is mass spectrometry, which separates ions by their mass to charge ratio. Although mass spectrometry has been used since the late 1800s, recent innovations have allowed scientist to use it to probe large portions of the proteome of an organism [23]. At the moment, there is considerable debate as to which technology produces less error-prone results.

[2] A common reporter gene is the gene for green fluorescent protein, a molecule which fluoresces green when exposed to blue light.

Besides assays that measure the presence or absence of binding between molecules or the presence or absence of molecules themselves, assays have been developed to probe large scale genetic interactions. One mainstay assay of molecular biology is the knock-out experiment, in which a gene of interest is permanently turned off in order to see how the lack of the gene affects the rest of the cell. Now assays have been developed that allow knock-outs to be tested in mass [24]. The most promising approach is synthetic lethal screens, in which two viable single knock-out organism are crossed to form a double knock-out organism. The basic premise is that if the two genes are located in the same pathway, the organism will die; however, if the genes are in different or redundant pathways, the organism might live. Unfortunately, given the complexity of biological pathways, making these assessments is extremely difficult.

1.1.2 Computational analysis

With respect to learning from large biological datasets, computational analysis is first used simply to normalize data and to create error models. Comparing data from different runs of the same assay may already be a challenge; however, the situation is far worse in practice where biological datasets are often cobbled together from a variety of experiments under any number of conditions. If data is normalized unfairly or error models are inaccurate, further analysis can be moot, yielding insignificant results.

Normalization aside, one of the first problems tackled with computational analysis was clustering of expression profiles generated by microarrays. When the first large datasets were generated, many genes had yet to be annotated with a function. One application of clustering was to functionally annotate unknown genes based on the annotation of genes with which they co-clustered. Basic clustering techniques were able to yield significant results. For example, simple extensions of hierarchal clustering were sufficient to annotate many genes in yeast, while extensions of correlation were adequate to automatically distinguish between two types of leukemia [3, 25]. Since then, almost every method in the pattern classification book has been applied to expression clustering, without significantly better results [26].

In earlier works, co-expression was taken to imply co-regulation or at least co-function. However, since this is often not the case, recent works have focused on using additional information to find clusters of genes, known as modules, which function together. One method for module discovery is to seed a module with statistically significant data, and then expand the modules in a probabilistic fashion [6, 27]. This hybrid approach uses ChIP (protein-DNA binding) data as the core of modules, and expression data to expand modules. The result is that the discovered modules tend to be significant (i.e., have significant overlap with existing biological knowledge); however, the hard requirement of DNA-protein binding, and the relative dearth of ChIP data, results in many genes that are not represented in any module.

Other approaches involve probabilistic graphical models. One group has successfully abstracted this approach so as to make it applicable to many different types of data with only minor changes to the algorithm [28]. So far this approach has been applied to solely expression data, expression data and sequence, and expression data and protein-protein interaction data [4, 29, 30].

1.2 Previous Work

Relevant previous research includes both software for visualizing networks as well as the actual graphs seen in publications. Given this immense space of research, we will focus on software and graphs that pertain to genetic regulatory networks.

1.2.1 Visualization Software

As we will discuss extensively in this thesis, creating a useful graph is a complicated process involving data management, graph drawing[3] (i.e. layout rendering), and many other details. Typically, after a network has been discovered and validated, scripting tools are used to import the data into one of many visualization or layout programs[4]. Then networks are usually meticulously refined by hand, which often entails iteratively repeating the entire layout process, until the author deems the figure sufficiently polished.

At one end of the spectrum are generic graphics software packages—Adobe Photoshop, Microsoft Draw, etc—where the user uses a palette of shapes and colors to draw cartoons of networks. The most powerful example is Microsoft Visio[5], which was used to create many of the figures in this thesis. Visio has many advanced features such as dynamically connecting and rerouting lines between points, and templates for common

[3] It should be noted that graph drawing (creating a legible figure from lists of edges and nodes), despite the passing treatment it receives in this thesis, is far from a simple problem. The complexity arises from the introduction of constraints—minimization of bends in an edge, minimization of edge crossings, etc. [31]. For an overview, see this tutorial:

http://www.cs.brown.edu/people/rt/papers/gd-tutorial/gd-constraints.pdf

[4] http://directory.google.com/Top/Science/Math/Combinatorics/Software/Graph_Drawing/

[5] www.microsoft.com/office/visio/

figures. Unfortunately, the amount of human interactions that these programs require limits their use to creating figures with relatively small numbers of edges and nodes.

For the more programmatically inclined, there are also a number of graph drawing and layout software packages. The most popular of these are GraphViz[6] and Pajek[7] [32, 33]. Both packages are capable of automatically rendering graphics of large networks and can output the graphics in a wide variety of formats. GraphViz is lauded for the sophistication of the implemented graph rendering algorithm and for the flexibility a user has in specifying the appearance of the graph. Meanwhile, Pajek has implemented graph analysis algorithms, and can be used to find clusters of nodes, and then reduce edges between clusters in order to highlight the true relationship of clusters.

Other applications have built upon these graph layout packages to create more powerful visualization tools. There are many software packages that provide GUIs which allow the user to interact with networks derived from a specific type of data, or a strongly constrained network. For example, GeneNet is a formal model for fully specifying components (gene, protein, transcription factor, promoter, etc.) in a genetic network, along with a companion database of specifications for components gleaned from research and literature [34]. Visualization and interactions with this network are achieved by a complimentary Java application, GeneNet viewer[8]. Likewise, a large number of protein centric interactions have been gathered together in the GRID (General Repository of Interaction Datasets) database [35]. The interactions from this network of roughly 7000

---

[6] http://www.research.att.com/sw/tools/graphviz/overview.html

[7] http://vlado.fmf.uni-lj.si/pub/networks/pajek/

[8] http://wwwmgs.bionet.nsc.ru/mgs/gnw/genenet/

nodes and 26000 edges can be visualized using Osprey[9]. Although Osprey does not exhibit elaborate layouts, it does allow a user to examine networks in an intuitive fashion, where individual nodes (proteins) in the network are linked to a wide assortment of non network information.

Finally, there are some more ambitious projects that attempt to provide more general and extensive user interfaces to networks and network layout tools. One such project is daVinci, a "universal, generic visualization system for automatic generation of high-quality drawings of directed graphs" [36]. daVinci does possess an impressive set of methods for drawing graphs; however, it is not open source, rendering it somewhat unsuitable for extension in an academic setting. Another promising example is Cytoscape [37]. Cytoscape [10] was originally geared towards visualizing genetic interaction networks, however, the newest version (scheduled to be released in the fall of 2004) promises to be a full-fledged platform for interactively modeling arbitrary networks. Most importantly, the new version of Cytoscape will be completely open source, and will build upon the open source Java graph layout package GINY[11] (Graph INterface LibrarY). The original version of Cystoscape had been built upon the proprietary yFiles[12] Java graph layout package.

[9] http://biodata.mshri.on.ca/osprey/servlet/Index

[10] http://cytoscape.org/

[11] http://csbi.sourceforge.net/

[12] http://www.yworks.com/en/products_yfiles_about.htm

1.2.2 Graphs of networks

Graphical displays of networks are as diverse as the myriad tools that created them. The following figures have been selected for their relatively clear presentation and appeal. Nonetheless, even among the "cream of the crop" graphs, the weaknesses of static images are obvious. Although Sonnet does not provide a direct solution to these problems, it does provide a mechanism with which end users can create and interact with many graphs, ameliorating the main problem of trying to represent too much data in one graph.

The graph in Figure 3 illustrates many of the tradeoffs needed to create a graph. While the column of the graph aids in the distinction between transcription factors and modules, it also creates many unnecessary edge crossings. Fortunately, the number of nodes is relatively small, and this shortcoming is not excessively distracting. However, the layered annotations can be ambiguous. For instance, it is unclear if red edges (regulation supported by literature) are also inferred from the data. Still, the main drawback of this graph is that the presentation in graphical form does not provide much additional information. The crisscrossing lines sufficiently break up the flow so that connections have to be meticulously traced. If the image must be static (i.e. for publication), the data might be better presented using a simple table, in which modules are indexed by transcription factors and modules are colored to show correlation between transcription factors. An alternate solution would be to use an interactive graph where the nodes and edges only represent interactions, and users attain additional information by browsing the network.

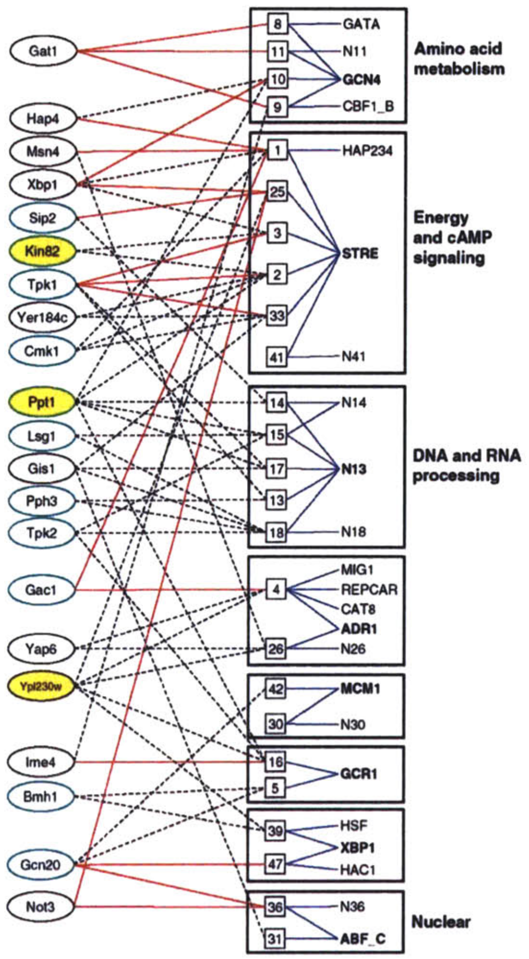


Figure 3. Example network graph from Segal et al [29]
This graph represents a network. Regulation factors are represented by ovals and modules of genes are represented by small squares. Groups of modules that share motifs are represented by the larger boxes. If broad functional annotations exist, they are written to the right of the large boxes in bold. Color is used extensively to differentiate objects. For instance, green ovals represent transduction molecules and black ovals represents transcription factors. Ovals that are shaded gold represent factors that that have been experimentally confirmed. Red edges represent regulation that is supported in literature, while black dotted edges represent inferred regulation.

Figure 4 is an example graph in which the author has taken special care to spatially arrange related objects. Initially drawn with a graph layout program, the author has rearranged objects to pull apart modules so that modules appear to be in distinct clusters. The end result is visually appealing, however, upon closer inspection, the hard assignments of transcription factors and modules to functional categories seem to be arbitrary. For example, one interesting conclusion a reader might draw is that transcription factors that span multiple functional categories might coordinate those functions; but without other evidence or annotation, this may be an artifact of the drawing rather than a true finding of the data. A series of graphs could reduce this problem by showing possible assignments instead of one assignment. Furthermore, interactivity could improve this graph if one were able to "collapse" modules. For instance, if the user could interactively create layouts where all the modules within a functional category were represented by one node, then the effect of transcription factors spanning functional categories would be readily apparent.

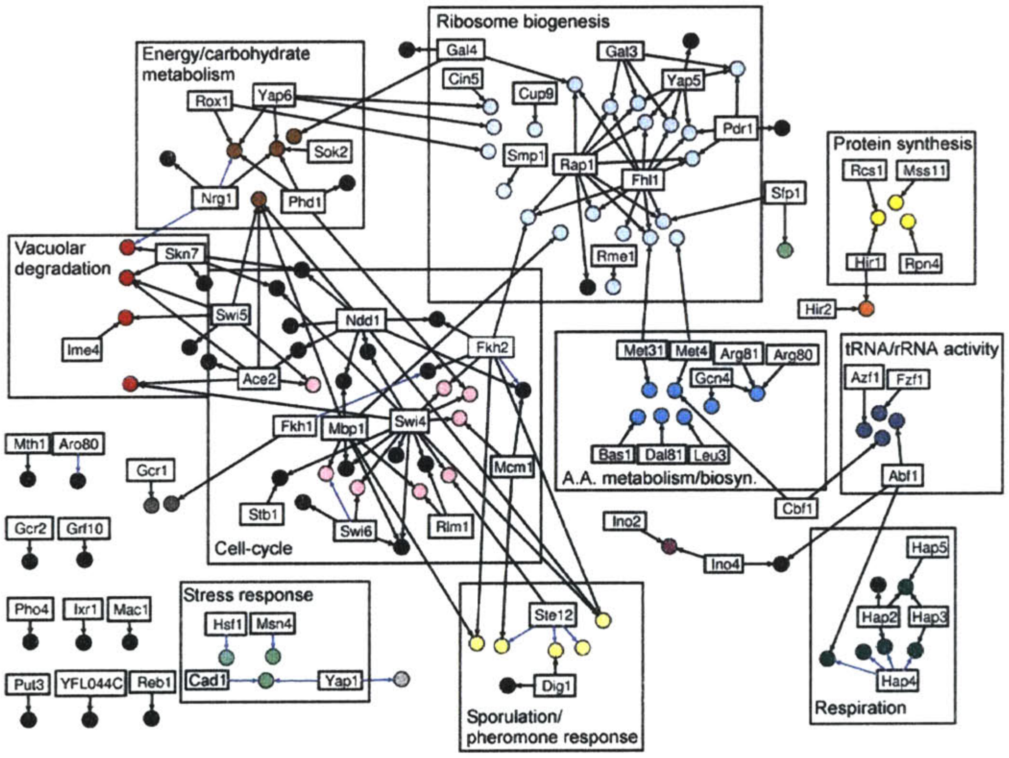


Figure 4. Network graph from Bar-Joseph et al [6]
This graph represents a network discovered by the GRAM algorithm from data in rich media conditions. Modules of co-expressed and co-regulated genes are represented by circles. Transcription factors are represented by small rectangles. Directed arrows represent regulation, where blue signified activation and black represents unknown function. Modules are also color coded by functional categories. When several modules have the same functional category, they are grouped, and a larger box surrounds them, with the name of the functional category. Black circles represent modules of mixed, or unknown function. This graph contains 68 transcription factors and 106 modules that represent 655 genes.

The graph in Figure 5 is an example where the author has done relatively little to rearrange objects after a layout program has rendered the graph. Ultimately, as networks grow in size, this may become the predominant approach. However, it is obvious that more care is required during layout to make the graph tractable. The use of line thickness to convey probability is a clever idea; unfortunately, the reader is left with many faint lines along with many heavy lines. Moreover, the reader gains almost no information from the spatial relation of objects. If the reader gains no information from spatial relationships, one has to wonder if the data could be better conveyed through a table.

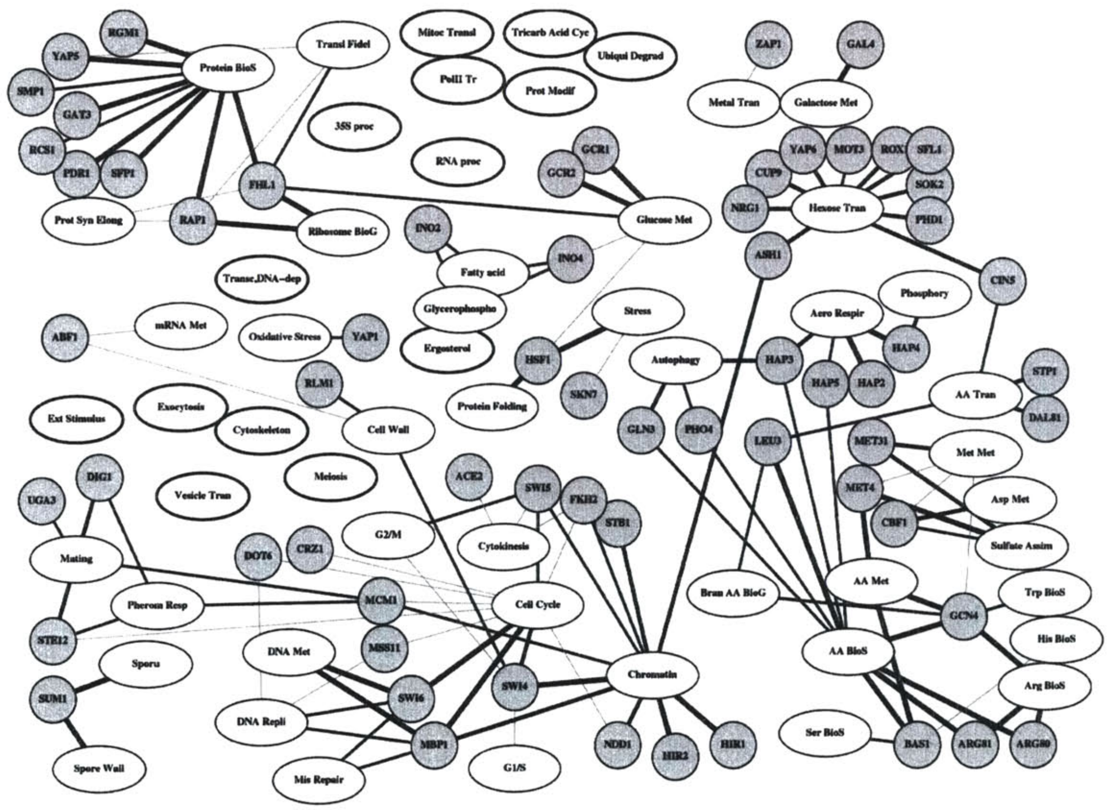
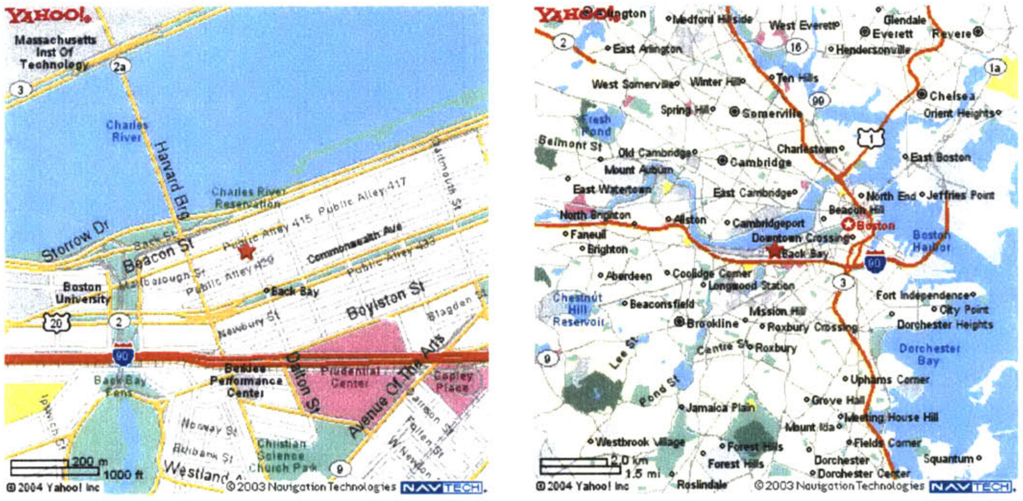


Figure 5. Network graph from Tanay et al [5]
This is a network of functional modules learned by the SAMBA algorithm from a wide variety of data sources. Modules of genes are represented by white ovals with the name of the represented process. Gray circles represent transcription factors. The thickness of the line connecting transcription factors to modules is proportional to the p-value of the enrichment of the

transcription factor in the module. An interactive version of this figure with additional data can be seen online[13].

Ultimately, one way to visualize the data has already been implemented by online mapping websites (see Figure 6). As the user browses a map, salient features are highlighted and emphasized depending on the level of resolution. This example of an online map automatically summarizes data as the user expands the scale; however, detailed features and relationships are presented as the user focuses on specific areas. For purposes of visual presentation, this map simplifies a complex network and encourages exploration.



[13] http://www.cs.tau.ac.il/~rshamir/samba/yeast_system.html

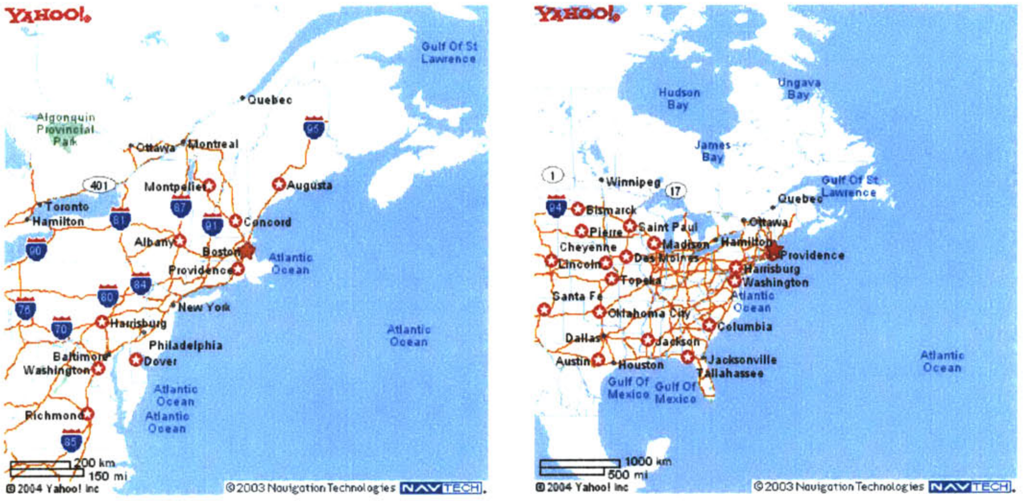


Figure 6. Maps of the location of the author's home (at the start of the project)
These are maps depicting the author's home (at the time of writing) at varying levels of detail. The graph automatically highlights the relevant details as the user changes the level at which he wishes to view the data. Some connections exist at multiple levels (interstate highways) whereas some connections are unique to one level (street names).

1.3 Objective

The objective of this work is to create a tool that facilitates the creation of excellent interactive graphical displays of high dimensional biological data, as well as aid in the interpretation and exploration of these graphical models. Practically and programmatically, this objective can be broken down into three sub-objectives. First, a logical framework will be defined and implemented for representing the data in these models. Then, this programmatic framework will be extended with interfaces that allow the user to easily filter and organize the models. Finally, this software will be coupled to an existing graph layout program so that the user can modify the model and view the graphical representation in an interactive fashion. In addition, the software tool will provide programmatic interfaces, so that users can focus on the possible layouts of the model, and so that new summaries of data represented by nodes and edges can be added. Although this project will focus on models of genetic regulation in yeast, the framework this project defines and the software this project implements can be applied to a wide variety of graphical display problems.

Excellence in graphical displays deserves further attention and discussion. As Professor Edward Tufte's explains in his landmark book on statistical graphics, graphical displays should [38]:

- induce the viewer to think about substance rather than about methodology, graphic design, etc.,
- make large data sets coherent,
- encourage the eye to compare different pieces of data, and
- reveal the data at several levels of details, from broad overview to fine structure.

Although Tufte was referring to more conventional graphics, these guidelines still serve well when thinking about interactive graphical display of complex biological models. For this project, excellence in interactive graphical displays of complex dimensional data also entails that the software should allow one to:

- filter out extraneous data in a intuitive and simple manner
- easily compare different representations of the same model
- quickly browse through the model based on selected information
- understand the representation of all interactions by simply browsing the network

Put another way, the objective of this project is to help the reader create a series of excellent graphics from a model that is too complex to be represented by one static graphic. As discussed in section 3.2, a large number of specialized software tools have been developed for examining biological networks. The most widely used tools are those that allow complete flexibility in layout and specification. This project will build on that philosophy of flexibility, and at the same time, incorporate an interactive interface for manipulating the graphs and models.

2.0 Design of Sonnet

The word Sonnet derives from an acronym, a **S**imple **O**ntology for **NET**works (SONNET), which is the name of the schema used to represent network like data. As the project grew beyond just data representation, the name Sonnet has come to encompass the entire software package. See Section 6.1 for an overview of the terminology and conventions used in the rest of the thesis.

After examination of use cases, the necessary functionality can be broken into the three main types. Foremost is data representation. In the case of Sonnet, data is assumed to be text based and will be modeled as a network of nodes and edges. In addition to representing the data, a framework will be created that allows data to be manipulated, loaded, saved, etc. After data management, the next step is the creation of views. In the Sonnet framework, a view is a selection of nodes and edges, and the visual characteristics of those nodes and edges. In other words, a view is a specification of which nodes and edges should appear and how those nodes and edges should appear, i.e. their shape, size, color, etc. Again, besides simply representing views, the framework will allow a view to be created, edited, saved, etc. Finally, once a view has been created, the view will be rendered into a network diagram. Sonnet will use third party applications (layout programs) to crcate the diagrams. A robust and extensible interface will be created so that the integration of new layout programs will only require a minimal amount of additional software. If the layout program supports interaction (selection of edges and nodes, etc.), the interactions can be monitored by Sonnet through the interface, allowing Sonnet to react to the users' actions. Moreover, Sonnet will work on seamlessly integrating these three core functions. Users should be able to easily maneuver from the data, to the view, to the layout, and vice versa.

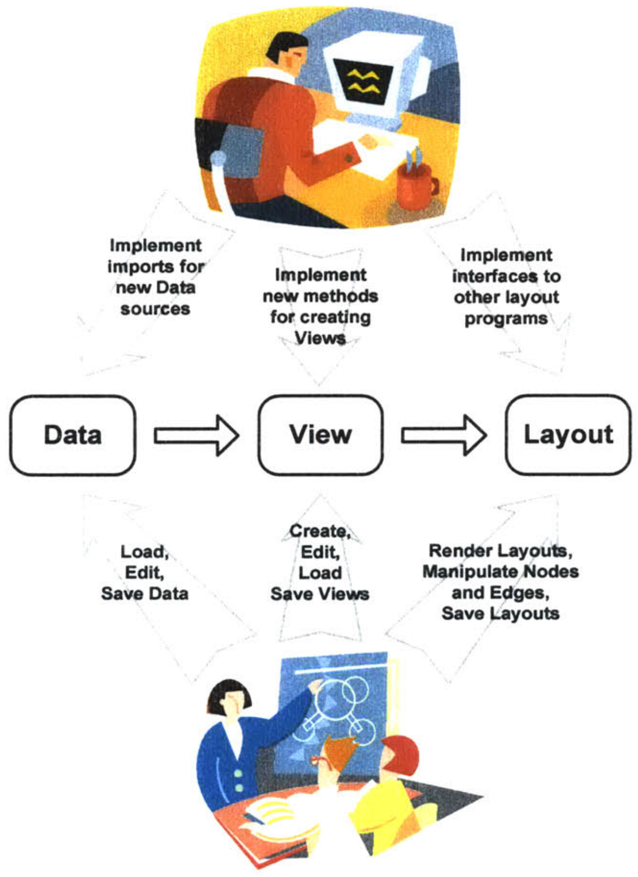

Implement imports for new Data sources
Implement new methods for creating Views
Implement interfaces to other layout programs
Data
View
Layout
Load, Edit, Save Data
Create, Edit, Load Save Views
Render Layouts, Manipulate Nodes and Edges, Save Layouts

Figure 7. Summary of use cases for Sonnet
Sonnet supports casual users who wish to use the existing software to create and interact with network diagrams. Sonnet also provides support, in the form of abstractions and extensible software, for users who wish to extend the capabilities of Sonnet.

Sonnet is designed to support two main kinds of end users. One type of user simply uses Sonnet to create network graphs, or to interact with network-like data. This user will use the provided software as-is to complete his tasks. However, other users will wish to augment the capabilities of Sonnet with new ways to create views, interfaces to new layout programs, etc. Sonnet will support this user by providing a simple interface to core functions and abstracting away common elements of code. A summary of how Sonnet is designed to be used is shown in Figure 7.

2.1 SONNET

Simple **O**ntology for **N**etworks (SONNET) is a framework for representing data as networks. This framework captures only the necessary dependencies required to represent a consistent network, giving SONNET the capability of representing a variety of networks. SONNET has three kinds of objects—Node, Edge, Attribute. Each object is strongly typed[14].

A type is a taxonomical label and represented by a pseudo-object, Type. For purposes of explanation, all Type[15] objects will simply be text labels—Person, Location, School, etc.—where equality of the text implies equality of the Type.

An Attribute is the basic object of SONNET and represents a mapping between a Type and a value. In SONNET, there is no specific object for representing a value, and in general, a value can be any object. For purposes of explanation, values[16] will be

[14] Lower case edge, node, attribute and type refer to the common conceptual meaning of the word, whereas upper case Edge, Node, Attribute and Type refer to objects in the SONNET definition. Edge, Node, Attribute and Type objects model their counterpart concepts with restrictions.

[15] A Type will be represented by a Capitalized word, such as Person, Location, School, etc. When a Type is more than one word long, the Type will be enclosed in double quotes, such as "MIPS Category," "Credit Card," etc.

[16] Values will be denoted in *italics*.

limited to objects that can be represented by text. The Type and value of an Attribute have an "is a" relationship. For example, the Attribute Location::*Boston*[17] can be interpreted colloquially as *Boston* is a Location, or literally as *Boston* is an object of the Type Location. An Attribute is considered equivalent to another Attribute if both their Types and values are equivalent. For instance, Location::*Boston* is not equivalent to Location::*New York*, or to City::*Boston.* In addition, Attributes can restrict the kind of values that can be associated with a Type. For example, the Type Birthday might only be allowed to map to valid dates such as *12 JUNE 2004*, but not to other values such as *This is not a valid date value.*

A Node is a special version of an Attribute. The only difference is that a Node is mapped to other Attributes, whereas, an Attribute is not mapped to other Attributes. For instance, if Person::*Dacheng Zhao* is represented as a Node, then Person::*Dacheng Zhao* could be mapped to the Attributes Location::*Boston*, School::*MIT.* The interpretation of the mapping between a Node and is Attributes is ill-defined and depends entirely on the interpretation of the Attributes. In our example, the Person::*Dacheng Zhao* is located at Location::*Boston* and attends School::*MIT.* More importantly, there is only one Node Person::*Dacheng Zhao* in a SONNET framework. In other words, two references to Person*::Dacheng Zhao* would refer to the same Node. Moreover, those two references would have the same mapping to Attributes, namely, Location::*Boston* and School::*MIT.* However, the Attribute Person::*Dacheng Zhao* is not equivalent to the Node Person::*Dacheng Zhao*.

[17] An Attribute with Type "Person" and value "Dacheng Zhao" will be denoted Person::*Dacheng Zhao*.

An Edge is a special version of a Node. Since a Node can only map to Attributes, an Edge is necessary to represent mappings between two Nodes. In addition to the requirements of a Node, an Edge is required to have two Attributes—one of Type Directed, one of Type Interpretation—as well as two Nodes—source, and target. The Attribute of Type Directed can only have two values—*true* or *false*. The values for the Attribute of Type Interpretation should explain to a reader how to interpret the Edge. Since an Edge is also a Node (hence, also an Attribute), an Edge can be represented by a Type and a value; however, the Type and value of an Edge are completely determined by the four parameters—Attribute objects with Types Directed and Interpretation, which will be referred to as directedness and interpretation, and two Nodes which will be referred to as source and target. Hence, if our system had another Node, Person::*Georg Gerber*, and Person::*Georg Gerber* is a friend of Person::*Dacheng Zhao*, then we could create the Edge "Person is a friend of Person, not directed"::*Georg Gerber is a friend of Dacheng Zhao, not directed*[18] which has the source Person::*Georg Gerber*, the target Person::*Dacheng Zhao*, the directedness Directed::*false* and the interpretation Interpretation::*is a friend*. When an Edge is not directed (Directed::*false*), the source and target Nodes can be switched. For example, the aforementioned Edge could equally be represented as "Person is a friend of Person, not directed"::*Dacheng Zhao is a friend of*

[18] In our example, the method for creating a Type from the source Node "A Node Type"::*Source Node*, target Node "Another Node Type"::*Target Node*, directed Attribute Directed::*True*, and interpretation Attribute Interpretation::*connects to* is: "A Node Type connects to Another Node Type, is Directed". The method for creating the value from the four parameters is: "Source Node connects to Target Node, is Directed"

Georg Gerber, not directed[19]. However, the aforementioned Edge is not equivalent to the Edge "Person works in the same lab as Person, not directed"::*Dacheng Zhao works in the same lab as Georg Gerber, not directed*[20] (edges differ in terms of interpretation, Interpretation::*is a friend of* vs. Interpretation::*works in same lab*). Two directed Edges are equivalent if they have the same target, source, directedness, and interpretation. Two undirected Edges are equivalent if they have the same directedness, interpretation, and the same two Nodes (Nodes in the two Edges do not have to be assigned to the same variable). An undirected Edge cannot be equivalent to a directed Edge.

Outside of discussing the SONNET specification, a reference to an "edge" is a reference to an Edge object and all Attribute objects associated with that edge. Likewise, a reference to a "node" is a reference to a Node object and all Attribute objects associated with that node.

2.2 *View*

Although a view is simple conceptually, it can be complex to create in practice. By definition, a view is a mapping between nodes and edges[21], and visual characteristics. The most straightforward method would be to specify an individual set of visual characteristics for each node and each edge. The components of the set of visual

[19] which has the source Person::*Dacheng Zhao*, target Person::*Georg Gerber*, the directedness Directed::*false* and the interpretation Interpretation::*is a friend*

[20] which has the source Person::*Dacheng Zhao*, target Person::*Georg Gerber*, the directedness Directed::*false* and the interpretation Interpretation::*works in the same lab as*

[21] Nodes and edges include the attributes that they are associate with

characteristics may not necessarily be unique (i.e. two different nodes could be the same color, etc.); however, the data for each node and edge would be stored and edited separately. Although state information for visual characteristics may be stored in files in such a way, this one-to-one approach is not tractable for large numbers of nodes and edges. Even with current graphs that have hundreds of nodes and thousands of edges, manipulating the visual characteristics of edges and nodes one by one would be impracticable.

Instead, we will present a framework for designing computational filters and rule-based approaches to assigning visual characteristics. We imagine a scenario where diverse rules and filters have been implemented. Some filters may use graph theoretic techniques to assign visual characteristics, others might be a series of rules based on the data attributes of a node or edge, etc. Furthermore, we envision that rules and filters can be layered in systematic ways.

2.3 Interface to Layout Programs

Layout rendering programs have a wide variety of features. While some programs focus on allowing users to manually manipulate nodes and edges, others focus on using algorithms to computationally determine the best layout. The wide range of features already offered by graph layout programs is one of the reasons that Sonnet focuses on the data and visual characteristic management aspects of creating a graph. Only a few aspects of layout programs are needed to create an interface with Sonnet. First, one must determine the legal visual characteristics of nodes and edges in the layout program (i.e. which colors, shapes, etc., nodes and edges are allowed to have). Then, one must have a programmatic method for instructing the program to draw nodes and edges with the appropriate characteristics. Although it would be ideal if the program had an exposed interface to commands such as "draw edge" or "draw nodes," more roundabout methods, such as exploiting file specifications, can also work. Finally, one hopes to capture in a timely fashion messages about how nodes and edges are being manually moved, so that Sonnet can respond with appropriate actions or graphics.

3.0 Implementation of Sonnet

Sonnet is completely implemented in the Java programming language (specifically, Sonnet makes use of features that are only available after version 1.4 of Java[22]) [39]. Hence, Sonnet can be used on any platform for which a Java Runtime Environment (i.e. Java Virtual Machine) exists[23].

Furthermore, Sonnet adheres to the object-oriented paradigm. Sonnet makes extensive use of interfaces and static classes to decouple implementation from specification. An overview of Sonnet can be seen in Figure 8.

[22] The latest released version of Java is 1.4.2. Its specifications can be seen here:

http://java.sun.com/j2se/1.4.2/docs/api/

[23] http://java.sun.com/j2se/1.4.2/download.html

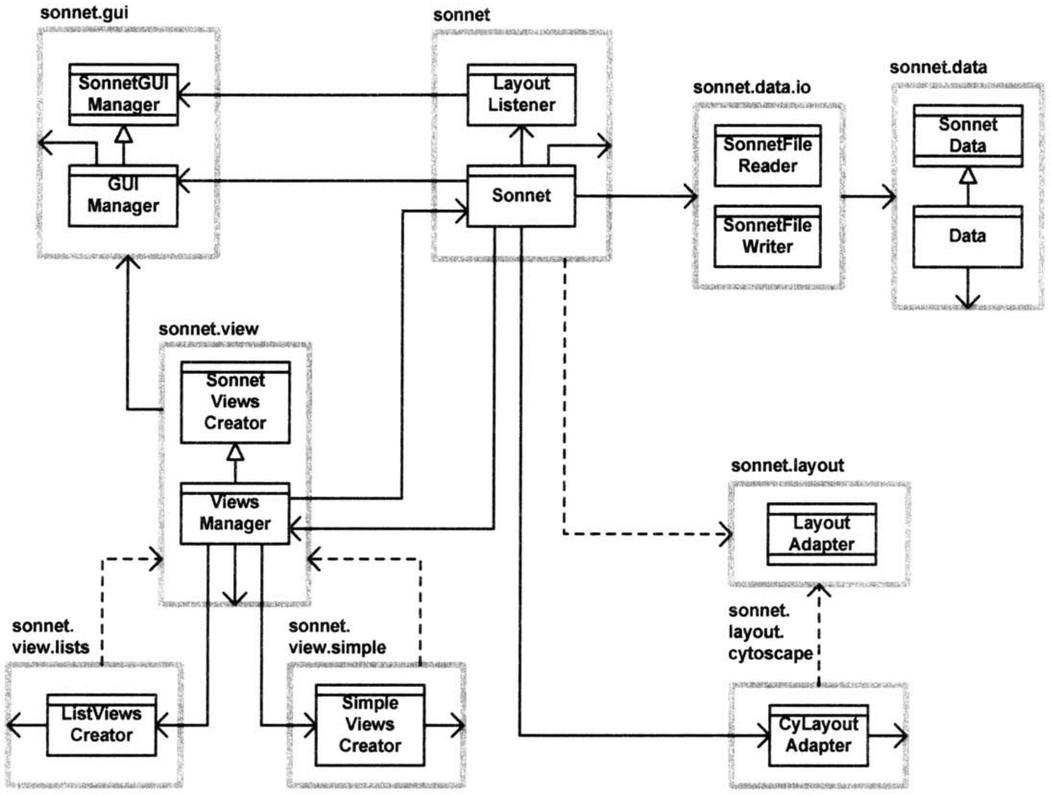


Figure 8. Overview of the implementation
This overview of Sonnet focuses on key classes and dependencies. See appendix Section 6.2 for a detailed explanation of the various symbols. In short, dashed lines represent use of interfaces, including implementation if the line terminates at a package. Solid lines represent use of classes. A solid line between packages implies use of classes and interfaces; however, a dashed line between packages implies no classes in the target package are used by the source package. Moreover, a solid line between one class and a class in another package implies that the source class does not use any other class in target package (that would be represented by a solid line between a class and a package). For brevity, dashed lines have been omitted when the target package is `sonnet.data` since every other package uses that package. Notable design features include the relative dearth of dependencies, especially the complete lack of dependencies of the `sonnet.data` package (no outward bound edges).

A central theme to this implementation is the reduction of dependencies through the use of a few singleton classes [40]. Node, edge, and attribute data is managed and by an instance of the class `Data`, an implementation of the interface `SonnetData`[24] (see

[24] The class `Data` is not strictly a singleton since it does not enforce the singleton policy of only one instantiation per Java virtual machine. However, there is only one copy exists per Sonnet environment

Section 3.1). Views are managed through the class ViewsManager, an implementation of the class SonnetViewsManager (see Section 3.2). Interactions with a layout rendering program are managed through an instance of the class LayoutAdapter (see Section 3.3; currently the only layout program for which an adapter has been written is Cytoscape. The implementing class is CyLayoutAdapter). Graphical user interfaces (GUIs) are managed by the class GUIManager (see Section 3.4). GUIManager updates GUIs when data changes, or when the program receives notice that the user has selected a node or edge in the layout program. Finally, all of these classes are managed by the class Sonnet. Sonnet, and Sonnet alone, instantiates the classes ViewsManager, CyLayoutManger, and GUIManager. Sonnet also possesses an instantiation of SonnetData (instantiated by SonnetFileReader). When one of these classes needs to access other functionality (e.g. the ViewsManager needs to tell the LayoutAdapter to render a View), that class simply calls Sonnet, which returns the interface of the requested functionality, hence isolating the requesting classes from the concrete implementations. Similarly, when any class needs to access the data represented by the system, they call on Sonnet which returns an instantiation of an implementation of SonnetData[25].

Another hallmark of Sonnet is ease of extension, specifically regarding new interfaces to layout programs, and new methods of creating views. In each case, the

---

since the class Sonnet only maintains one copy, which it then shares with all classes that need access to Data.

[25] Currently, implementing classes are often located in the same directory as the interfaces. However, this may change, since separating classes from interfaces would only enhance the modularity and abstraction

implementing classes need only extend one package (two if you include the ubiquitous `sonnet.data`) and implement two or three classes. Again, the fact that all interactions pass through only a handful of classes, and the relative lack of dependencies, greatly improves a user's ability to understand how the system works, and how to extend it.

3.1 Nodes, Edges and Attributes

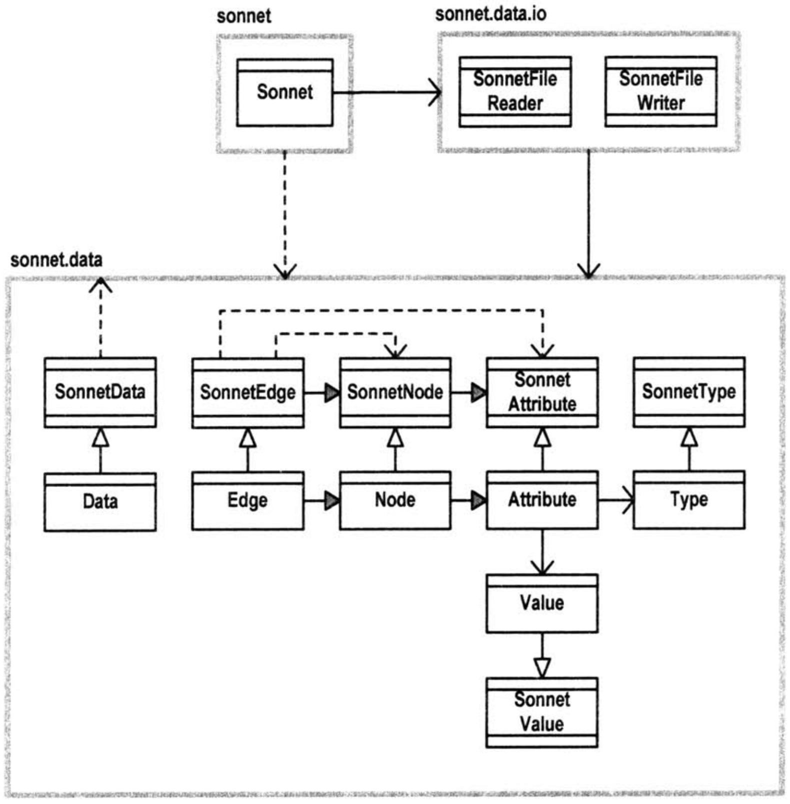


Figure 9. Overview of the `sonnet.data` and `sonnet.data.io` packages
For brevity, some edges have been omitted from this diagram. Notably, every single class and interface in these packages (except `Value` and `SonnetValue`) uses the interface `SonnetType`. In addition, use or implementation arrows between interfaces implies use arrows between implementing classes since only one class implements each interface. These packages are unique in the Sonnet framework because of their complete lack of dependency on other Sonnet packages.

A robust version of the SONNET specification is implemented in the package `sonnet.data`. An overview outlining the most salient relationships can be seen in

Figure 9. This package regularly trades memory for computational efficiency by hashing basic data-types as well common queries (such as which edges a node is connected to). In addition, the package makes extensive use of static factory methods to facilitate extensibility [40]. More importantly, this package, along with the auxiliary `sonnet.data.io` package for saving and loading files, has no dependencies on any other package in Sonnet. Hence, this package can be readily imported into any Java application that needs to model a network comprised of nodes and edges, where nodes and edges represent other data.

Nodes, edges and attributes in this framework are strongly typed. In this implementation, every node, edge and attribute contains a non mutable object instance of a `SonnetType` interface, which is implemented by the class `Type`. Conceptually a type is simply a text label; hence the class `Type` is a wrapper for the class `String`. In the Java framework, instances of the class `String` are immutable and are hashed, meaning if one instantiates two instances of a class `String` containing the text "hello world," both instances point to the same object in memory. In anticipation of future definitions of a type that might not be simple text labels, the class `Type` also hashes all instances of itself. The class `Type` is by far the most instantiated class in the framework since every node, edge and attribute has an instance. On larger networks, hashing of the class `Type` could result in significant savings in time and memory.

Values are generally text based; however, depending on the text, the actual interpretation is very different. For example, the text "123.0" should be interpreted as a decimal number whereas the text "www.yahoo.com" should be interpreted as a URL reference. To accommodate this difference at the base level, each value is passed to the class `Value` which then interprets the text appropriately and creates an instance of class `Double`, `Integer`, `URL`, `Boolean`, or `String`. `Value` returns an object of the generic class `Object`; however, this object can then be cast into one of the aforementioned classes. It is debatable whether the values should be represented by a unique class to help restrict the possible kinds of acceptable values. Since values are currently stored as the generic class `Object` in `Type`, changing all values to instances of a specific class would be a trivial task.

An attribute is a binding between a single type and a single value. In this implementation, an attribute is an instance of the `SonnetAttribute` interface which is implemented by the class `Attribute`.

A node, like an attribute, is also a binding between a type and a value. However, a node is allowed to be associated with, or mapped to, a set of attributes. A node is an instantiation of the `SonnetNode` interface which is implemented by the class `Node`. The class Node is an extension of the class `Attribute`[26]. There is little difference between `Node` and `Attribute` except that `Node` has an additional `getName()` method since, grammatically, it is more reasonable to refer to the name of a node rather than the value of a node . In addition, the `equals(Object obj)` of the class `Node` has been rewritten to override the method inherited from `Attribute`. For two `Node` instances to be equal, `obj` must be able to be cast as a `SonnetNode` in addition to

---

[26] `SonnetNode` is also an extension of `SonnetAttribute`. Extension of `SonnetAttribute` by `SonnetNode` is somewhat unnecessary, however, it reinforces the conceptual logic that a Node is simply a special type of Attribute.

having an equivalent `SonnetType` and value. In other words the Node Person::*Dacheng Zhao* is not equivalent to the Attribute Person::*Dacheng Zhao*.

An edge represents a connection between two nodes. Hence, at minimum an edge needs to encapsulate two nodes, the directedness of the edge, and the interpretation of the edge. In this implementation, an edge is an instantiation of the interface `SonnetEdge` which is implemented by the class `Edge`. A `SonnetEdge` manages two `SonnetNode` objects, a variable determining whether the edge is directed, and a `String` object encapsulating a text interpretation of the meaning of the edge. `SonnetEdge` extends from the `SonnetNode`, and `Edge` extends from `Node`. In this way, an edge is also a node and an attribute. However, the type and value of an `Edge` is automatically generated from its two nodes, directedness and interpretation. At instantiation, an `Edge` concatenates the types of the two nodes, directedness and interpretation to create its `SonnetType`. For instance, an Edge that is directed, signifies "binds to" and contains the edges "Transcription Factor"::*GAL80* and ORF:*YNL159C* would create the type "Transcription Factor >binds to> ORF". If this Edge were not directed, then the Type would be "Transcription Factor binds to ORF." In addition, an Edge will concatenate the names of the two nodes along with the directedness and interpretation to create its name (i.e. value for an attribute). In this case, the name of this edge would be "GAL80 >binds to> YNL159C." Again, if the edge were not directed, the name would be "GAL80 binds to YNL159C". By extending from `Node`, and hence `Attribute`, the `Edge` class is able to share some machinery, chiefly that which is used for checking equality. The `Edge` class overrides the `equals(Object obj)` from `Node` so that `Edge` can check that `obj` can be cast as an `SonnetEdge`, but then, it uses the inherited equals(Object obj) to perform the rest of the equality check. In addition, by extending Edge from Node, and Node from Attribute, we reinforce the concept that a Node is a special kind of Attribute, and that an Edge is a special kind of Node. This will allow us to natively exploit the similarities when creating user interfaces and other software.

Edges, nodes and attributes are all tied together by instantiations of the SonnetData interface, which is implemented by the class Data. The class Data performs all the expected functions, such as allowing one to add SonnetNode objects and SonnetEdge objects, mapping SonnetNode objects and SonnetEdge objects to SonnetAttribute objects, and allowing the removal of SonnetNode and SonnetEdge objects. While these tasks are being performed, the environment, which contains these edges, nodes and attributes, is maintained consistently. For instance, if a node belongs to an edge, and the node is removed from the environment, then, the edge is also removed since an edge must have two nodes to exist. In addition, the class Data makes extensive use of the classes HashMap[27] and HashSet[28] so that many common queries are cached. For example, the set of all edges that a node belongs to can be retrieved in constant time (getNodeEdges(SonnetNode node)). Likewise, this holds for retrieval of all the attributes associated with a node, or all the attributes associated with an edge. Furthermore, Data also caches all the types of nodes, edges

[27] http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

[28] http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashSet.html

and attributes in the system. Hence, one can find in constant time all nodes that are of type "ORF" or all edges of type "ORF >belongs to> Module," etc.

This implementation of SONNET makes extension simple and elegant. One common feature might be to require certain types of nodes and edges to have instances of certain attributes[29]. This capability could be added by simply extending the class `Data` and then overriding any methods that add nodes and edges[30]. In the overridden methods, the extended class would check to see if the nodes and edges possessed the required attributes. If the nodes and edges did not possess the required attributes, an error would be passed back to the user. If the nodes and edges did possess the required attributes, then the extending method would simply pass its variables to the extended method in `Data`. Similarly, if a user wished to view certain attributes differently (see Section 3.4), he could extend the classes `EdgeDataPane` and `NodeDataPane`.

3.1.1 SONNET File Type

State information for an object that implements `SonnetData` can be saved in a simple verbose format using the classes `SonnetFileReader` and `SonnetFileWriter` found in the `sonnet.data.io` package. The specification for

[29] For example, biologists often need to verify the source of data. Sources could be represented as attributes Experiment, Laboratory, Reference, etc. Then, all edges between transcription factors and ORFs (representing a binding event) could be required to have the attributes Experiment, Laboratory, and Reference (e.g., Experiment::*ChIP*, Laboratory::*Young Lab*, Reference::*Lee et al, Science 2002*).

[30] `addEdge(SonnetEdge edge, Set attributes)`, `addNode(SonnetNode node, Set attributes)`, etc.

the Sonnet File Type (SFT) consists of a rule for saving attributes, a rule for saving a list of attributes, a rule for saving nodes and a rule for saving edges.

SFT is a prototype solution for specifying the state data of `SonnetData` as text. The main advantage of SFT is that it requires very little code to parse and write. Unlike more advanced specifications such as RDF or XML[31], or graph specific specifications such as DOT[32] and GML[33], SFT does not require any advanced libraries to parse the text. Instead, simple string tokenizing tools available in most programming languages (`java.lang.StringTokenizer` in Java) or regular expression tools are more than adequate. Data is currently housed in a variety of formats, from spreadsheets and Matlab workspaces, to databases and online repositories. Unable to support all formats, the next best solution was to create the simplest possible format so that programmatic users of Sonnet need only spend a minimum amount of time on importing data. However, as data grows in size and complexity the shortcomings of SFT (i.e. verbosity, no inherent error checking) may become more pronounced. Future work will involve reexamining these issues and modifying SFT, or abandoning SFT altogether in favor of more advanced specifications.

SFT uses different delimiter tokens to separate different values. Attributes are saved as:

```
Type-As-Text Attribute-Delimiter-Character Value-As-Text
```

[31] http://www.w3.org/TR/rdf-syntax-grammar/

[32] http://www.research.att.com/~erg/graphviz/info/lang.html

[33] http://infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html

The variable representing the attribute-delimiter-character is typically named `aDelim`. An `aDelim` can only be one ASCII character with a few restrictions (e.g. it cannot be the space or carriage return character). However, it is imperative that the `aDelim` character is not used as a character in any text representation of a type or of a value and that the `aDelim` character is not used for any other delimiter in the Sonnet file type. Currently `aDelim` is set to "\" by default and a typical attribute in the Sonnet file type looks like this:

```
ORF \ YER074W
```

Lists of attributes are separated by another token, typically represented by the `lDelim` variable. Currently `lDelim` is set to "\t" (i.e. tab) and a list of attributes looks like this:

```
Binding Condition \ YPD   interp \ binds to   dir \ true
```

Since a node can be thought of as an attribute that is allowed to map to attributes, saving a node amounts to outputting an attribute and delimited list of attributes. The character that separates the attribute and list of attributes is named `vDelim` (nodes versus attributes) and is set to "|" by default. A node typically looks like this:

```
ORF \ YPR132W | ORF Function \ 40S small subunit ribosomal protein S23.e
```

An edge could conceivably be represented in two ways. One way would be to save the type and value (i.e., name) of an edge, hence treating an edge like a node. The other way would be to save the two nodes of an edge as well saving the directness and interpretation of an edge. This second method has been chosen because the first method relies on the implementation of `SonnetEdge` since the type and value of an edge is generated by the implementing class. Hence, an edge is saved as two nodes, separated by `lDelim`. In the attributes section of the text, an edge must have the two special attributes "interp" (or "interpretation") and "directed" (or "dir"). If an edge is directed, then the first node listed must be the source and hence the second node will be the target. A typical saved edge looks like this (all on one line):

```
Transcription Factor \ FHL1	Module \ Module #9 | Binding Condition \ YPD	interp
\ binds to	dir \ true
```

This line represents a directed edge that translates to: transcription factor FHL1 binds to a module, Module #9 w/11 genes. This edge is associated with a binding condition, YPD. It lies between a transcription factor, FHL1, and a module, Module #9. This edge is directed and the interpretation is "binds." Note that the two special attributes can be located anywhere in the attributes list. Further, a node in the edge (source or target) does not have to be specified before the edge is specified in the text file. In Sonnet, when adding an edge, the existences of the source and target nodes are first checked. If they do not exist, they are instantiated at that time.

The main benefit of this file type is a flexibility that matches the flexibility of the Sonnet data package. However, the current disadvantage is the need for three unique single character delimiters to serve as `aDelim`, `lDelim`, and `vDelim`. The current three delimiters ("\", "\t", "|") were chosen after analysis of a several example of networks. An especially troublesome feature of the data in these networks is the use of URLs that can contain many non-alphanumeric characters. For example, the following URL specifying a search for a specific text string in the Entrez database disqualifies many characters that might otherwise be used as more legible delimiters (e.g. ":", "=", "?", etc.):

```
http://www.ncbi.nlm.nih.gov/gquery/gquery.fcgi?term=YMR194W
```

Future versions of `SonnetFileReader` and `SonnetFileWriter` will migrate away from using the simple text tokenizing class `StringTokenizer`, and move towards using more powerful regular expression tools found in the `java.util.regex` package (see Section 6.3, "Using regex in `SonnetFileReader` and `SonnetFileWriter`").

3.2 Views

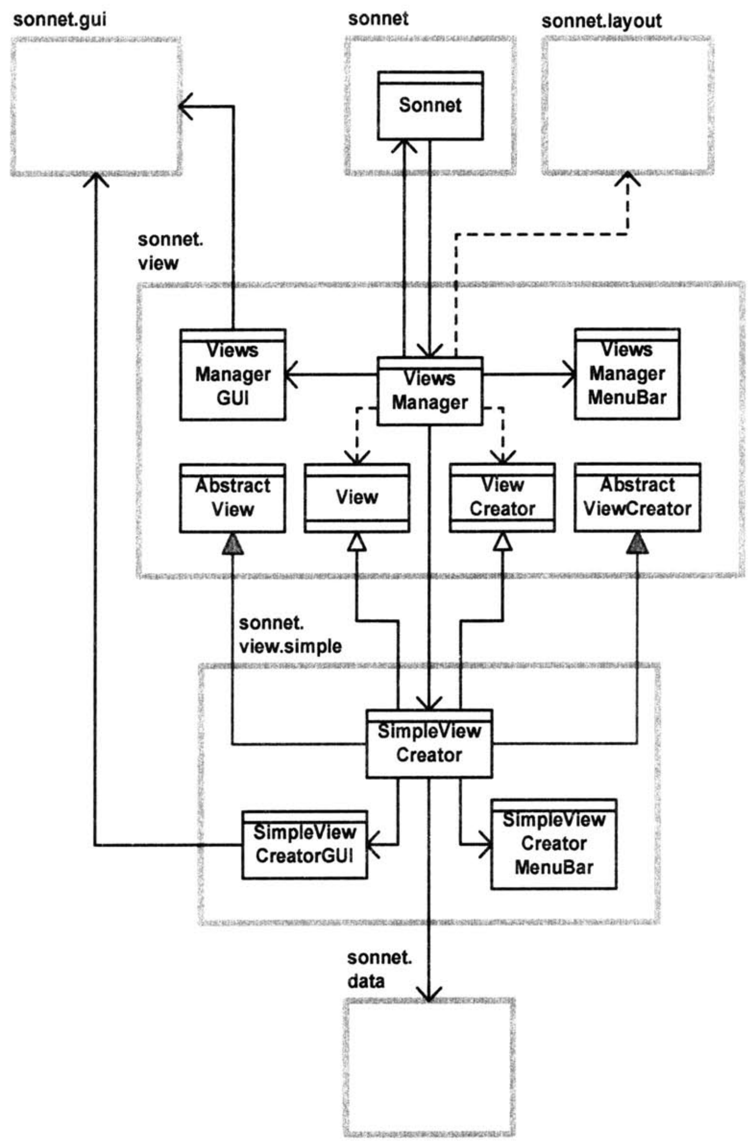


Figure 10. Overview of `sonnet.view` and `sonnet.view.simple` package
This diagram shows the key interactions of the `sonnet.view` package. The `sonnet.view.simple` package diagram demonstrates the key interactions when `sonnet.view` is extended. `SimpleViewCreator`, besides being the most basic way to create views, also demonstrates how one class can simultaneously implement both `View` and `ViewCreator` (a tactic that can easily solve the problem of sharing state information between a `View` and the `ViewCreator` that created it).

Sonnet's method of managing views is located in the package `sonnet.view`. An overview of this package can be seen in Figure 10. This package implements these functions: specification of a view, storage of views, management of views (through a GUI), specification of a view creator, and management of view creators. In the Sonnet framework, a view is any class that implements the `View` interface. The `View` interface only requires three methods:

```
    getNodeViz(SonnetNode node)
    getEdgeViz(SonnetEdge edge)
    getViewsCreator()
```

For every `SonnetNode` and `SonnetEdge` stored in the current `SonnetData`, an implementation of `View` is required to return a valid `NodeViz` or `EdgeViz`. If a `SonnetNode` or `SonnetEdge` does not exist in the `View`, then the `View` should return null. In addition, a `View` usually has access to the `ViewCreator` object that created it, and it will pass the `ViewCreator` to `ViewsManager` when requested. However, if the `View` does not have access to `ViewCreator`, it will be assumed to be a `SimpleView`[34] (which can accommodate any `View`) and the `SimpleViewCreator` will be used instead.

Classes that implement `ViewCreator` must implement methods for saving, loading and editing a `View`. Depending on how the `ViewCreator` intends the user to create and edit views, the state data can vary wildly from simple constants (perhaps from rules) to elaborate databases. `ViewCreator` does not impose any standard for saving the data. In addition, a `ViewCreator` usually implements a GUI to allow the

[34] `SimpleView` does not exist and the implementation of `View` in `sonnet.view.simple` is actually the class `SimpleViewCreator`. Because `SimpleCiewCreator` implements both `View` and `ViewCreator`, `SimpleCiewCreator` will sometimes be referred to as `SimpleView` when we are addressing the `View` interface aspects of the implementation (i.e. `getNodeViz(SonnetNode node)` and `getEdgeViz(SonnetEdge edge)`). See Section 3.2.1 for more details.

modification of a `View` interactively. `ViewsManager` will take care of setting up a `JFrame` to host the GUI of a `ViewCreator`, and hence, will ask the `ViewCreator` for a `JPane` and `JMenuBar`.

Classes that implement `View` and `ViewCreator` can extend the abstract classes `AbstractView` and `AbstractViewCreator`. These classes contain some common methods (such as `equals(Object obj)`) that should be useful for any classes implementing `View` and `ViewCreator`.

Managing `View` and `ViewCreator` objects, along with a GUI that allows users to interact with these objects is the responsibility of the class `ViewsManager`, which implements `SonnetViewsManager`. The `ViewsManager` class has a few primary responsibilities including:

- maintaining a list of `View` objects that the user can select in a GUI,
- instructing the `LayoutAdapter` to render a view if the user wants to render a view,
- setting up the `ViewsCreator` for a `View` if the user decides to edit a view or create a new view, and
- converting a view to `SimpleView` if the `ViewsCreator` for a `View` cannot be instantiated.

In short, the `ViewsManger` will attempt to allow a user to use the `ViewsCreator` that created the `View` to also edit, load, save, etc. the `View`. However, if that is not possible, `ViewsManager` will default to using the `SimpleViewCreator` implementation of `ViewCreator`.

3.2.1 Simple Views

The `sonnet.view.simple` package espouses the most basic approach to creating a view. In short, the user specifies a unique view for each edge and node that is to be rendered. This process is laborious and this approach was only created for two reasons. First, it serves as a concise example for how the interfaces in `sonnet.view` should be and can be extended. In addition, the approach of `sonnet.view.simple` is the only approach that every implementation of `View` is guaranteed to be compatible with. When a `View` does not return a valid `ViewCreator`[35], `ViewsManager` will call upon `SimpleViewCreator` to edit, save, load, etc. the `View`. Furthermore, in this case the `SimpleView` object is also the `SimpleViewCreator` object. This double implementation by `SimpleViewCreator` is quite natural, as it allows state data regarding the `View` to be innately shared between the `View` and the `ViewCreator`.

In addition, `sonnet.view.simple` also serves as a demonstration of the flexibility of the `sonnet.data` package. Without any modification or extension to the `sonnet.data`, `sonnet.view.simple` uses the `sonnet.data.io` package to save and load `Data` objects. When saving a View, `SimpleViewCreator` simply converts an `EdgeViz` or `NodeViz` to a set of `SonnetAttribute` objects, which then are added to `Data`. When all `EdgeViz` and `NodeViz` objects have been converted, `SimpleViewCreator` uses `SonnetFileWriter` to save `Data`.

[35] This is can happen for valid reasons. For one, the author of the view may not intend it to be editable through an implementation of the `ViewCreator` class.

Similarly, SimpleViewCreator uses SonnetFileReader to load the data and then reverses the process.

3.2.2 List Style Views

If the sonnet.view.simple is the zeroth order approach to creating views, then the sonnet.view.list approach is the first order approach to creating views and sonnet.view.list can be thought of as a logical extension of sonnet.view.simple. In this package, the creation of the View has three tiers. First, the user can specify EdgeViz or NodeViz simply based on individual SonnetEdge and SonnetNode objects. Then, the user can specify EdgeViz and NodeViz based on the SonnetType objects of the SonnetEdge and SonnetNode objects (e.g., all SonnetNode objects of type "ORF" should be small blue circle). Finally, a user can specify specific EdgeViz and NodeViz based on specific ranges or values for SonnetAttribute objects that SonnetEdge and SonnetNode objects map to (e.g. all SonnetNode objects with the attributes Module::*Module #47* should have green borders). In View mode, the process is the same. When returning an EdgeViz, the getEdgeViz(SonnetEdge edge) method checks to see if the edge maps to a unique EdgeViz. If not, it checks to see if the SonnetType of the edge matches a SonnetType that has a unique EdgeViz. If that fails, it finally compares the SonnetAttribute objects of the edge with sets of specified SonnetAttribute objects that map to an EdgeViz. The ordering of this logic can be user defined; however, it seems that the default ordering (individual node or edge, SonnetType, sets of SonnetAttribute) is the most powerful.

3.3 Layout Interface

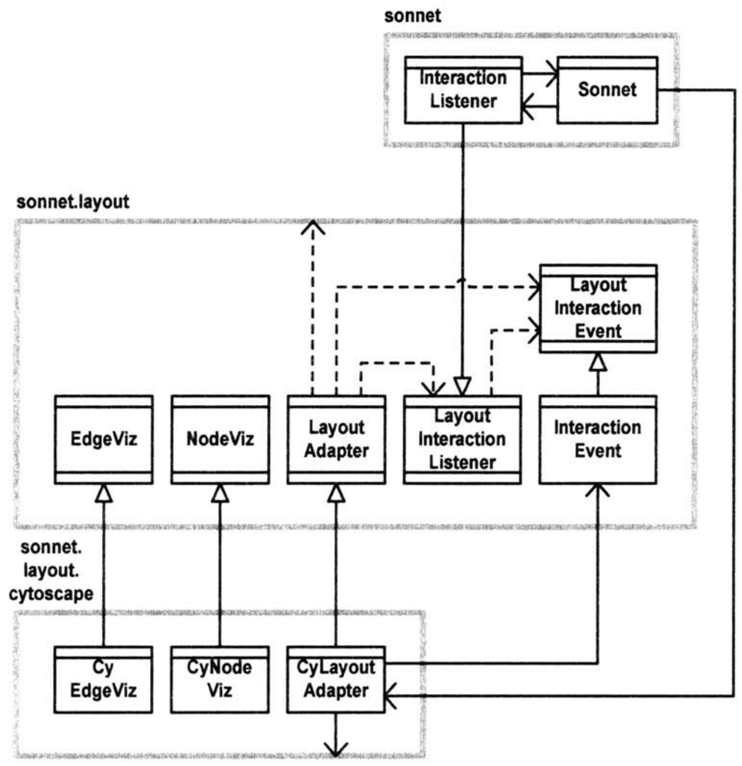


Figure 11. Overview of the `sonnet.layout` and the `sonnet.layout.cytoscape` package
This diagram demonstrates the `sonnet.layout` package and the salient aspects of the extension, `sonnet.layout.cytoscape`.

All the interfaces required to be implemented in order to extend Sonnet to a new layout program are located in the `sonnet.layout` package. An overview of the package can be seen in Figure 11.

`EdgeViz` and `NodeViz` specify the boundaries of what constitutes a visual characteristic for a node and edge. Currently, this includes: fill color, line color, border color (for nodes), shape (for nodes) height, width, target end shape (for edges), source end shape (for edges), and transparency. It is the task of the `LayoutAdapter` to translate these characteristics into a form the layout program can understand, or if a layout program does not support certain characteristics, `LayoutAdapter` should filter out the extraneous characteristics.

3.3.1 Cytoscape

Currently, the only layout program for which a Sonnet interface has been implemented is Cytoscape. Specifically, the interface is for Cytoscape 2.0 alpha release number 3. Cytoscape 2.0 is set to be released in the fall of 2004. The `sonnet.layout.cytoscape` package should only require minor changes (in the class `CyAbstractPlugin`) to work with the fully released version of Cytoscsape 2.0.

In addition to implemented classes for `EdgeViz`, `NodeViz` and `LayoutAdapter`, other classes, unique to the Cytoscape integration, exist in the `sonnet.layout.cytoscape` package. Cytoscape uses instances of `PInputEventListener` for handling node and edge interaction events. To interface with this class, `SimplePInputEventListener` implements `PInputEventListener` and raps events (`PInputEvent`) into Sonnet's `InteractionEvent`. Then, `SimplePInputEventListener` calls Sonnet's `InteractionListener`, which triggers an action with the new `InteractionEvent`. `CyPluginSonnet` extends `AbstractPlugin`, the default class for creating a plug-in for Cytoscape. `CyData` holds state data for both Cytoscape and Sonnet and maps between the node and edge objects in Cytoscape, and the node and edge objects in Sonnet.

3.4 GUI

The package `sonnet.gui` contains a variety of graphical user interface elements. In general, any GUI class that has been designed to be reused by other classes is located in the `sonnet.gui` package. GUI classes that are specific to one application are located in the package where they are used.

All GUI classes follow a simple pattern in Sonnet. Although not specified in an interface, all GUI classes have two methods:

```
setup(Object obj1, Object obj2, etc.)
update()
```

In addition, GUIs that are based on one state Object—such as `EdgeViz`, `NodeViz`, `Edge`, `Node`, etc.—have a third method, `Object grabData()`, where `Object` is `EdgeViz`, `NodeViz`, `Edge`, `Node`, etc.

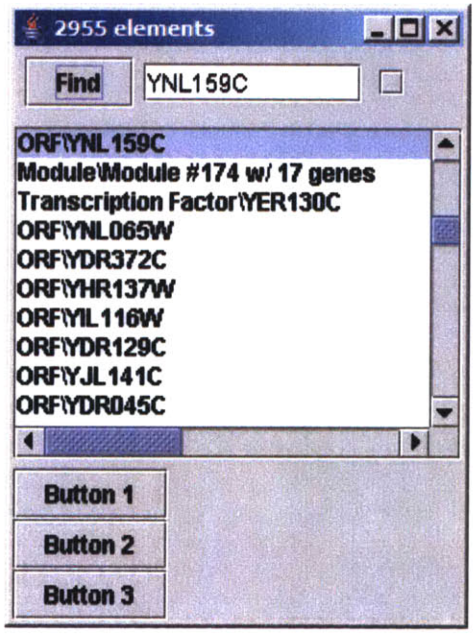
The method `setup()` contains logic that creates all they physical components of the GUI. This includes instantiating `JComponent` objects (`JPanel`, `JButton`, etc.) as well as setting up default behavior, such as behavior of scroll bars, behavior of lists (number of items that can be selected at once), etc. In addition, `setup()` will set the values of internal state variables with the variables passed into `setup()` (i.e., `Object obj1, Object obj2`, etc.), or with default values. After `setup()` has been called, accessing any state variable in the GUI should never return null. `setup()` should only be called if the user wishes to create the GUI, and in general, `setup()` is only called by the constructor.

The method `update()` is called to update GUI elements that contain state data once state data has changed. For example, if a GUI is based on an `Edge`, once the `Edge` has been changed externally (i.e., not by the GUI), `update()` should be called so that all fields and labels based on the `Edge` are updated. Even though the update method wastefully updates all fields, labels, lists, etc., the readability and reusability of the code is greatly enhanced. In our example, the GUI object might have a method `setEdge(SonnetEdge edge)`. We would expect that method to only have two lines of code:

```
    this.edge = edge;

    update();.
```

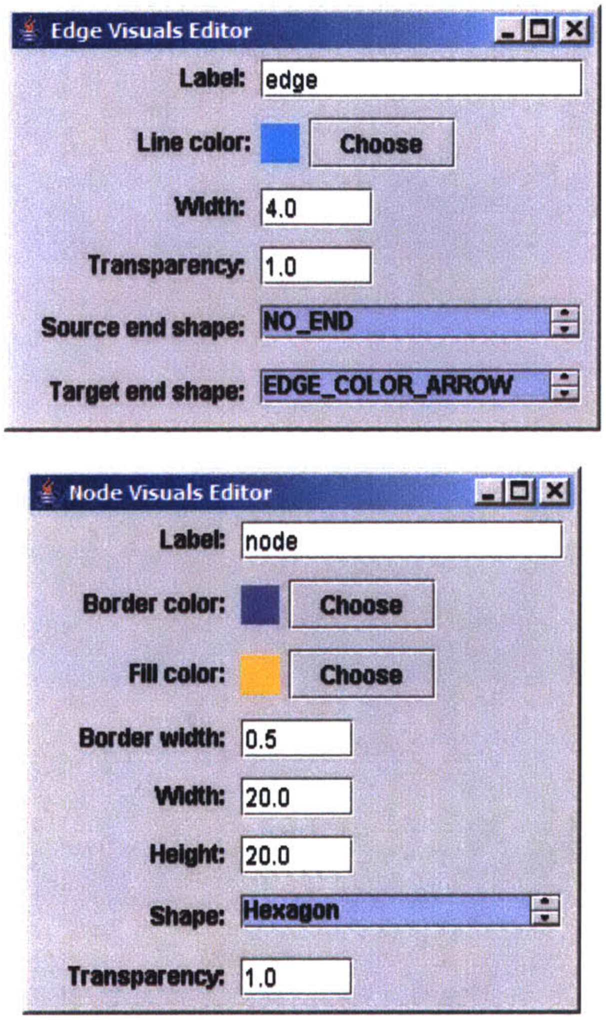
The method `grabData()` is called to return the state data encapsulated in an `Object`. For example, in a GUI that allows a user to edit data related to an `Edge`, `grabData()` should return an instantiation of `Edge` based on the fields of the GUI.



ORF\YNL159C
Module\Module #174 w/ 17 genes
Transcription Factor\YER130C
ORF\YNL065W
ORF\YDR372C
ORF\YHR137W
ORF\YIL116W
ORF\YDR129C
ORF\YJL141C
ORF\YDR045C

Figure 12. An example of the versatile `ListPane`
An example instance of a ListPane filled with 2955 elements. ListPane encapsulates many common functionalities needed when using a selectable list, including a search function (based on regular expression), and formatting and layout issues. Selection of the box directly following the field with `YNL159C` enables search text to be treated as a regular expression. When the box is unchecked, the search is equivalent to the regular expression `.*YNL159C.*`.

There are a few highly reusable classes in `sonnet.gui`, one of which is the class `ListPane` (see Figure 12). `ListPane` encapsulates all the logic for a selectable list. Moreover, it also supports adding buttons, menus, as well as listeners. It has a built in search function based on regular expression, and it has logic for automatically shifting the appearance of the list to the selected item. Many of the GUIs in Sonnet are based upon `ListPane`.


Edge Visuals Editor

Label: edge

Line color: 🟦 Choose

Width: 4.0

Transparency: 1.0

Source end shape: NO_END

Target end shape: EDGE_COLOR_ARROW

Node Visuals Editor

Label: node

Border color: 🟦 Choose

Fill color: 🟨 Choose

Border width: 0.5

Width: 20.0

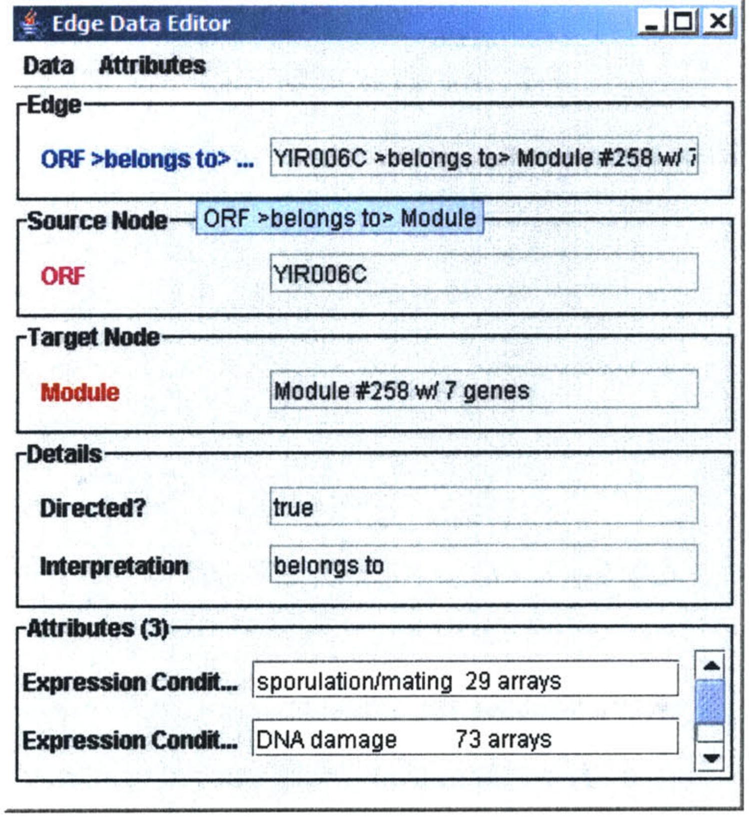
Height: 20.0

Shape: Hexagon

Transparency: 1.0

Figure 13. GUIs for editing Edge Visuals and Node Visuals
These two classes, `EdgeVizPane` and `NodeVizPane`, allow an end user to create an `EdgeViz` or `NodeViz`. These classes include such conveniences as a color chooser, and a pull down list for shapes. Furthermore, they also handle error handling and will warn the user if they try to input invalid data, such as `This is not a number` for Transparency.

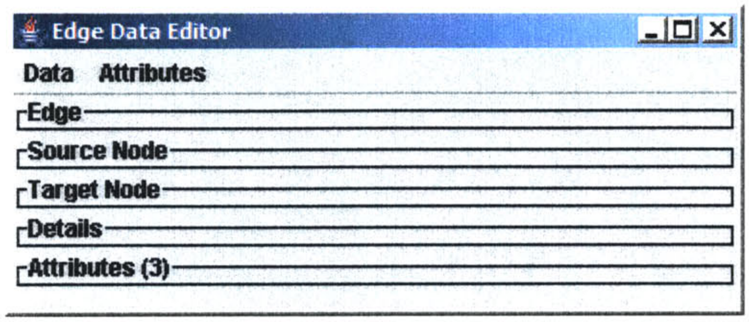
Two other GUIs that are designed for extension and embedding in other GUIs are the Edge and Node visuals editors—`EdgeVizPane` and `NodeVizPane` (see Figure 13). As their name suggests, theses classes encapsulate an `EdgeViz` and a `NodeViz` respectively. These two panes allow an end user to create a specific `EdgeViz` or `NodeViz`. Hence, it allows a programmatic user to easily create new methods for creating Views. These classes also highlight the versatility of `ListPane`, since the scrolling lists in both GUIs (Source end shape, Target end shape, Shape), are actually instantiations of the `ListPane` class. In these cases, the search portion of `ListPane` has been disabled and the scrolling list portion has been sized to fit one row. In addition, no buttons were added to the `ListPane`.



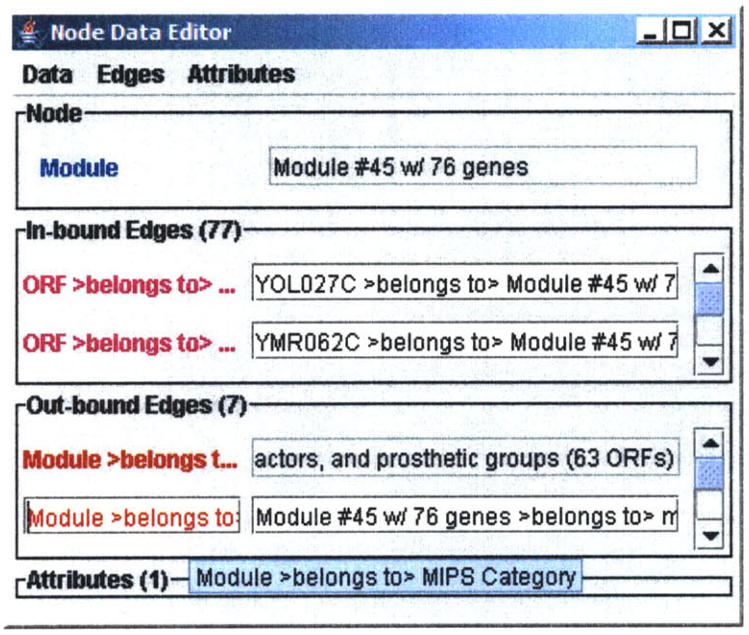
ORF >belongs to> ...	YIR006C >belongs to> Module #258 w/ 7
ORF	YIR006C
Module	Module #258 w/ 7 genes
Directed?	true
Interpretation	belongs to
Expression Condit...	sporulation/mating 29 arrays
Expression Condit...	DNA damage 73 arrays

ORF >belongs to> Module

(a)



(b)



Module: Module #45 w/ 76 genes

ORF >belongs to> ... YOL027C >belongs to> Module #45 w/ 7

ORF >belongs to> ... YMR062C >belongs to> Module #45 w/ 7

Module >belongs t... actors, and prosthetic groups (63 ORFs)

Module >belongs to Module #45 w/ 76 genes >belongs to> m

Module >belongs to> MIPS Category

(c)

Figure 14. Node and Edge Data Editor Pane

(a) This is an example of `EdgeDataPane` where the mouse cursor (currently unseen) is resting over the Type field of the Edge section. Hence, the pop-up box with the text `ORF >belongs to> Module` can be seen floating under the Edge section. White boxes represent fields that can currently be modified by the end user. In this case, the end user can only edit the values of Attributes.

(b) This is an example of the `EdgeDataPane` where all sections have been collapsed. A section can be collapsed and expanded by double-clicking on the title of the section.

(c) This is an example of the `NodeDataPane` where the user is currently editing a type field in the Out-Bound Edges section, hence, the pop-up blue box with the text `Module >belongs to> MIPS Category` can bee see floating under the Out-Bound Edges section. The Attributes section has been collapsed.

Likewise, the classes `EdgeDataPane` and `NodeDataPane` allow an end-user to modify data related to `Edge` and `Node` objects. Since `Edge` and `Node` objects do not contain any information regarding the `Attribute` objects that an edge or node maps to, `EdgeDataPane` and `NodeDataPane` require access to the current `SonnetData`.

These panels contain numerous time-saving and error-detection features for both the end user and the programmatic user. For example the programmatic user can specify which fields are editable. If a field is editable, the end user can toggle its editing mode by double-clicking on the field. To protect from accidental changes in data, an end-user might wish to selectively turn off the editing mode of certain fields. Moreover, these panes will automatically check the text that the user inputs for compatibility with Sonnet. For instance, if all Attributes of Type "Entrez Search" are valid URLs, then, all edits of the value field of an "Entrez Search" Attribute must also result in valid URLs.

3.5 Testing

Sonnet makes extensive use of JUnit[36] testing [41]. In general, any non-GUI functionality, including event handling, is tested. Test classes are named after the class they test. For example, `EdgeTest` is a JUnit test class for `Edge`. Any package with JUnit tests also contains the class `AllTests`, which is a JUnit suite (one per package). Each `AllTests` class will execute every test class in that package as well as the `AllTests` of sub packages. This culminates in the `AllTests` class in the `sonnet` package. Hence, running `AllTests` in the `sonnet` package will run all tests in all packages of Sonnet. In general, `sonnet.AllTests` should be executed whenever functionality changes or when new functionality is added. When new functionality is added, a test for that functionality should be created in the appropriate test class.

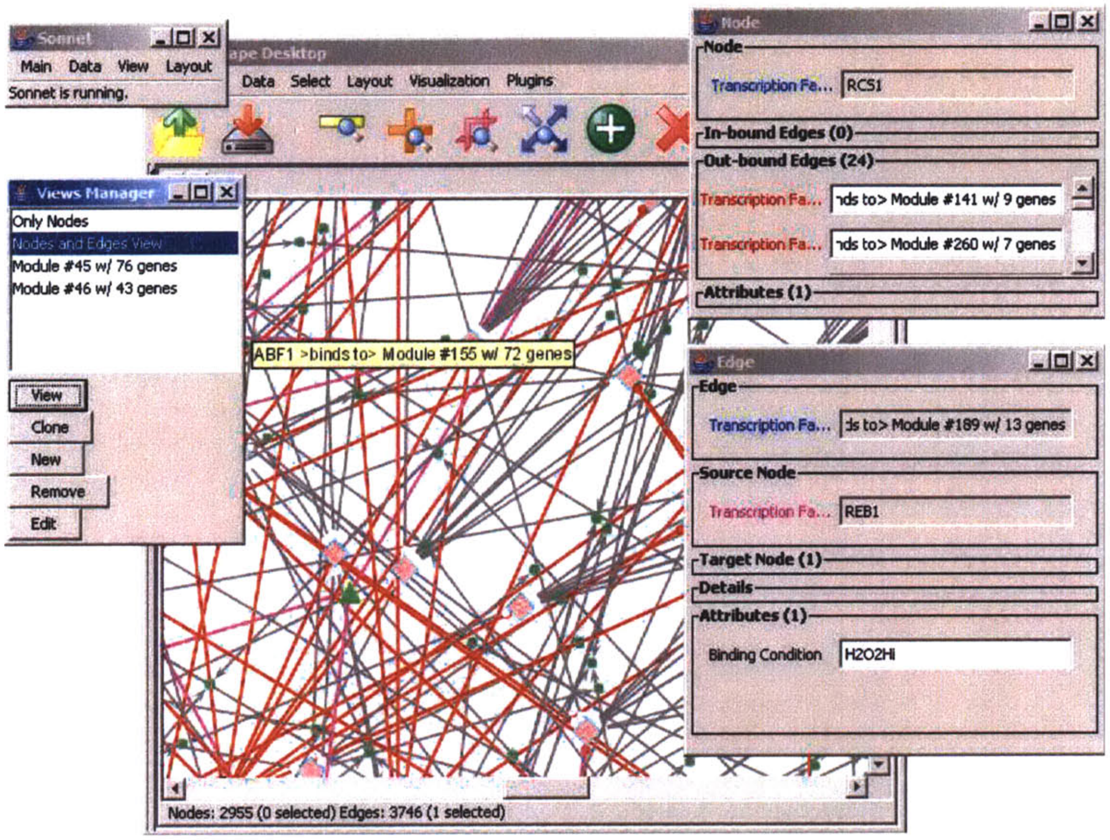
[36] http://www.junit.org/index.htm

4.0 Results

Sonnet's main contribution is not its effectiveness as-is, but rather its abstraction of common code and ease of extension. Sonnet can readily be extended to interact with other layout programs and new methods for creating views can easily be created. Moreover, the GUIs classes in `sonnet.GUI` implement components for helping programmatic users create new graphical interfaces. It is our hope that Sonnet becomes the core of many specialized tools, and the platform from which a universal data, view, and layout management tool is built.

For the casual user, Sonnet is currently stable and useful as a standalone network modeling tool. Combined with Cytoscape, it also useful as a rendering and interactive tool. However, much can be done to further improve the end user experience.

4.1 Usage



Sonnet is running.

ABF1 >binds to> Module #155 w/ 72 genes

Transcription Fa... RCS1

Transcription Fa... ∩ds to> Module #141 w/ 9 genes

Transcription Fa... ∩ds to> Module #260 w/ 7 genes

Transcription Fa... ds to> Module #189 w/ 13 genes

Transcription Fa... REB1

Binding Condition H2O2Hi

Nodes: 2955 (0 selected) Edges: 3746 (1 selected)

Figure 15. Example of Sonnet in use with Cytoscape
Here is example where a graph of a network has been rendered by Cytoscape and the data and views are managed by Sonnet. The Cytoscape frame is in the background. In the foreground, clock-wise from the upper-left hand corner are the main Sonnet menu bar and frame, the Node data editor frame for the last selected Node, the Edge data editor frame for the last selected Edge, and the Views Manager frame with the view of the current layout selected (view is names "Nodes and Edges View").

When combined with Cytoscape, Sonnet is an efficient method for managing data and views. Figure 15 demonstrates how the typical runtime environment of Sonnet and Cytoscape might appear. When used with Cytoscape, a user executes Cytsocape and then loads the Sonnet plugin. Then, using Sonnet, the user can load existing data and views, as well as modify existing views and create new views. A user can instruct Sonnet to render a view, which creates a graph in the Cytoscape frame. By selecting an edge or a node in the Cytoscape frame, the user triggers Sonnet to bring up panels that summarize data about the node or view. These panels also allow the user to modify the data. In the edge or node data editor panes, the user can add and delete attributes and other state information. In addition, selecting the edge section of the node data pane will cause the selected edge to become selected in the Cytoscape pane. Similarly, a node section in the edge pane will cause Cytoscape to select the node.

4.2 Future work

Besides integrating Sonnet with more layout programs and creating more methods for creating views, the most pressing issue with Sonnet is integration of the spatial location of nodes and edges. Currently, the spatial location of nodes and edges is handled entirely by the layout program. However, if Sonnet is to create truly layered graphs (i.e. a sequence of graphs where each succeeding graph simply adds nodes and edges to the previous graphs), then Sonnet must be able to assign the spatial location of nodes and edges. This could be approached in two ways. First, Sonnet could render a layout via a layout program and ask the layout program to return the location of the nodes and edges. In subsequent views, Sonnet could build on those previous locations. A more direct method would be to implement layout programs in Sonnet, so that Sonnet has the capability of determining where nodes and edges should be located. In this manner, layout programs would only be used to render the graphics and perhaps provide interactivity.

5.0 Works Cited

[1] P. Uetz, L. Giot, G. Cagney, T. A. Mansfield, R. S. Judson, J. R. Knight, D. Lockshon, V. Narayan, M. Srinivasan, P. Pochart, A. Qureshi-Emili, Y. Li, B. Godwin, D. Conover, T. Kalbfleisch, G. Vijayadamodar, M. Yang, M. Johnston,

S. Fields, and J. M. Rothberg, "A comprehensive analysis of protein-protein interactions in Saccharomyces cerevisiae," *Nature*, vol. 403, pp. 623-7, 2000.

[2] T. Ito, T. Chiba, R. Ozawa, M. Yoshida, M. Hattori, and Y. Sakaki, "A comprehensive two-hybrid analysis to explore the yeast protein interactome," *Proc Natl Acad Sci U S A*, vol. 98, pp. 4569-74, 2001.

[3] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein, "Cluster analysis and display of genome-wide expression patterns," *PNAS*, vol. 95, pp. 14863-14868, 1998.

[4] E. Segal, R. Yelensky, and D. Koller, "Genome-wide discovery of transcriptional modules from DNA sequence and gene expression," *Bioinformatics*, vol. 19 Suppl 1, pp. I273-I282, 2003.

[5] A. Tanay, R. Sharan, M. Kupiec, and R. Shamir, "Revealing modularity and organization in the yeast molecular network by integrated analysis of highly heterogeneous genomewide data," *PNAS*, pp. 0308661100, 2004.

[6] Z. Bar-Joseph, G. K. Gerber, T. I. Lee, N. J. Rinaldi, J. Y. Yoo, F. Robert, D. B. Gordon, E. Fraenkel, T. S. Jaakkola, R. A. Young, and D. K. Gifford, "Computational discovery of gene modules and regulatory networks," *Nat Biotechnol*, vol. 21, pp. 1337-42, 2003.

[7] A. C. Gavin, M. Bosche, R. Krause, P. Grandi, M. Marzioch, A. Bauer, J. Schultz, J. M. Rick, A. M. Michon, C. M. Cruciat, M. Remor, C. Hofert, M. Schelder, M. Brajenovic, H. Ruffner, A. Merino, K. Klein, M. Hudak, D. Dickson, T. Rudi, V. Gnau, A. Bauch, S. Bastuck, B. Huhse, C. Leutwein, M. A. Heurtier, R. R. Copley, A. Edelmann, E. Querfurth, V. Rybin, G. Drewes, M. Raida, T. Bouwmeester, P. Bork, B. Seraphin, B. Kuster, G. Neubauer, and G. Superti-Furga, "Functional organization of the yeast proteome by systematic analysis of protein complexes," *Nature*, vol. 415, pp. 141-7, 2002.

[8] Gene Ontology Consortium, "The Gene Ontology (GO) database and informatics resource," *Nucl. Acids. Res.*, vol. 32, pp. D258-261, 2004.

[9] H. Mewes, K. Heumann, A. Kaps, K. Mayer, F. Pfeiffer, S. Stocker, and D. Frishman, "MIPS: a database for protein sequences and complete genomes," *Nuc. Acids Res.*, vol. 27, pp. 44:48., 1999.

[10] J. D. Watson and F. H. Crick, "The structure of DNA," *Cold Spring Harb Symp Quant Biol*, vol. 18, pp. 123-31, 1953.

[11] B. Alberts, *Molecular biology of the cell*, 4th ed. New York: Garland Science, 2002.

[12] F. Sanger, S. Nicklen, and A. R. Coulson, "DNA sequencing with chain-terminating inhibitors," *Proc Natl Acad Sci U S A*, vol. 74, pp. 5463-7, 1977.

[13] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. Gabor Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran,

R. Charlab, K. Chaturvedi, Z. Deng, V. Di Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. J. Heiman, M. E. Higgins, R. R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Wang, A. Wang, X. Wang, J. Wang, M. Wei, R. Wides, C. Xiao, C. Yan, et al., "The sequence of the human genome," *Science*, vol. 291, pp. 1304-51, 2001.

[14] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, J. Howland, L. Kann, J. Lehoczky, R. LeVine, P. McEwan, K. McKernan, J. Meldrim, J. P. Mesirov, C. Miranda, W. Morris, J. Naylor, C. Raymond, M. Rosetti, R. Santos, A. Sheridan, C. Sougnez, N. Stange-Thomann, N. Stojanovic, A. Subramanian, D. Wyman, J. Rogers, J. Sulston, R. Ainscough, S. Beck, D. Bentley, J. Burton, C. Clee, N. Carter, A. Coulson, R. Deadman, P. Deloukas, A. Dunham, I. Dunham, R. Durbin, L. French, D. Grafham, S. Gregory, T. Hubbard, S. Humphray, A. Hunt, M. Jones, C. Lloyd, A. McMurray, L. Matthews, S. Mercer, S. Milne, J. C. Mullikin, A. Mungall, R. Plumb, M. Ross, R. Shownkeen, S. Sims, R. H. Waterston, R. K. Wilson, L. W. Hillier, J. D. McPherson, M. A. Marra, E. R. Mardis, L. A. Fulton, A. T. Chinwalla, K. H. Pepin, W. R. Gish, S. L. Chissoe, M. C. Wendl, K. D. Delehaunty, T. L. Miner, A. Delehaunty, J. B. Kramer, L. L. Cook, R. S. Fulton, D. L. Johnson, P. J. Minx, S. W. Clifton, T. Hawkins, E. Branscomb, P. Predki, P. Richardson, S. Wenning, T. Slezak, N. Doggett, J. F. Cheng, A. Olsen, S. Lucas, C. Elkin, E. Uberbacher, M. Frazier, et al., "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, pp. 860-921, 2001.

[15] A. Goffeau, B. G. Barrell, H. Bussey, R. W. Davis, B. Dujon, H. Feldmann, F. Galibert, J. D. Hoheisel, C. Jacq, M. Johnston, E. J. Louis, H. W. Mewes, Y. Murakami, P. Philippsen, H. Tettelin, and S. G. Oliver, "Life with 6000 genes," *Science*, vol. 274, pp. 546, 563-7, 1996.

[16] E. M. Southern, "Detection of specific sequences among DNA fragments separated by gel electrophoresis," *J Mol Biol*, vol. 98, pp. 503-17, 1975.

[17] S. P. Fodor, J. L. Read, M. C. Pirrung, L. Stryer, A. T. Lu, and D. Solas, "Light-directed, spatially addressable parallel chemical synthesis," *Science*, vol. 251, pp. 767-73, 1991.

[18] M. Schena, R. A. Heller, T. P. Theriault, K. Konrad, E. Lachenmeier, and R. W. Davis, "Microarrays: biotechnology's discovery platform for functional genomics," *Trends Biotechnol*, vol. 16, pp. 301-6, 1998.

[19] R. Ekins and F. W. Chu, "Microarrays: their origins and applications," *Trends Biotechnol*, vol. 17, pp. 217-8, 1999.

[20] M. Schena, D. Shalon, R. W. Davis, and P. O. Brown, "Quantitative monitoring of gene expression patterns with a complementary DNA microarray," *Science*, vol. 270, pp. 467-70, 1995.

[21] A. P. Blanchard, Kaiser R. J., and H. L. E., "High-density oligonucleotide arrays," *Biosensors and Bioelectronics*, vol. 11, pp. 687-690, 1996.

[22] B. Ren, F. Robert, J. J. Wyrick, O. Aparicio, E. G. Jennings, I. Simon, J. Zeitlinger, J. Schreiber, N. Hannett, E. Kanin, T. L. Volkert, C. J. Wilson, S. P. Bell, and R. A. Young, "Genome-Wide Location and Function of DNA Binding Proteins," *Science*, vol. 290, pp. 2306-2309, 2000.

[23] J. R. Yates, 3rd, "Mass spectral analysis in proteomics," *Annu Rev Biophys Biomol Struct*, vol. 33, pp. 297-316, 2004.

[24] A. H. Tong, M. Evangelista, A. B. Parsons, H. Xu, G. D. Bader, N. Page, M. Robinson, S. Raghibizadeh, C. W. Hogue, H. Bussey, B. Andrews, M. Tyers, and C. Boone, "Systematic genetic analysis with ordered arrays of yeast deletion mutants," *Science*, vol. 294, pp. 2364-8, 2001.

[25] T. R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, C. D. Bloomfield, and E. S. Lander, "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring," *Science*, vol. 286, pp. 531-7, 1999.

[26] S. Datta, "Comparisons and validation of statistical clustering techniques for microarray gene expression data," *Bioinformatics*, vol. 19, pp. 459-66, 2003.

[27] T. I. Lee, N. J. Rinaldi, F. Robert, D. T. Odom, Z. Bar-Joseph, G. K. Gerber, N. M. Hannett, C. T. Harbison, C. M. Thompson, I. Simon, J. Zeitlinger, E. G. Jennings, H. L. Murray, D. B. Gordon, B. Ren, J. J. Wyrick, J.-B. Tagne, T. L. Volkert, E. Fraenkel, D. K. Gifford, and R. A. Young, "Transcriptional Regulatory Networks in Saccharomyces cerevisiae," *Science*, vol. 298, pp. 799-804, 2002.

[28] N. Friedman, "Inferring Cellular Networks Using Probabilistic Graphical Models," *Science*, vol. 303, pp. 799-805, 2004.

[29] E. Segal, M. Shapira, A. Regev, D. Pe'er, D. Botstein, D. Koller, and N. Friedman, "Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data," *Nat Genet*, vol. 34, pp. 166-76, 2003.

[30] E. Segal, H. Wang, and D. Koller, "Discovering molecular pathways from protein interaction and gene expression data," *Bioinformatics*, vol. 19 Suppl 1, pp. i264-71, 2003.

[31] G. D. B. Iannis G. Tollis, Peter Eades, Roberto Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs*. Upper Saddle River, NJ: Pearson Education, 1998.

[32] V. Batagelj and A. Mrvar, "PAJEK -- Program for large network analysis," *Vladimir Batagelj and Andrej Mrvar. PAJEK -- Program for large network analysis. Connections, 21:47--57, 1998.*, 1998.

[33] Emden R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw. Pract. Exper.*, vol. 00, pp. 1-5, 1999.

[34] F. Kolpakov, E. Ananko, G. Kolesov, and N. Kolchanov, "GeneNet: a gene network database and its automated visualization," *Bioinformatics*, vol. 14, pp. 529-537, 1998.

[35] B. J. Breitkreutz, C. Stark, and M. Tyers, "The GRID: the General Repository for Interaction Datasets," *Genome Biol*, vol. 4, pp. R23, 2003.

[36] M. Fröhlich and M. Werner, "daVinci V2.0.x Online Documentation," vol. 1996. Universität Bremen, 1996.

[37] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks," *Genome Res.*, vol. 13, pp. 2498-2504, 2003.

[38] E. R. Tufte, *The visual display of quantitative information*. Cheshire, Conn. (Box 430, Cheshire 06410): Graphics Press, 1983.

[39] "Java (tm) 2 Platform, Standard Edition v. 1.4," 1.4.2 ed: Sun Microsystems, 2003.

[40] J. Bloch, *Effective Java : programming language guide*. Boston: Addison-Wesley, 2001.

[41] V. Massol and T. Hutsed, *JUnit in action*. Greenwich, Conn.: Manning, 2004.

6.0 Appendix

6.1 Terminology and Conventions

Sonnet refers to this entire project and specifically to the Java software that this project is has created. SONNET, a **S**imple **O**ntology for **Net**works, refers to the data representation specified in section 2.1. Outside of discussing SONNET, a reference to an "edge" is a reference to an Edge object and all Attribute objects associated with said edge. Likewise, a reference to a "node" is a reference to a Node object and all Attribute objects associated with said node.

"Data" almost always refers to data that is being modeled. In the examples used this thesis, it was a regulatory network for yeast. Hence, data included the names of genes, the names of modules, MIPS categories, etc. Data regarding the physical appearance of edges and nodes is referred to as "visual characteristics," "visual attributes," or simply "visuals." A view is a mapping between nodes and edges and sets of visual characteristics. Another important aspect of a view is the inclusion and exclusion of nodes and edges. In other words, not all edges and nodes have visual characteristics in every view. A layout (i.e. graphic, network graph) is picture representation of a view. A layout may or may not be interactive. A layout is usually created or rendered from a view.

References to "a class" refer to the Java implementation of a class object[37]. Although a Java class can roughly be thought of as an object, from object-oriented programming, there is no simple definition for what a class can or cannot do. An "interface" is an object that only specifies requirements. In object-oriented programming terms, an interface is known as a specification or as requirements. A class can implement an interface by satisfying all the requirements of the interface. A class can also inherit or extend the functionality of another class. The extending class is called a child class, and the class that was extended is called the parent. When an interface extends another interface, it simply means that any class that extends the child interface must meet the specifications of both the child interface and the parent interface. When the class for code is read and interpreted by the Java virtual machine, the class is said to be instantiated, and the resulting object in memory is referred to as an instance of the class.

Section 3.0 discusses the implementation of Sonnet in Java. Java specific references—class names, interface names, data types, etc.—are displayed in the `Courier New` fixed width font. Examples related to the SONNET file type are also displayed in `Courier New` font.

6.2 *Key for software dependency figures*

[37] http://java.sun.com/docs/books/tutorial/java/concepts/class.html

The diagrams used in the implementation section are meant to show both relationships and dependencies. There are a few relatively simple conventions.

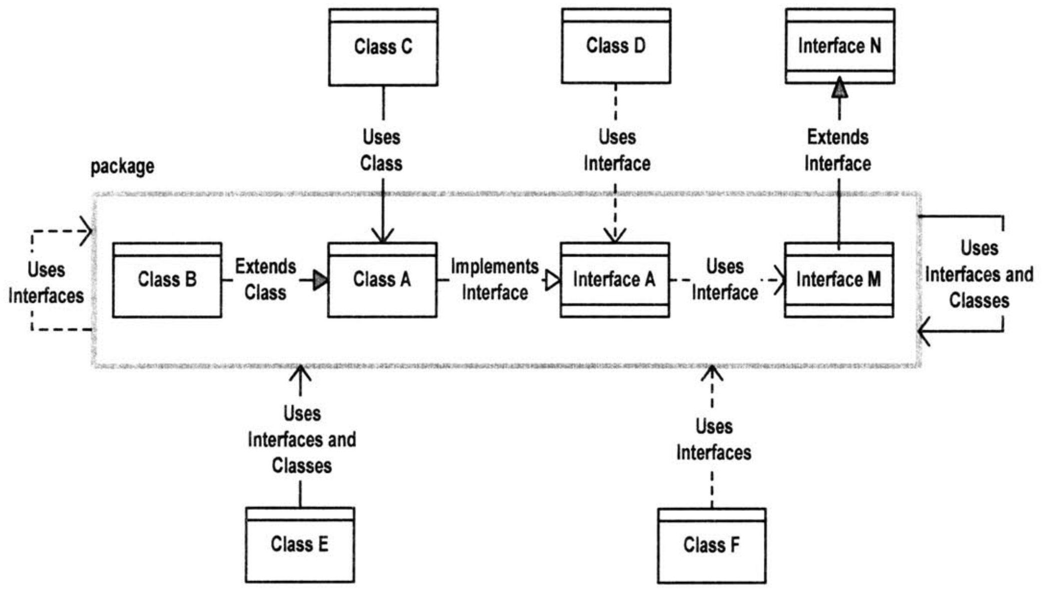


Figure 16. Conventions for code centric figures
This diagram is a model software system that demonstrate the conventions used in code centric figures throughout this Thesis.

A package is represented by an empty box. The name of the package is above the top right hand corner of the box. For brevity, the part of the package name that is identical for all packages (`edu.mit.lcs.psrg`) is omitted in all diagrams.

A class is represented by a box with a single bar above the name of the class. An interface is represented by a box with two bars; one above and one below the name of the interface.

An arrow with a fully formed white head represents implementation. The arrow extends from the implementing class to the implemented class.

An arrow with a fully formed shaded head represents extension or inheritance. The arrow originates from the class that is extending and terminates at the class that is being extended. (Hence, it also represents inheritance. The arrow goes from the class that is inheriting to class that has been inherited). This arrow has a synonymous meaning between two Interfaces.

A dashed arrow between two objects represents use of an interface. The most basic form is when a dashed arrow extends from a class to an interface. When a dashed arrow extends between a class and a package, it not only means that the class uses interfaces in the package, but also it means that the class does not use any classes in the package. This means, that the source class must be using other classes that implement the interfaces in the package. Also, given the definition of an interface, only dashed arrows are allowed to extend from an interface.

A solid arrow with out a fully formed head represents use of a class. This is most obvious when the arrow is between two classes. When the terminal end of this arrow is a packages, it also implies that the source object uses the interfaces in the terminal package. When this arrow extends between a class and a class in another package, it implies that the source class only uses the terminal class in the terminal package.

Finally, for brevity, certain types of connections will usually be omitted. Specifically, if Class M were to implement Interface M, and Class M was the only class to implement Interface M, the "uses class" arrow would be omitted from Class A to Class M, since Interface A (the Interface for Class A) uses Interface M, which would then automatically require Class A to use Class M since Class M is the only implementation of Interface M.

6.3 Known Unresolved Issues

Package definition no longer matches reality.

When this project was initiated, I was working in the Laboratory for Computer Science (LCS) and specifically in the Programming Systems Research Group (PSRG). Since then, LCS has combined with the Artificial Intelligence Laboratory (AI) to form the Computer Science and Artificial Intelligence Laboratory (CSAIL). In addition, PSRG has changed focus and has become the Computational Genomics Research Group (CGRG).

Fix `org.apache.log4j` *import*

Currently the log4j package has been refactored to be located at `edu.mit.lcs.psrg.sonnet.org.apache.log4j`. This is due to an idiosyncrasy of Cytoscape, where packages located outside of the main plug-in package are not loaded. Since Cytoscape does not use the `log4j` package, a temporary solution was to move all needed packages from `log4j` to a location where Cytoscape would load it. Hence, the package `org.apache.log4j was` re-factored to the package `edu.mit.lcs.psrg.sonnet.org.apache.log4j`, etc. One possible solution is to migrate away from using log4j and instead use a package with similar capabilities `java.util.logging`. Since this package is part to the default Java 1.2 distribution, Cytoscape should have ready access to it.

Normalize logging

Although logging is a powerful debugging tool during implementation, it should be an equally useful tool during maintenance and upgrades to the system. However, most of the logging performed is on the "debug" threshold. One should examine the code closely and determine when other messages on the "warn," "info", etc., thresholds. In addition, error logging messages should be captured to a text file that can be examined when the program fails. A relevant place to start is by rethinking the initialization of the logging apparatus in the `Sonnet` in the `setupLogger()` method.

Improve value handling in **`edu.mit.lcs.psrg.sonnet.data.Value`**

When converting View to SonnetData and saving, integers are saved as floating point decimals (i.e. 128.0 instead of 128). Hence, when View is created out of a SonnetData created from such a file, integers must be read as floating point numbers and then cast into integers.

Use regular expressions in **`SonnetFileReader`** *and* **`SonnetFileWriter`**

The current implementation of `SonnetFileReader` uses the class `java.lang.StringTokenizer`, which can only tokenize with tokens of one character. Hence, tokens such as "`::`" or "==" are invalid. Because attributes often include URL references, many non-alphanumeric characters are disqualified from being tokens (e.g. ":", "=", "?", etc.), and in fact, almost all characters are found in legal URLs, rendering most characters unsuitable for tokens. This problem can be resolved by modifying `StringFileReader` to use the regular expression tools found in the `java.util.regex` package.