Integrated Web-Based Analysis of High-Dimensional Biological Information

by

Christopher Bowron Kingsley

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Biological and Medical Informatics

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, SAN FRANCISCO

[redacted]

Advisor: Ajay Jain Christopher B. Kingsley

Integrated Web-Based Analysis of High-Dimensional Biological Information

Acknowledgements

At some point during the preparation of my PhD thesis, it occurred to me that this document was likely to be the last gasp of my long phase as a student. Such milestones often inspire a moment of reflection, so I paused for a moment and tried to recount the series of steps that led from my days as a dazed 18 year old freshman at Berkeley to my current position. What struck me first was how so many of these steps had been the result of seemingly minor or serendipitous events. One such event occurred during my freshman year of college when I noticed a copy of Jim Watson's 'Double Helix' sitting on a bookshelf. The inspiration I received by reading that book sent me down a ten year long path of splicing and dicing DNA, as well as regular exposure to chemical fumes and radiation.

In contrast to that single transformative moment, my transition from molecular biology to bioinformatics was a journey of several smaller steps. My initial interest in bioinformatics arose as a result of my background as an experimental molecular biologist, and my frustration with the pace of discovery. Although we don't care to admit it, a single moment of inspiration followed by six months of hard, repetitive work at the bench and the tissue culture hood is enough to wear down even the most idealistic among us. Thankfully (and out of the blue) a coworker of mine at Chiron Corporation suggested that I apply for a job opening in the bioinformatics department, and the next ten years of my professional life was settled. I would like thank my coworkers at Chiron, especially

George Lamson, David Pot and Altaf Kassam for making the transition a smooth and enjoyable one.

The decision to return to graduate school was not an easy one, but the professors in the Biological and Medical Informatics Graduate Program made the transition as smooth as possible. I acknowledge all the members of the department, but particularly Tom Ferrin, Patsy Babbitt, and Teri Klein for all their advice and support.

No graduate thesis is a product of isolated effort, and I have many people to thank for their support and assistance in this effort. First and foremost, I would like to thank my family. I acknowledge my father for instilling in me an interest in science at an early age. Almost everything else I owe to my mother. My Grandmother, my sister, and uncle were also extremely supportive, and I always appreciated having the excuse of a family visit to justify my forays into the Sawtooth Mountains and the jungle of Manhattan.

Thanks go to my good friends John, Arjuna, and Yoon for keeping my spirits up during those dark days when I couldn't help but wonder why a man of my advancing years had given up a stable (if sometimes tedious) position in the real world in order to return to the ivory towers of academia. Thanks also to Daniel Murphy and Tobias Dansen for temporarily chasing the pointer errors from my brain by periodically dragging me up Mt. Tamalpais on my bicycle. Beautiful scenery, endorphins, hypoxia and a heart rate of 170 beats per minute have a funny way of putting things in their proper perspective.

No applied bioinformatics project can go very far without data. I thank my many experimental collaborators for their willingness to share their hard earned results with me. I would first thank Joe Gray and the members of his laboratory, especially Koei

Chin, Wen-Lin Kuo, Rich Neve, and Daniel Polikof for their data from ovarian and breast tumors. Thanks to Frank McCormick, Luika Timmerman and Jennifer Yeh for their data from breast cancer cell lines. Thanks also to Martin McMahan and Stephan Gysin for sharing their data from cultured pancreatic cancer cell lines.

I would also like to thank the members of the Jain laboratory for their camaraderie and advice. Lawrence Hon could always be relied on for a good conversation over lunch on topics that varied from coding strategies to job prospects to the latest computer games. Barbara Novak was instrumental in helping to develop and refine the Magellan database, and Karen Kimura was also very helpful in this area. Adam Olshen and Jane Fridlyand were two postdoctoral statisticians in the lab who (thankfully) believed that helping me decipher the mysteries of statistics was not an utter waste of their time. Thanks also to Tuan Pham and Taku Tokuyasu, who each shared an office with me and were able to withstand my many moods while maintaining a pleasant demeanor. I am indebted to them all.

Lastly, I must thank my thesis adviser Ajay Jain for all of his guidance and support. I have worked in laboratories in industry and academia for almost 20 years, and I have found that good mentors in this field are a precious commodity. Ajay is one of those rare advisers who has an abundance of ideas and advice, but who knows when to give you a shove and when to leave you alone. In some respects, running a productive laboratory is as much about psychology as it is a about intellect, and Ajay excelled at both.

In closing, one event that was jogged from my memory while reminiscing over the past several decades was a conversation I had with a coworker during my days as an

experimentalist. After a long day of pipetting little drops of liquid into little plastic tubes, my colleague and I were lamenting the pace of our respective projects when he made a statement that stuck with me: "One day, we'll look back at what we're doing now and realize that we were working with clubs and spears." While I would not say that we have reached this point just yet, I do see signs of its pending arrival, and I believe that the journey toward it will continue to be a fruitful and fascinating one.

Abstract

Integrated Web-Based Analysis of High-Dimensional Biological Information

Christopher B. Kingsley

Recent advances in high throughput biological methods allow researchers to generate enormous amounts of data from a single experiment. In order to extract meaningful conclusions from this tidal wave of data, it will be necessary to develop analytical methods of sufficient power and utility. It is particularly important that biologists themselves perform many of these analyses, such that their background knowledge of the experimental system under study can be used to interpret results and direct further inquiries.

This dissertation describes the development of a web-based system, Magellan, which allows the upload, storage, and analysis of multivariate data and textual or numeric annotations. Data and annotations are treated as abstract entities, to maximize the different types of information the system can store and analyze. Annotations can be used in analyses/visualizations, as a means of sub setting data to reduce dimensionality, or as a means of projecting variables from one data type or data set to another. Analytical methods are deployed within Magellan such that new functionalities can be added in a straightforward fashion.

The Magellan system has been used to analyze a number of cancer genomics data sets. These analyses have involved the development and deployment of a number of analytical methods that relate different types of genomic variables, typically comparative genomic hybridization (CGH), mRNA expression and clinical information. In addition, I have worked with the National Cancer Institute on the Cancer Bioinformatics Grid (caBIG) initiative, to develop and deliver the functionality of Magellan as an open source project available to any researcher.

Approved [illegible] Ajay N. Jain

Table of Contents

Table of Figures

Chapter 1
Introduction

The field of molecular biology was born just over half a century ago as an attempt by researchers in biology, chemistry and physics to understand the molecular basis of biological systems. The result of this collaboration has been fifty years of extraordinary advances toward the realization of this goal. In a single lifetime, we have gone from possessing merely abstract notions of the physical nature of genes to sequencing and characterizing entire mammalian genomes. This transformation of biology was driven by the application of the quantitative molecular techniques of chemistry and physics to the experimental systems of biology and genetics. Although our knowledge of biological systems is still very far from complete, this hybrid approach of studying living systems at the molecular level has revolutionized biology and fundamentally altered our understanding of biological systems, in both the normal and disease state.

As molecular biology enters the 21st century, a new transformation is taking place, this one driven by technological advances that have dramatically increased the magnitude of data available to researchers. Recent developments in high-throughput genomic techniques are generating unprecedented amounts of quantitative biological data. Experimental approaches that only a few years ago required many days of painstaking work and high cost to generate a small amount of information have been supplanted by techniques that can generate tens of thousands of data points in a matter of hours. Large-scale biological data sets containing protein structure, whole genome sequences, proteomics data, microarray based mRNA expression data, and comparative genomic hybridization (to name but a few) have been curated and assembled into publicly available datasets for analysis.

These technical advances and the data arising from them have the potential to affect biology just as profoundly as did the impact of physics and chemistry fifty years ago. The scale of the resulting data, however, makes it virtually impossible for biological researchers to identify trends and patterns through direct examination. In order to extract biologically significant conclusions from the emerging large-scale data, a new field of study has emerged. Just as molecular biology was born of a hybrid of biology, physics, and chemistry, the field of bioinformatics has emerged as a hybrid of molecular biology, computer science, and mathematics. As a discipline, bioinformatics represents an attempt to utilize the techniques of information technology to mine biological data as a means of characterizing and modeling biological systems.

For bioinformatics to fully realize its promise, advances in analytical algorithms must be translated into applications that can be used by the research community. Furthermore, there is a great need for these applications to be usable by the experimentalists who understand the system under study and who generate the data. Unfortunately, there is still a fairly wide gulf between the biological researchers who perform experiments and the computational researchers who analyze the resulting data. This situation is problematic for two reasons. First, there are not enough good bioinformaticists and statisticians to be shared among all of the biological research community. Second, even if a bioinformaticist is available, the interaction between experimentalist and analyst sometimes resembles a game of baseball; data is thrown at the quantitative analysts and the results are hit back to the biologists. It is frequently (and sadly) the case that the biologists do not understand the methods that are applied to the data sets that they generate, while the quantitative analysts do not understand the biological system from which the data was generated nor the questions that motivated the experiments in the first place.

My graduate research has centered on the analysis of heterogeneous biological data in cancer. The major accomplishment of this research has been the development of analytical tools that can be used by biological researchers to analyze their data over the internet. The motivation for building these tools was to provide researchers at the UCSF Cancer Center with the ability to analyze and understand the relationship between the different types of data they generate from tumors and tumor models.

This document begins with a brief introduction to cancer and the various types of genomic data currently generated in the study of cancer. It then covers some of the statistical issues associated with analyzing high dimensional cancer genomic data, and several of the analytical methods that are employed to deal with these issues. Several of the existing analytical applications are then discussed, and how the Magellan system differs from them. The Magellan application itself is then covered, including its development, functionality and use cases. I conclude with a discussion of my collaboration with the NCI on the cancer bioinformatics grid (caBIG) project.

Chapter 2
The Biology of Cancer

2.1. Introduction

Cancer is a disease that directly or indirectly affects almost everyone in the developed world. In the United States, cancer is second only to heart disease as a cause of death, responsible for roughly twenty percent of all mortalities. For much of the twentieth century, cancer death rates increased every year as a result of a growing incidence of smoking, greater exposure to environmental carcinogens, dietary changes and the gradual aging of the population. Over the last several years, however, there has been a slight reversal in this trend. From 1993 through 2002, the overall death rate from cancer has declined by 1.1% per year in the United States (Edwards, Brown et al. 2005). While much of this drop can be accounted for by a decline in smoking among males, other factors such as improved screening, early detection, and improved treatments also played a significant role. The goal of the cancer research community is to further this decreasing trend in cancer-associated mortality through the identification of better diagnostic and prognostic biomarkers, the discovery of better therapeutic targets and the development of new classes of drugs.

Cancer is, fundamentally, a disease of excess cellular proliferation. Tumors arise when cells no longer respond to the normal signals that regulate cell division and cell death, and grow unchecked. This situation becomes life threatening when these unregulated cells spread to surrounding tissues, and disrupt the normal functions of the organs they invade. The underlying cause of cancer has been debated for centuries, but over the last fifty or so years a number of lines of evidence have suggested that normal cells become tumors due to alterations in their DNA. First, a majority of the chemical carcinogens whose exposure leads to cancer are mutagens that chemically modify DNA. Second, many cancers run in families, indicating that they are caused by inherited mutations in DNA. Third, several well characterized cancers are associated with specific alterations in the chromosomes, such as microscopically visible translocations. These and other observations have led to the paradigm that cancer is a disease of genetic alterations.

2.2. Cellular Changes in Cancer

For most cancers, incidence increases dramatically with age as shown in Figure 1 (Alberts, Johnson et al. 2002). This observation is best explained by a model in which tumors arise from normal cells through a series of specific changes that accumulate over time, a theory that has been condensed to the mantra 'cancer is a multistep process' (Knudson, Hethcote et al. 1975). This theory hypothesizes that each step in tumor formation occurs in a single cell in a population, causing that cell to acquire a growth advantage. Cancer development can therefore be viewed as a series of bursts of clonal expansions, in which one cell overcomes a bottleneck to growth (Nowell 1976). Leading credence to this theory, experiments using various animal models have shown that a relatively small number of engineered mutant genes can confer abnormal growth phenotypes, though no single gene has been shown do so (Hahn, Counter et al. 1999).

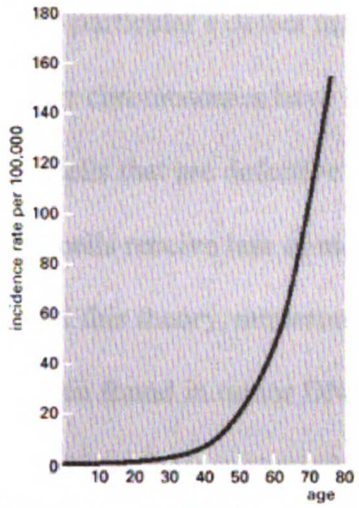


Figure 1– The effect of age on the incidence of ovarian cancer

It has been hypothesized that six different cellular changes must occur to transform a normal cell to a cancerous one: self-sufficiency in growth signals, insensitivity to antigrowth signals, evasion of programmed cell death, limitless replicative potential, angiogenesis, and tissue invasion and metastasis (Hanahan and Weinberg 2000). Although it is believed that most if not all cancers must acquire these properties, each individual tumor may accomplish these changes through different genetic alterations. Thus, tumors that resemble each other histologically may result from alterations in different sets of genes, and those genes can be affected by point mutations, copy number changes, and epigenetic phenomena such as altered chromatin structure.

One characteristic observed in almost all cancers is DNA copy number abnormalities (Albertson, Collins et al. 2003; Rajagopalan and Lengauer 2004). When 19th century researchers first examined cancerous tissues using newly invented and more powerful microscopes, one of the first things they noticed was that cancerous cells frequently had excess chromosomes as well as structural aberrations such as deletions and translocations. It is widely believed that chromosomal instability arises as a result of errors in mitosis, in particular a defect in the mitotic spindle checkpoint. Normal cells do not divide until their chromosomes have fully replicated and are properly aligned in the metaphase plate. Cells that are defective in this checkpoint can divide asymmetrically, such that daughter cells receive less or more than the normal complement of genomic DNA. In support of this theory, numerous mutations in components of the spindle checkpoint have been found in tumor DNA (Cahill, Lengauer et al. 1998).

Some researchers have suggested that copy number abnormalities may be a consequence of tumorigenesis rather than a cause, but several experimental observations argue against this. One such observation is the existence of certain tumors that show very little chromosomal aberration, but instead have defects in DNA repair processes such as mismatch repair (Lengauer, Kinzler et al. 1997). The implication of this finding is that the two classes of tumors have found different mechanisms to generate the genetic alterations that cause cancer. Mismatch repair deficient cells have a high rate of point mutations and small scale genetic instability while other cancers can also acquire genomic changes through amplifications, deletions and translocations.

2.3. The Molecular Basis of Cancer

Many of the early studies into the cellular and molecular events that cause cancer involved transmissible models, such as the retrovirus identified by Peyton Rous as the cause of certain avian sarcomas. This and other tumor viruses gave researchers an experimental toehold into cancer and led to the identification of viral oncogenes that promote growth, and cellular tumor suppressors (as the targets of transforming viral proteins) that inhibit growth. The seminal discovery by Bishop and Varmus that viral oncogenes were, in fact, mutated versions of normal cellular genes linked the discoveries from tumor virus models to cancer biology as a whole (Spector, Smith et al. 1978). Many of these cellular proto-oncogenes were subsequently shown to be mutated in a variety of sporadic human cancers. This finding indicated that even though the vast majority of cancers are not caused by infectious agents, the process of altered gene function is to blame for tumor formation in both sporadic and transmissible cases. In the case of tumor retroviruses, cellular genes are hijacked by the virus and mutated such that the host cells are programmed to proliferate. In the much more common case of non-infectious cancer, normal cellular genes are mutated into forms that promote unregulated cellular growth. The vast majority of cancers are, therefore, caused by alterations in normal genes (Hanahan and Weinberg 2000).

A great deal of the cancer research performed over the last 20 years has been an attempt at determining exactly which mutations in which genes are responsible for the different steps in tumorigenesis. This research has involved many experimental approaches, and has identified many different genes that can contribute to abnormal cellular proliferation when mutated. Cellular oncogenes such as myc (Shen-Ong, Keath et al. 1982), src (Spector, Smith et al. 1978) and ras (Parada, Tabin et al. 1982) have been found as homologues of the genes of tumor viruses. Identification of the cellular targets of viral oncoproteins has led to the discovery of tumor suppressors such as p53 (Harris 1996). Characterization of the cell cycle regulatory machinery in model genetic systems such as yeast has identified homologous mammalian genes such as the cyclins and cyclin dependent kinases that act as key players in cell division (Hartwell, Culotti et al. 1974;

Morgan 1997). Genetic studies in families that show inherited susceptibility to cancer have led to the identification of genes such as APC in colon cancer (Kinzler and Vogelstein 1996), and the BRCA genes in breast cancer (Miki, Swensen et al. 1994). The identification of individual genes that play a role in cancer has been used as experimental entry points into the pathways in which the genes function. By using the techniques of biochemistry, cell biology and genetics, entire pathways have been elucidated starting with the identification of a single pathway member.

2.4. The Future of Cancer Research

It has been over 30 years and hundreds of billions of federal dollars since President Nixon declared a 'war on cancer', and this war has had several notable successes. Not so long ago, the only treatment options available to cancer patients were brute force approaches that targeted dividing cells in general, such as radiation and drugs that inhibit key steps in DNA synthesis. While these drugs were often successful (and are still in use in many cases), they are relatively non-specific and have a number of undesirable side effects. Indeed, some of the harsher chemotherapy regimens are the equivalent of walking a tightrope between killing the tumor and killing the patient.

The identification of key molecular players in cancer development has created a potential for much more specific diagnostics and therapeutics. The past several decades have witnessed the adoption of screening methods such as mammography and PSA that, while far from ideal, have enabled doctors to detect many cancers at much earlier stages than previously possible. Gradually, there has been an increase in the availability of so-called 'magic bullet' therapeutics such as Gleevec (Wang, Healy et al. 2000), Herceptin (Goldenberg 1999), and Iressa (Ciardiello, Caputo et al. 2000) that were designed to target known gene products expressed in tumors and have fewer side effects as a result. While the war on cancer has witnessed prominent successes such as these, there is still much to learn about the molecular origin of cancer and a long way to go in developing better biomarkers and therapies. For example, there have not been nearly as many advances in understanding the later steps in tumorigenesis such as invasion, metastasis and angiogenesis. These later steps may be the most clinically relevant, since it is the acquisition of invasive properties by tumors that make them life threatening and resistant to surgical intervention.

In reflecting on the last 30 years of progress in cancer research, the conceptual breakthroughs in identifying the molecular causes of cancer have been impressive, but their translation into useful diagnostics and treatments has been somewhat underwhelming. Rather than waiting decades for further clinical advances, many patient advocates (including Andy Grove of Intel) have suggested a shift toward an engineering approach in cancer research. This would involve fewer attempts at broad conceptual understanding of the disease and a greater emphasis on translational research such as biomarker discovery. While reasonable people can disagree on this point, most would accept that accelerating the pace of discoveries that positively impact patient's lives is crucial.

A key step in increasing the rate of discovery is increasing the productivity of the researcher. For most of its history, molecular biology has been a 'cottage industry' in which individual laboratories containing small numbers of researchers perform very laborious studies to generate a relatively small amount of data. Until fairly recently, researchers have been limited by the technologies available to them; most experimental protocols were labor intensive and focused on a small number of genes or proteins at a time. Over the past decade or so, several technological advances have been made that allow experimental biologists to generate very large data sets in a relatively short amount of time. These technological innovations include high throughput proteomics, whole genome sequencing, and high throughput chemical screening. Arguably the most important advance, however, has been the development of DNA microarrays. The statistical challenges presented by the scale of microarray data, primarily the volume of data in relation to the number of independent samples in most experimental studies, is what motivated the development of Magellan. Microarray technology is the focus of the next chapter, while the statistical issues involved in analyzing microarray data will be discussed in Chapter 4.

Chapter 3
Microarray Technology in Cancer Research

3.1. Introduction

DNA microarrays are based upon an experimental concept that has been in use for several decades. The experimental technique known as the 'dot blot' involves hybridization of heterogeneous radioactively labeled nucleic acids in solution to purified homogenous DNA samples which have been spotted and immobilized onto nitrocellulose membranes. Dot blots have been used to detect and quantify the amount of a particular nucleic acid sequence in a heterogeneous mixture such as labeled genomic DNA or first strand cDNA. Microarray technology has improved upon this approach using automated techniques to apply each purified DNA sample onto a small solid surface, thereby increasing the number of samples that can be spotted onto a given unit surface area (DeRisi, Iyer et al. 1997). At first glance this may seem like a minor modification, but the practical consequences are quite significant. The degree of miniaturization in currently available microarrays allows for tens of thousands of targets to be placed in a surface area of a few cm^2. This small target size allows for nucleic acids purified from small amounts of biological material to be hybridized to arrays containing large numbers of targets, such that information on the mRNA expression of every known human gene can be obtained in a single hybridization experiment.

3.2. Microarray Platforms

There are currently two major microarray technologies in use: two color hybridization onto spotted DNA arrays and single sample hybridization to lithographically deposited oligonucleotides (Affymetrix arrays). The two-color method is illustrated in Figure 2 (Baggerly and Coombes 2004). Briefly, mRNA is extracted from two different biological samples (one of which is a common reference) and separately used as a template for the synthesis of first strand cDNA labeled with Cy3 or Cy5 fluorescent dies. These cDNA mixtures are combined and hybridized to a glass slide containing individual spots of purified DNA (typically cloned cDNA's amplified by PCR). The ratio of the fluorescence of the test to the reference is normalized and can be used to measure the relative concentration of individual mRNA's in the biological sample.

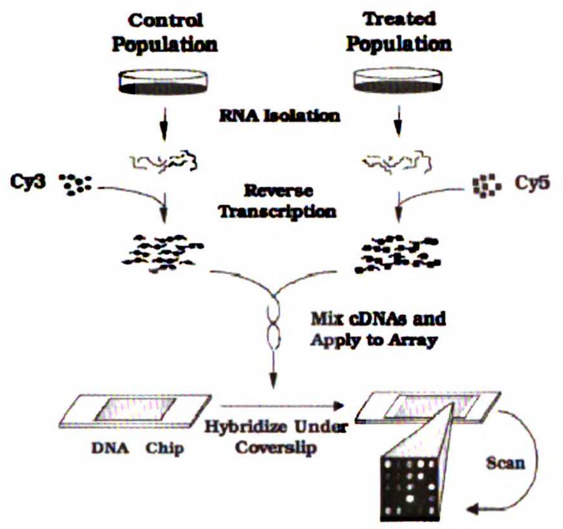


Figure 2 – Overview of two color spotted cDNA microarrays

The arrays commercialized by Affymetrix use ~20 bp long oligonucleotide targets that are generated in situ using lithographic techniques similar to those used in the semiconductor industry. Since the degree of specificity of hybridization of a 20mer oligo is much less than cDNAs of several hundred base pairs, Affymetrix arrays use a combination of several perfectly matched and mismatched targets to quantify each mRNA. A single fluorescently labeled cDNA sample is applied to the array, washed, and scanned, yielding an image such as Figure 3 (www.affymetrix.com). The hybridization signals of all match and mismatch oligos are fed into a proprietary algorithm that combines all data points into a single mRNA expression value for that gene.

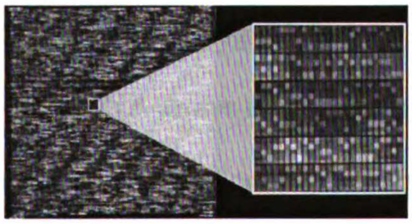


Figure 3 – Scanned image of an Affymetrix microarray

The fact that competing microarray platforms are available begs the question as to which technology is superior. The two color array was the first method to become widely available, and is still the method of choice for custom DNA arrays (i.e. organisms that are not widely studied enough to justify commercial manufacture). Two color arrays have historically been less expensive that Affymetrix arrays, although the cost differential has come down in recent years. Because Affymetrix arrays are fabricated using lithography for which the masks can be generated by an automated process, the design of new Affymetrix arrays is much more rapid than two color arrays, for which new array targets must be separately purified. In terms of quality measures of the normalized data such as reproducibility and signal to noise ratio, Affymetrix technology typically outperforms two color hybridization. Also, since two-color arrays depend on measurement of a sample relative to a reference, it is difficult to compare mRNA expression data from two samples that are measured relative to different references. Currently, many experimental researchers who have the resources to do so are transitioning from two color microarrays to Affymetrix chips because of better data quality, better reproducibility, and decreasing costs.

3.3. Microarray Based Comparative Genomic Hybridization

DNA microarrays were originally developed in order to quantify the concentration of mRNA in biological samples. The majority of microarrays are still used for this purpose, but DNA arrays are increasingly used for other reasons as well. Arrays containing hybridization targets corresponding to different single nucleotide polymorphisms have been used in high throughput genotyping. Chromatin immunoprecipitation using antibodies against specific transcription factors followed by hybridization to arrays containing genomic DNA sequences has been used to determine binding sites for transcription factors in the genome (Buck and Lieb 2004). In cancer biology, however, one of the most important uses of DNA microarrays is the determination of sites of genomic aberration (amplifications and deletions) using the technique of comparative genomic hybrization, or CGH (Pinkel, Segraves et al. 1998; Snijders, Nowak et al. 2001).

CGH involves the hybridization of fluorescently labeled genomic DNA from cancerous tissue and normal tissue to a common target. Briefly, genomic DNA samples are separately purified from normal and cancerous tissue, separately labeled with different fluorescent dies (typically Cy3 and Cy5), mixed and hybridized to the target. Initially, the target was a metaphase spread of chromosomes (FISH, or fluorescent in situ hybridization), but over the last five years this has been supplanted by array based CGH which offers a far superior resolution. For array based CGH, genomic DNA samples from specific regions of the genome are purified and spotted onto the array. For each spot on the array, the normalized ratio of signals from the tumor vs. normal DNA is a direct measure of amplification or deletion of tumor DNA from that region of the genome (Pinkel, Segraves et al. 1998).

Currently, two color arrays are used for the majority of comparative genomic hybridization (CGH) studies. CGH arrays typically contain on the order of several thousand spotted genomic DNA's, although CGH arrays containing approximately 30,000 spots are in development. Recently, Affymetrix single nucleotide polymorphism (SNP) genotyping chips have also been used for CGH. These arrays are capable of genotyping over 100,000 SNP's in a single hybridization. Since Affymetrix genotyping arrays can distinguish between different SNP alleles on the maternal vs. paternal chromosomes, they also have the advantage of distinguishing between aberrations on the two copies of each chromosome (Nannya, Sanada et al. 2005).

3.4. The Nature and Source of the Primary Signal in DNA Microarray Experiments

The ability of Microarray experiments to comprehensively measure cellular characteristics such as mRNA expression and DNA copy number have allowed researchers to pursue several fundamental questions about cancer biology that were unattainable prior to the advent of the technology. One such question is the degree to which tumorigenesis leads to changes in mRNA gene expression, as compared to the signature that was present in the normal parental cells. Several publications have performed microarray analysis on cancers from different cell types of origin (lung, kidney, ovary, colon, etc), and clustered the data together; in most cases, clusters derived from a given cell type tend to group together (Ross, Scherf et al. 2000). This result indicates that the primary signal in mRNA expression data in cancer does not reflect the tumor state, but the cell type of origin. For example, one of the earliest published attempts at performing molecular diagnosis on tumors using microarray gene expression data involved the classification of AML (acute myeloid leukemia) from ALL (acute lymphoblastic leukemia). The genes that were strongly differentially expressed between these two classes were used to build a classifier that could discriminate between AML and ALL (Golub, Slonim et al. 1999). These genes, however, are largely markers of myeloid or lymphoid origin, as opposed to being directly related to cancer.

This fact somewhat complicates the interpretation of analyses of differentially expressed genes in collections of cancers. It is sometimes the case that cancers that were thought to be a single disease are actually derived from fundamentally different cell types. For example, a recent study of breast tumors using microarray based mRNA gene expression data clustered tumors into two broad classes, indicating the presence of samples derived from two distinct cell lineages (possibly of epithelial and mesenchymal origin) in the collection of tumors (van 't Veer, Dai et al. 2002). While the genes that distinguish these two classes may prove useful as diagnostic markers, it seems unlikely that very many of them will be good therapeutic targets since they are more likely to be markers of cell lineage than to be causative agents in cancer.

Figure 4 shows an agglomerative clustering of cell-lines based on mRNA expression data. The cell-lines with common tissues of origin tend to cluster together. This is particularly evident in the melanoma and colon cancer cases. A somewhat contrasting case is presented by the cell-lines derived from breast tumors. The eight samples representing these cell-lines are spread throughout the dendrogram, again indicating a greater degree of heterogeneity among the collection of breast tumors.

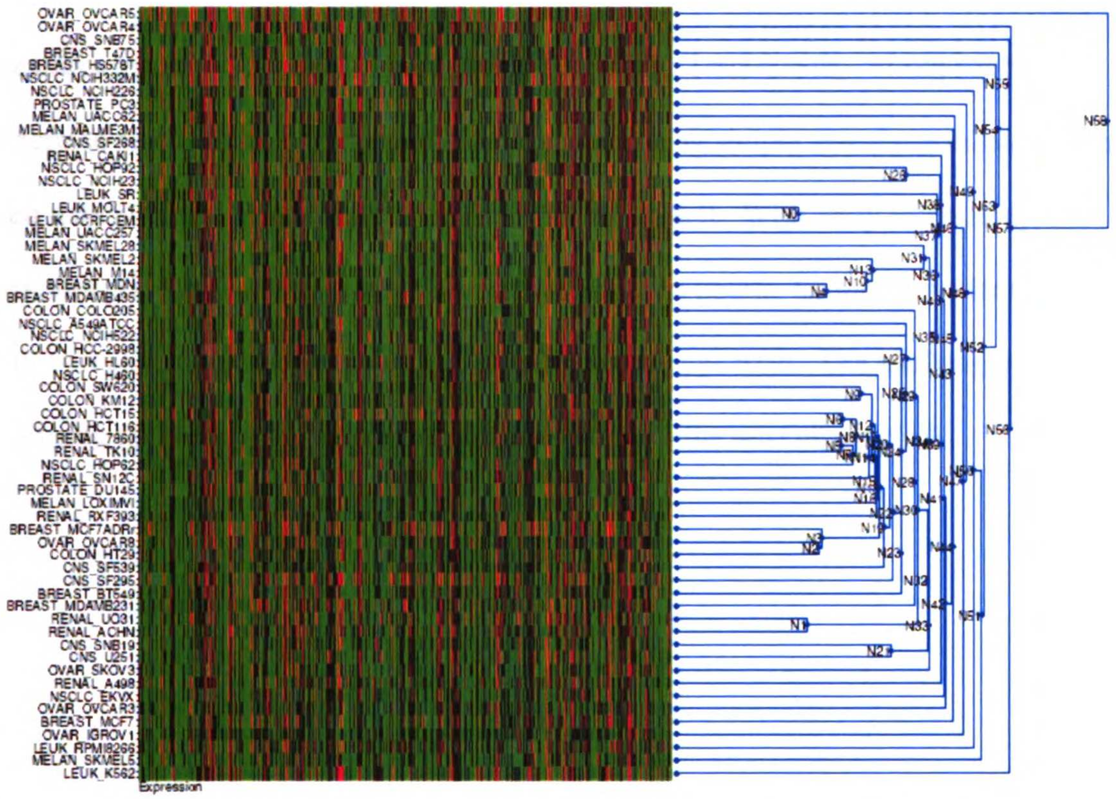


Figure 4 – Clustering of tumor samples of different cellular origin using Affymetrix mRNA expression data from the NCI60 cell lines.

3.5. Conclusion

The advent of microarray technology has dramatically increased the amount of information generated by cancer researchers. Prior to the use of microarrays, quantitation of mRNA involved time consuming procedures such as Northern blots or RNase protection which only measured a small number of genes at a time. Copy number measurements using FISH were subject to fairly low resolution (~20Mb). Microarrays have increased both the productivity and accuracy of both types of experiments, to the point where a single researcher can comprehensively analyze mRNA expression and CGH on a large sample of tumors in a matter of weeks.

While these technological advances were the dream of cancer researchers two decades ago, they do create some difficulties as well. Traditionally, biologists have generated small enough amounts of data that examining simple visual representations of their data allowed them to detect trends and patterns in it. With array technology, an experimental data set may contain several million data points, far too many for the pattern recognition machinery of the human brain to cope with. The challenges of analyzing these large scale data sets are the subject of the next chapter.

Chapter 4
Statistical Issues Associated With High Dimensional Genomic Data

4.1. Introduction

When interpreting the quantitative results of a biological experiment, it is essential to determine whether any observed effect could have occurred by chance. This class of question falls into the domain of statistics, specifically the field of hypothesis testing. Hypothesis tests seek to distinguish between the so called null and alternative hypotheses, where the null hypothesis generally refers to a case in which there is no difference between two observations. In order to perform a hypothesis test, a statistic to measure the difference between the two entities must be defined, and the distribution of that statistic under the null hypothesis must be theoretically determined or empirically calculated. The percentile value of the calculated statistic in the null distribution is defined as the p-value, which can be thought of as the likelihood that the observation could have happened simply by chance. A p-value less than 0.05, therefore, indicates that in only 1 case out of 20 a statistic of the magnitude observed or greater would occur by chance.

Different statistics are employed in different cases, but in most analyses of differences in gene expression between two classes, the t-statistic is used. This statistic is a quantitative description of the difference between two populations, and is defined as the absolute value of the difference in their means divided by their standard deviations. Consider a case with a series of mRNA gene expression values from samples that fall into two different classes (x1, x2, ..., x_M) and (y1, y2, ..., y_N). The t-statistic is calculated as follows:

$$t = \frac{|\bar{x} - \bar{y}|}{\sqrt{\frac{\sum_{i=1}^{M}(x_i - \bar{x})^2}{M} + \frac{\sum_{j=1}^{N}(y_j - \bar{y})^2}{N}}}$$

Theoretical determinations of null distributions of t in gene expression data sets are difficult, since they are susceptible to parameters such as signal to noise ratios. Also, there is no reason, *a priori*, to presume that there are no systematic experimental biases that might cause deviation from the assumed normal distribution. Furthermore, given that one is frequently computing thousands of p-values from a gene expression data set, the probability that some of the computed statistics will have extreme and nominally significant values is very high, even in the case where the null hypothesis is true.

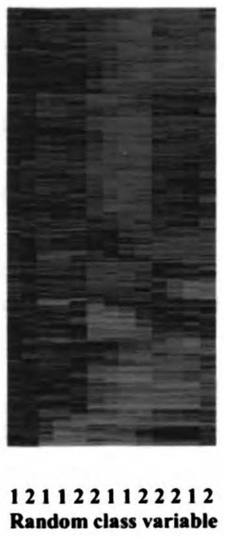
Most analyses of differences in gene expression calculate the null distribution of t empirically using permutation and resampling based approaches. These techniques attempt to simulate the null hypothesis by randomly assigning values to each class and calculating t statistics for each random shuffling of the data. Since the values are divided into groups randomly, there should be no difference in population parameters such as means or standard deviations. By performing these random reshufflings many times and calculating a value of t each time, a null distribution of t can be determined. The p-value of the non-permuted t statistic can be determined by finding its percentile in the derived null distribution.

While the standard statistical approaches work well when the dimensionality of the data is relatively small, they tend to fail on gene expression data due to the large number of measurements given the small number of independent experimental samples. This phenomenon is known as the multiple comparisons problem. The following sections discuss the multiple comparisons problem in detail and present some strategies for addressing it.

4.2. The Multiple Comparisons Problem

Microarray experiments pose particular statistical problems because of the sheer amount of data generated. A single hybridization can generate tens of thousands of measurements per sample, and when the ratio of the number of measurements to the number of samples becomes very large, false relationships between variables can emerge. Because so many statistical comparisons are made, strong correlations between variables may be observed even under the null hypothesis, simply by chance. This 'multiple comparisons problem' is one of the most challenging aspects of analyzing large biological data sets.

For example, consider a case in which random class variables are assigned to a publicly available mRNA gene expression data set, and signed t-statistics calculated for the differences in expression values between the two classes (Figure 5). Because the class assignments are made at random, the null hypothesis is in effect and none of the resulting t-statistics can be considered significant. The distribution of the statistic values is roughly normal, and because tens of thousands of statistics are calculated, some of those statistics have very large magnitudes simply by chance. Although several of the extreme values would be considered significant in other cases, here the sheer size of the data set generates false positives. Keep in mind that the definition of $p<0.05$ for a given value of the statistic is that in 5% of cases a statistic would be observed at the given value or higher just by chance. If 10,000 statistics are to be calculated at an uncorrected p-value cutoff of $p<0.05$, then approximately 5% of 10000 or 500 false positives would be expected, clearly too great a number.



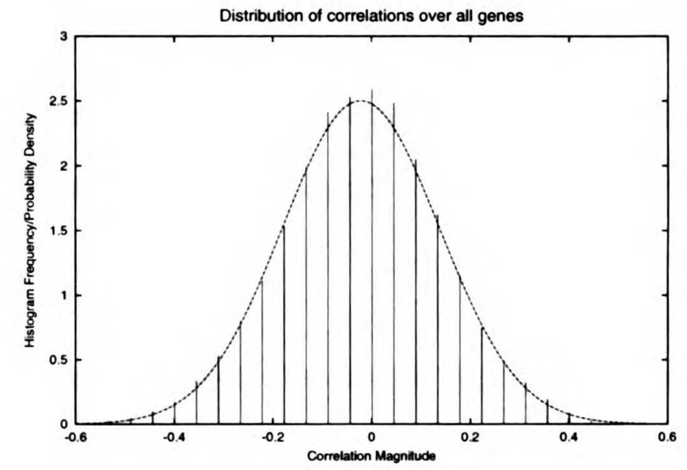


Figure 5 – Distribution of signed t-statistics using mRNA gene expression data assigned to random classes

The multiple comparisons problem is a substantial issue in analyzing genomic data, but not an insurmountable one. Several approaches have been developed to calculate corrected significance cutoffs on even very high dimensional data sets (Westfall and Young 1993; Jain, Chin et al. 2001; Tusher, Tibshirani et al. 2001; Olshen and Jain

2002; Segal, Dahlquist et al. 2003; Storey and Tibshirani 2003). Three of the more commonly used techniques will be described in the following sections: Bonferroni, false discovery rates, and maxT

4.2.1. Bonferroni Method

The Bonferroni method is conceptually the simplest technique to correct for multiple comparisons. The Bonferroni method is performed by simply dividing the p-value cutoff by the number of comparisons made. Consider a case in which a gene expression data set containing 10000 genes is to be tested for differential expression with respect to two classes. If a significance cutoff of $p<0.05$ would be used in the single gene case, then Bonferroni would adjust the p-value to 0.05/10000 or 5×10^{-6}. While this concept is easy to understand, it tends to generate significance cutoffs that are far too conservative. In most real world cases where there is genuine signal of moderate strength, no observed statistics meet these stringent cutoffs.

4.2.2. False Discovery Rates

False discovery rate (FDR) is a technique that controls the number of type I errors (false positives) when performing hypothesis tests on large data sets (Benjamini and Hochberg 1995). FDR controls the fraction of type I errors, such that it predicts the expected percent of false positives in the total set of predictions. For example, if an FDR based algorithm predicts that 100 genes are differentially expressed at a false discovery rate of 0.3, then one can expect roughly 70 of these predictions to be correct.

The significance analysis of microarrays (SAM) application is probably the most commonly used tool to determine differential expression using false discovery rates

(Tusher, Tibshirani et al. 2001). SAM uses a statistic called the 'relative difference' or d (which is similar to the t statistic) to rank genes based on differential expression across two classes. SAM then employs a permutation based approach of randomly assigning sample labels to determine the null distribution of the relative difference at each position in the list of ranked genes. For a given threshold expression difference, the FDR is taken as the fraction of permuted genes that were called significant to the total number of non-permuted genes that were called significant.

4.2.3. MaxT

The maxT method employs a resampling based approach to determine the null distribution of maximum correlation values for a particular data set (Westfall and Young 1993). For each round of permutation, sample labels are assigned at random to ensure the null hypothesis is in effect. A t-statistic is calculated for each gene, but only the maximum value is saved to generate the null distribution of the statistic. This process is repeated many times, and the value of the unpermuted statistic is compared to the distribution of maximum observed statistics. Because the distribution was generated from maximal observed statistics only, the p-value can be thought of as the likelihood that any gene in a data set of the given size would have a statistic of the observed magnitude or greater under the null. MaxT is therefore one measure of the so called family wise error rates (FWER), because it estimates the probability of at least one false positive over the collection of tests.

4.3. Correlation of CGH and mRNA Expression Data

While these techniques and others can provide a means of determine meaningful significance cutoffs in the face of multiple comparisons, there are some situations that overwhelm them as well. One of the unique opportunities made available by collaborations within the UCSF Cancer Center is access to tumor data sets that contain multiple types of genomic data, such as CGH and mRNA expression data. One of the obvious questions to ask of data sets containing both of these data types is the degree to which DNA copy number changes affect mRNA levels across the genome. For example, the proto-oncogene myc is frequently amplified in certain cancers, leading to increased myc mRNA and protein levels. Since myc is a transcription factor, however, amplifying myc should lead to changes in the expression of many other genes. One could reasonably expect, therefore, that amplification of the genomic locus containing myc would affect the transcript levels of other genes in addition to myc itself.

One approach that can be used to detect these regulatory relationships is to correlate the mRNA expression data for each gene with the amplification data from each genomic region, and select those gene-BAC pairs that have high correlation values. To perform one such correlation, two arrays of values are assembled that contain the sample to sample behavior of one gene and one BAC. The Pearson's correlation coefficient (r) of the two arrays of numbers x and y is calculated as follows:

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}}$$

Pearson's correlation coefficient is defined as the covariance of two variables, divided by their standard deviations. In practical terms, r is a measure of the linearity of a relationship between two continuous data types (except for perfectly horizontal or vertical lines, for which r is undefined). The values of r vary from -1 (perfectly linear relationship with negative slope) to 1 (perfectly linear relationship with positive slope).

Exhaustively calculating all possible correlations of genes and BACs presents a multiple comparisons problem of extreme proportions, however, since the number of comparisons performed is the product of the dimensionality of each data type. For example, if 2500 BACs are analyzed for CGH and 10,000 genes for mRNA expression, this totals 2.5×10^7 total correlations. Even when genuine correlations exist in these data sets, they are difficult to detect against the backdrop of all the spurious high correlations observed by chance.

To illustrate the magnitude of this problem, an all by all correlation of genes and BACs in an ovarian tumor data set was performed, correcting for multiple comparisons using the maxT method. No gene-BAC pair showed a significant correlation, even though it has been established that gene copy number affects transcript levels for a significant proportion of genes in the amplified or deleted regions.

Clearly, standard approaches do not fare well when faced with data sets of this scale. This failure of existing methods led us to try alternative methods that lie outside of the field of statistics. Our first attempt at finding quantitative relationships between CGH and mRNA gene expression involved the use of techniques from machine learning and optimization. This attempt is the topic of the next chapter.

Chapter 5
Optimization Methods to Determine the Relationship between DNA Copy Number and Gene Expression

5.1. Introduction

As stated in the previous section, the multiple comparisons problem often makes it difficult to find all but very strong signals in high dimensional datasets. This is particularly true when two genomic data types, such as mRNA gene expression and DNA copy number are to be exhaustively compared, as in the case of the ovarian tumor data set mentioned in the previous chapter. This data set first became available before the completion of the human genome sequence, so precise genomic locations were frequently not known for many BACs and genes. At the time, the genomic location of some BACs could be inferred from the presence of STS markers whose location was experimentally determined (at fairly low resolution) by techniques such as FISH or radiation hybrid mapping, while the genomic position of many genes was unknown.

The existing statistical methods failed to find significant correlations in the ovarian cancer dataset, but because there was significant interest in the genome-wide relationship between DNA copy number and gene expression, we pursued the question without the genomic mapping information. Lacking this information, however, required an indirect approach to establish the relationship. Instead of looking for direct correlations between variations in DNA copy number at particular loci with variation in the mRNA expression of particular genes, we considered the extent to which the inferred 'distances' between samples based on CGH data and the 'distance' based on expression data were related. Intuitively, this corresponds to the hypothesis that if there were a very strong relationship between the two signals, independent hierarchical clusterings of the samples based on the two data types would yield similar structures. Recall from Figure 4 that the dominant signal in expression data can be traced to tissue of origin or specific cell type that gives rise to a particular tumor. It turns out that this is not the case with DNA copy number data. So, direct comparison of the distance matrix among tumors from CGH data to the distance matrix from expression data does not yield an obvious and significant signal.

Because the existing statistical methods failed to find significant correlations in the ovarian cancer dataset, two machine learning and optimization approaches were used to select gene-BAC pairs that would enrich the relationship by reducing the effects of measurements that did not contribute to the signal. These approaches were a hill climbing method to select, in a binary sense, the genes and BACs to participate, and a gradient descent method to derive real-valued weights for the genes and BACs. Both techniques start with the entire mRNA expression and CGH data sets, and attempt to enrich for genes and BACs that correlate with each other in a way that reveals itself by increasing the similarity in the distance matrices based on mRNA expression and CGH.

In both approaches described here, an ovarian tumor data set containing CGH and mRNA expression was separated into 'training' and 'test' sets of equal size. The training set was used to train parameters (such as which genes and BACs are selected, or the values of weights). The test set was then used to confirm that the observation was not the result of overfitting.

In both the hilllclimbing and gradient descent approaches, two arrays of sample to sample distances are calculated, one using CGH data and one using mRNA expression data. These arrays contain all pairwise sample to sample distances in mRNA expression space and CGH space, respectively:

$D_{CGH} = [D_{(1)(2)CGH}, D_{(1)(3)CGH}, \ldots D_{(n-1)(n)CGH}]$

$D_{Exp} = [D_{(1)(2)Exp}, D_{(1)(3)Exp}, \ldots D_{(n-1)(n)Exp}]$

Where $D_{(1)(2)CGH}$ = Distance from sample 1 to sample 2 using CGH data.

An attempt is then made to minimize the distance between D_{CGH} and D_{Exp} by either choosing different subsets of genes and BACs (hill climbing) to calculate the sample to sample distances, or by assigning weights to each gene and BAC and allowing those weights to fluctuate (gradient descent). The rationale for this approach is that is if CGH loci and genes that correlate with each other are chosen, then distance vectors D_{CGH} and D_{Exp} will correlate with each other. Conversely, if the correlation distance between D_{CGH} and D_{Exp} is minimized, then genes and BACs that correlate with each other should be selected for. If this result bears out in a clean validation test set, then a large-scale relationship between DNA copy number and RNA expression has been shown, albeit somewhat indirectly.

5.2. Hill Climbing

Hill climbing is a so called 'greedy' method that is quite simple conceptually; first, a small number of genes and BACs are chosen from the total (100 of each in this case) and the distance arrays D_{CGH} and D_{Exp} are calculated using the data from these subsets of genes and BACs. With each iteration of hill climbing, a small proportion of the genes and BACs are randomly replaced, and the sample to sample distance arrays D_{CGH} and D_{Exp} are recalculated. If the distance between D_{CGH} and D_{Exp} is lower with the new set, then that set is kept for subsequent rounds, otherwise the new set is discarded and the old set is kept (hence 'greedy'). This process is repeated many times, and the final set of selected genes and BACs is subjected to further examination.

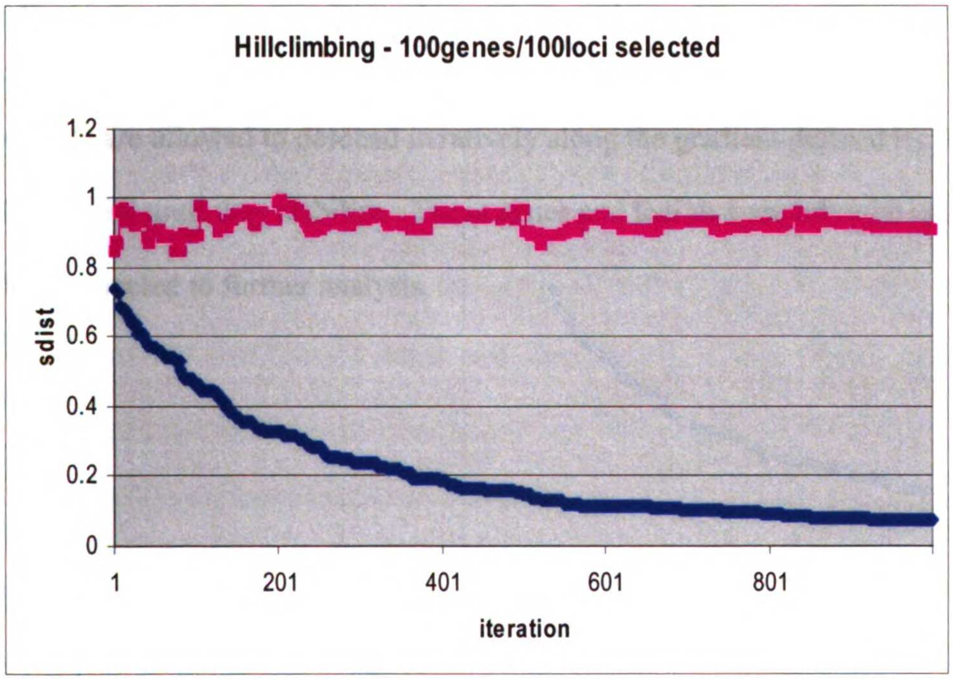


Figure 6 – The effect of hillclimbing on the training set (blue) and the test set (magenta).

The effect of hillclimbing on the training and test sets is shown in Figure 6. While the distance between D_{CGH} and D_{Exp} decreases using the training set data (computed as 1 – Pearson's correlation), it is unaffected by when using the test set. This clearly indicates that the hill climbing approach is subject to overtraining on noise in the training set. This suggests that it is easy to identify a small number of genes and BACs that yield a nominally good distance matrix correlation, but that it does not generalize in a meaningful way.

5.3. Gradient Descent

Gradient descent methods define an error function (such as the mean squared error E in Figure 7) to be minimized by adjusting weight values (w_k, for the kth gene or BAC in the data set). Once again, two arrays of sample to sample distances are separately calculated using CGH and mRNA expression data (D^{CGH} and D^{Exp}, respectively). The error function is differentiated with respect to each weight, and the weights are allowed to descend iteratively along the gradient defined by the value of this differential until they stabilize. Those genes and loci that are left with large weights are then subjected to further analysis.

$$E = \sum_{ij} (D_{ij}^{CGH} - D_{ij}^{Exp})^2$$

$$D_{ij}^{CGH} = \frac{\sum_k w_k^{CGH} e^{\frac{-(CGH_i - CGH_j)^2}{\sigma_{CGH}}}}{\sum_k w_k^{CGH}} \qquad D_{ij}^{Exp} = \frac{\sum_k w_k^{Exp} e^{\frac{-(Exp_i - Exp_j)^2}{\sigma_{Exp}}}}{\sum_k w_k^{Exp}}$$

$$\frac{\partial E}{\partial w_k^{CGH}} = \frac{2}{\sum w_k^{CGH}} \sum_{ij} (D_{ij}^{CGH} - D_{ij}^{Exp})(e^{-(CGH_i - CGH_j)/\sigma_{CGH}} - D_{ij}^{CGH})$$

$$\frac{\partial E}{\partial w_k^{Exp}} = \frac{2}{\sum w_k^{Exp}} \sum_{ij} (D_{ij}^{Exp} - D_{ij}^{CGH})(e^{-(Exp_i - Exp_j)/\sigma_{Exp}} - D_{ij}^{Exp})$$

$$w_k = w_k + \alpha \frac{\partial E}{\partial w_k}$$

Figure 7 – The mean squared error and distance functions used in the gradient descent method

The effect of the gradient descent method was assessed using a cross validation measure; a single tumor sample is held out and the remaining samples are used to train the system by calculating weights. The correlation between the distance arrays of the holdout to the remaining samples are compared before and after training and the process is repeated with a different holdout each time. Figure 8 shows the results of a 20-fold cross-validation, and in each case the correlation distances between the CGH and mRNA gene expression distance vectors were lower after training than before, evidenced by the lower right triangular enrichment of points. This indicates that training did have a positive influence on the correlation between the two distance vectors. Although the effect was not large, it was consistent, and it demonstrated a relationship between DNA copy number and gene expression.

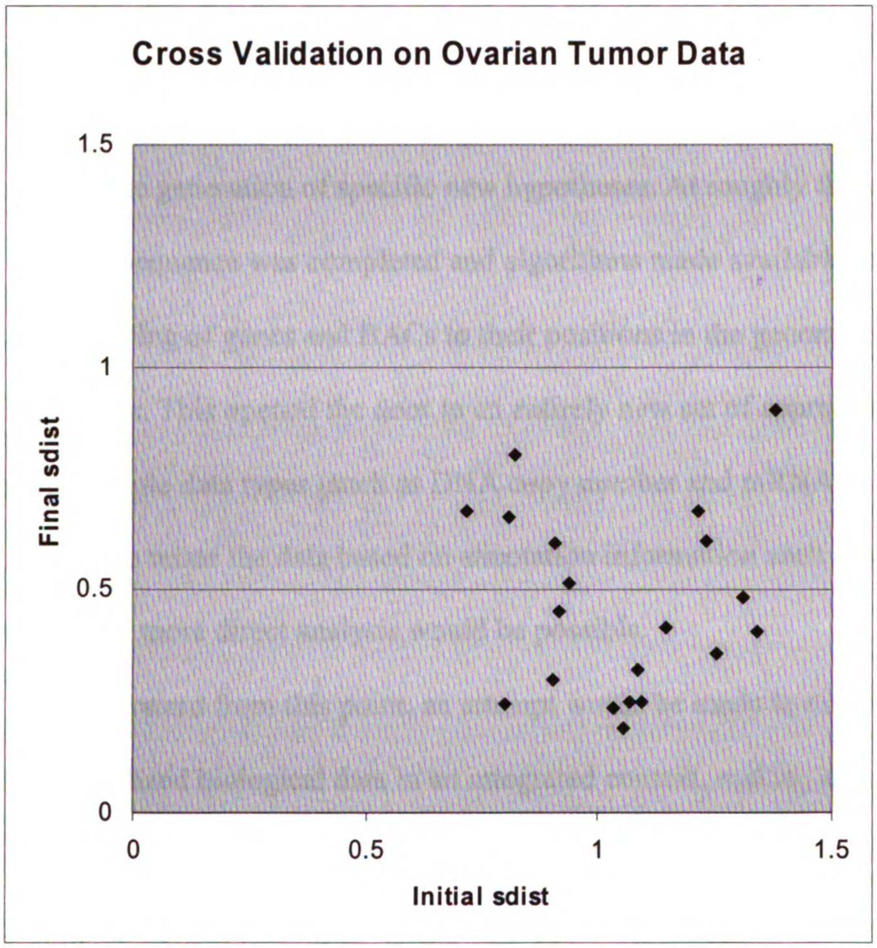


Figure 8 – Cross validation using the gradient descent method

5.4. Conclusions

Two approaches were taken to detect correlations between CGH and mRNA expression data, with limited success. The hill climbing method showed evidence of overtraining and did not yield usable results. In contrast, the gradient descent method showed some indication of success on the test set, although the effect was modest. Interpreting the results of the output is not straightforward, and the fact that these methods are both difficult for most biologists to understand and use raise questions as to their utility.

At this juncture in my thesis work, the analytical methods that had been used to investigate the relationship between CGH and mRNA expression data had generated a tangible result in the sense that a relationship was demonstrated. However, the approach did not lead to the generation of specific new hypotheses. At roughly this time, the human genome sequence was completed and algorithms made available for high-throughput mapping of genes and BACs to their positions in the genome based directly on their sequence. This opened the door to an entirely new set of approaches. If, in the analysis of multiple data types (such as DNA copy number and mRNA gene expression), it was possible to relate the data based on annotation information such as genomic position, a much more direct analysis would be possible.

Going forward from this point, an attempt would be made to explore analytical methods that utilized biological data in an integrated context, making extensive use of biological annotation information. However, integration of annotations for tens of thousands of measurements over multiple data types and utilization of these annotations in statistical computations is challenging, and certainly beyond the skill set of most experimentalists. Consequently, these methods would be deployed within a straightforward and intuitive system that experimentalists themselves could use. The development of this system, Magellan, is the subject of the next chapter.

Chapter 6
Magellan: System Design

6.1. Introduction

The previous two chapters demonstrated how both standard statistical techniques and sophisticated optimization approaches can fail to find significant relationships in very high dimensional biological data sets or yield results with limited interpretability. Since one of the aims of my work at the Cancer Center was to provide analytical tools for experimentalists to use, the next portion of my thesis work was devoted to building an analytical framework that would allow for a more intuitive approach to analyzing biological data sets.

With this aim in mind, I developed a server based system that allows biologists to perform analyses over the internet. Rather than creating a series of individual applications that perform very specific functionalities, it was decided that our analytical system would implement a framework such that different analytical methods could be interfaced to the system as a whole. The purpose of such a system is to allow end users to explore their own data sets and follow whatever direction of inquiry that they see fit.

Because of the exploratory nature of the system, it was named Magellan after the 16th century explorer.

This chapter elucidates the reasoning behind several of the major design decisions of Magellan, while the next chapter covers implementation issues. This chapter discusses the various types of information that can be stored and analyzed, and the high level representation of information.

6.2. Definitions and Examples of Terms

The terms 'data', 'variables', 'identifiers', and 'annotations' have been used to this point without formal definitions, but it is important to clarify their meaning. The following sections define each of these terms as they are used in the context of the Magellan application.

6.2.1. DataType

For the purposes of this discussion, a 'data type' is defined as a category of qualitative or quantitative information gathered from a biological sample. In the use cases that follow, data types include mRNA expression intensity measurements, genomic copy number measurements, and clinical data such as patient survival.

6.2.2. Variable

A 'variable' is defined as a single measurement of a multivariate data type. Variables would include mRNA intensity measurements for individual genes in mRNA expression data, or genomic copy number measurements for individual genomic regions from CGH data.

6.2.3. Identifier

An 'identifier' is a string that names one individual variable of a data type. Examples of an identifiers would include genbank ID's, LocusLink ID's, and Affymetrix gene chip ID's.

6.2.4. Annotation

An 'annotation' is defined as a quantitative or qualitative description of the variables of a data type. One of the key distinctions of the Magellan application is the ability to utilize biological annotations, both in analytical methods and in data pre-processing. As they are used within Magellan, annotations can be divided into two classes: curated and derived.

Curated annotations can consist of numerical information such as genomic mapping data for genes, textual information derived from ontologies of gene function, and formal descriptions of regulatory networks (Ashburner, Ball et al. 2000). Data and curated annotations are linked together through the use of 'identifiers', which are user-defined names such as Genbank or RefSeq Ids (Figure 9).

Derived annotations are *computed* from the experimental measurements that comprise a data set. These annotations are the results of a computation performed on the data and are, therefore, derived from that data. Derived annotations are typically quantitative and are linked to data directly rather than through identifiers (Figure 9). Direct linkage is required since individual identifiers (representing one gene on a microarray, for example) can be represented multiple times in a data set, with different quantitative measurements in each physical instance yielding different annotation values.

For the analyses performed here, derived annotations of correlation to patient outcome for gene expression and frequency of alteration of genomic copy number were used.

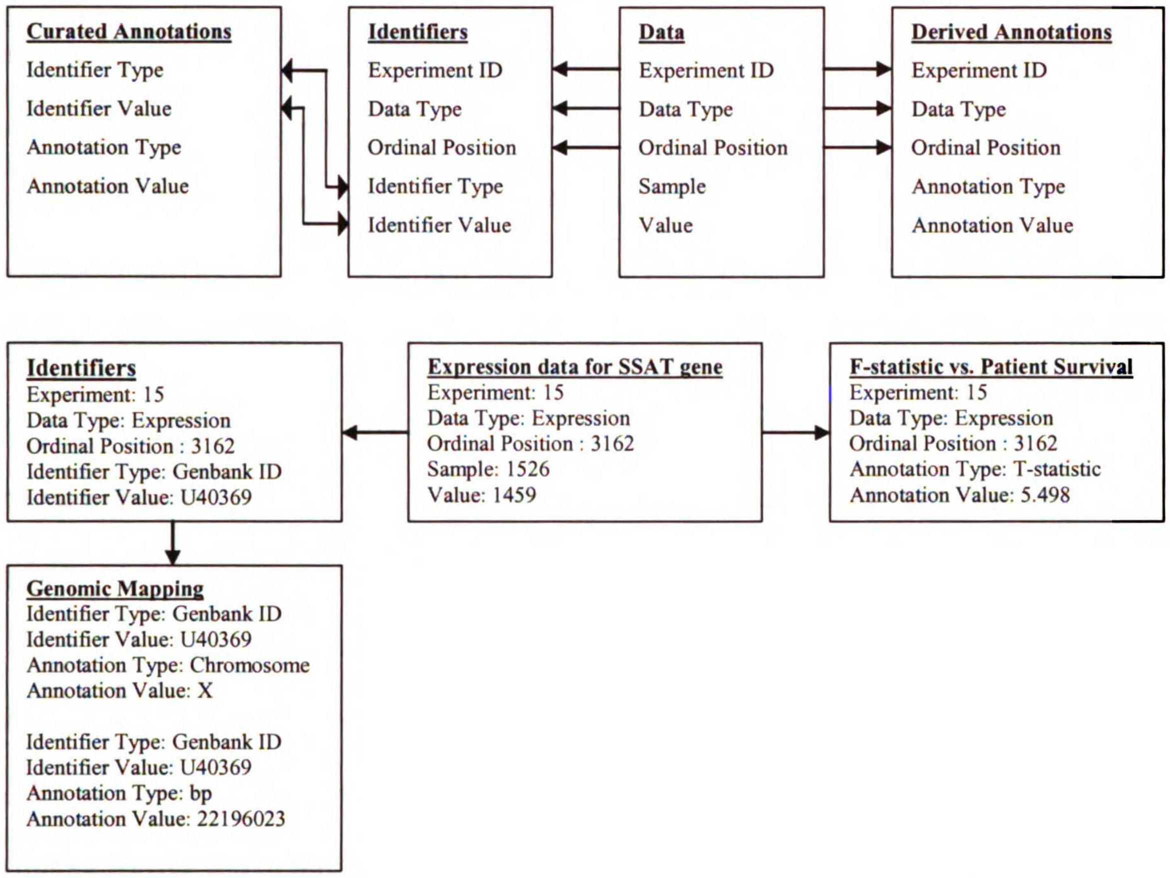


Figure 9 - Linking data to curated and derived annotation information. Variables of a data type (such as genes or genomic loci) are referenced by their ordinal position. The ordinal position can be directly linked to derived statistics (such as t-test against patient survival) or to named identifiers (such as a Genbank IDs). Identifiers are then used to link data to curated annotations (such as genomic mapping information).

6.3. Abstract Representation of Data and Annotations

The primary feature that enables Magellan to be used as a general purpose analytical tool is its ability to store information abstractly. Under this scenario, there is no assumption made as to the type of data or annotations that a user stores in the system. Rather than being specifically designed for microarray gene expression, CGH, proteomics, or some other data type, Magellan's abstracted means of storing data is designed to accommodate these and other data types as they arise. Rather than requiring particular data types, Magellan stores the data and annotation type designation that the user defines. In other words, the system does not tell users what kind of data they can store; the end users specify what kind of information they wish to store.

The only requirement placed on information to be stored in Magellan is that it can be represented as a two dimensional table. In the case of data, such an abstracted table would represent a data type with *m* samples and *n* variables collected per sample. In the case of annotations, the abstracted table would contain *m* annotation types (such as chromosome and distance from p-telomere designations that collectively make up genomic position annotations) and *n* variables to be annotated.

While this arrangement does provide extensibility such that new data types can be stored as they emerge, it does not have the advantage of some custom databases that are designed to work with particular data types. Such custom databases generally do a more comprehensive job of representing data and metadata for the data type for which they were designed, but have to be extensively modified to accommodate other data types. For our purposes, a decision was made to make Magellan an extensible system because of the heterogeneity of data currently available, and the strong likelihood that new data types would emerge that would benefit from Magellan's analytical framework.

While the storage and representation aspects of Magellan are generalized, the analytical functions deployed within the system require varying degrees of specificity with respect to the types of information that they utilize. While some analytical methods may be generalized enough to work with large numbers of data types, others will require specific types and representations of data and annotations. For example, resampling-adjusted correlation analysis can be applied to many different tabular quantitative data sets, while methods that seek to explore the relationship between CGH and mRNA expression data with respect to genomic position annotations are very specific in their requirements. Such specificity is allowed within the Magellan application, but only at the level of the analytical method. In summary, the Magellan database can store heterogeneous data and annotations in an abstracted fashion, but analyze this information along a continuum of specificities depending on the application chosen.

6.4. Use of Biological Annotations within Magellan

Given a series of tumor samples of known outcome, with experimental data comprising both DNA copy number and mRNA expression measurements, natural questions tend to span data types or require data annotation. Does genomic copy number directly account for some of the variation in gene expression across samples? Are the genes that map to loci that are frequently found to be of aberrant copy number more likely to show an association with outcome than other genes? Are genes that have functional annotations for processes involved in cancer (for example, adhesion, apoptosis, invasion, ...) more likely to be associated with tumor aggressiveness?

Because of the importance of biological annotations in answering these types of questions, Magellan has been designed to store and retrieve annotations for analyses that utilize them. In addition to the use of annotation information during analysis, Magellan provides two means of utilizing annotations prior to analysis: variable selection and variable projection.

6.4.1. Variable Selection

As has been previously discussed, a large number of variables collected from a relatively small number of biological samples can present statistical challenges. One particularly straightforward method to reduce dimensionality is to select only those variables (such as genes from mRNA expression data or genomic loci from CGH data) that meet criteria that are orthogonal to the property being investigated. Biological annotations can provide such a means of variable selection. By restricting a data set to only those variables whose annotations meet certain criteria, the dimensionality of the data set may be reduced such that multiple comparisons do not predominate.

In addition to statistical considerations, experimentalists are frequently interested in focusing on variable subsets of biological interest. For example, a user may wish to focus on a specific gene family that is described by a biological pathway designation such as 'G protein coupled receptor' or 'cyclin dependent kinase'. By providing a generalized means of selecting variables based upon categorical textual annotations, Magellan allows end users to select for virtually any text based annotation they choose.

Magellan also allows users to select variables based on quantitative biological annotations. By specifying the annotation to be used and the quantitative operator to apply, a user can perform any number of variable selections, such as retrieval of all BACs in a CGH data set that strongly correlate with a clinical outcome. By combining variable selections, a user can create compound queries such as choosing genes in a particular genomic region, which involves a text based selection for the chromosome name followed by a quantitative selection for the distance from the p-telomere.

6.4.2. Variable Projection

The term *projection* is used to describe the process of finding 'equivalent' variables between two data types by examining the relationship between the annotations associated with those variables. There are many cases in which experimentalists would use projection to look for effects across data types or data sets. For example, if a user has identified an interesting subset of genes in mRNA expression data and wishes to find an equivalent set BACs from CGH data, projection would involve finding BACs that are in close genomic proximity to the genes in question. Alternatively, an end user may wish to determine if a biological effect is consistently observed over multiple data sets by projecting variables that meet certain criteria in one experimental data set onto a second data set.

Projection can be used to find equivalent variables over a number of different variable equivalencies, both quantitative and qualitative. In order to project variables between data types, an end user must specify which annotations to use and what constitutes equivalency. In the case of projection over identifier names, projection would use a simple string equivalency over that namespace. In the case of projection over genomic position, the equivalence criteria would include both string equivalence of chromosome name and a difference threshold for the distance from the p-telomere.

Variable projection is performed by comparing annotations associated with the variables of two data types. By moving from variables, through identifiers, into a common annotation space, comparisons can be performed that are not possible to make on the variables themselves. In the case of projection from CGH space to gene space, for example, there is no direct way to compare genes and BACs. By projecting these individual variables into a common annotation space of genomic position, however, these comparisons can be made, and user defined criteria of equivalence can be established.

6.5. Other Analytical Applications

Biological data analysis has been the subject of intense research in many groups, and there are a number of systems available that are geared toward a similar user community. These range from single-use packages (such as clustering) to integrated packages for performing multiple types of analysis (such as univariate analyses and classification for multiple data types). Examples of the former include methods such as SAM and PAM, which address permutation-corrected statistics and pattern classification, respectively (Tusher, Tibshirani et al. 2001; Tibshirani, Hastie et al. 2002), and Cluster and TreeView (Eisen), which address clustering. Magellan differs from these in that it is a platform for offering multiple types of analysis. Examples of the latter include MeV (TIGR), caWorkBench (Columbia Genome Center), GeneCluster (Broad Institute), GeneSifter (VizX Labs LLC), and mAdb (NCI).

In terms of functionality, Magellan's primary distinguishing characteristics are its generality and its use of biological annotation information as a means of constraining analyses to variable subsets. Magellan is general in two respects. First, the internal schema for storing information supports any type of data that can exist in a table (either numerical or textual). Second, Magellan does not impose complex format requirements on data, which is frequently a hurdle in making use of other systems, where local procedures for data preparation may be at variance with expectations and requirements for data formatting.

6.6. Conclusions

Magellan was designed with generality and extensibility in mind, such that many data and annotation types can be stored and analyzed. As such, Magellan employs an abstract representation of both data and annotations. While the generality of the Magellan database means that certain information will not be represented in as much detail as a specialized database, this is a reasonable tradeoff for versatility. Magellan is designed to make extensive use of annotation information, which describes the variables of data types. The specificity of Magellan is contained at the level of a subset of analytical methods which require certain data and annotation types.

Chapter 7

Magellan: System Implementation

7.1. Introduction

The previous chapter detailed the design features of Magellan in general terms. This chapter describes the details of Magellan's implementation. The following sections cover the specific software design decisions that were made in the implementation of Magellan.

7.2. Magellan System Architecture

Magellan is composed of several different components. Figure 10 is a Unified Modeling Language (UML) diagram that shows the various components of the Magellan system, including the presentation logic, business logic, database layer, and application layer. All of these components will be discussed in the following sections, while connectivity to the data sources at the National Cancer Institute will be covered in Chapter 9.

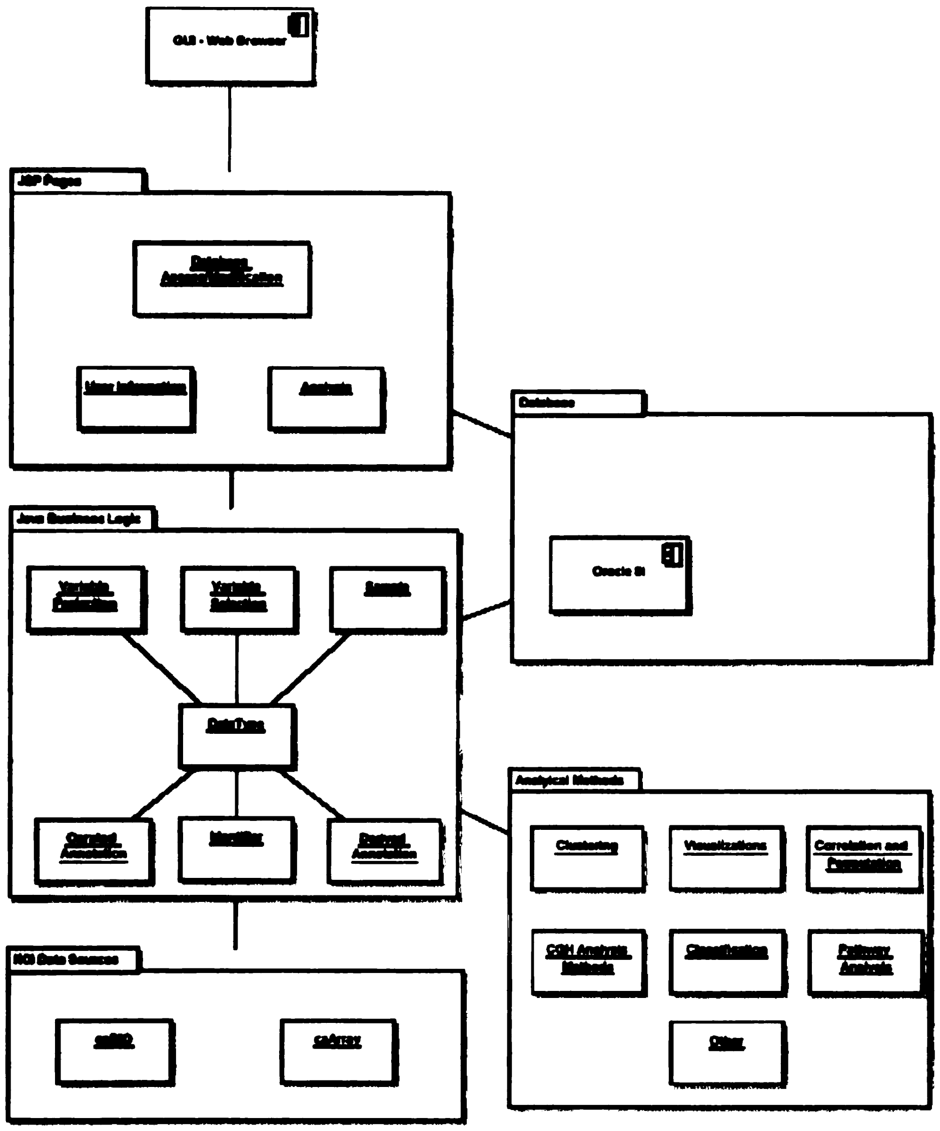


Figure 10 – UML component diagram of the Magellan system.

7.2.1. Server Sided Application Using JSP Technology

It was decided early on in the development process that Magellan should be a server sided application. As such, Magellan is installed on a centralized server that can be accessed by end users over the internet. The choice of a server side web application was made for several reasons: First, a server sided application overcomes many of the compatibility issues associated with the many different desktop platforms (PC, Mac, Unix) currently used by researchers, since interaction with the system is achieved through a web browser. Server sided applications typically have a lower barrier of entry for end users, since little or no software has to be installed on their own client computers. The centralized nature of server sided applications allows for all user data and annotation information to be stored in one location, rather than scattered over many decentralized computers. Updates to the application are also much more easily managed, since they need only be installed on a small number of servers, rather than a large number of clients.

After building prototype systems using Active Server Pages and Perl CGI, Java Server Pages (JSP) technology (http://java.sun.com/) was chosen for the development of Magellan. The primary reason for this choice is that JSP allows for the creation of server side applications using the full power of the Java programming language. Java has the advantage of being a true object oriented language, which allows for efficient code development and reuse. In moderate to large sized projects, Java provides a straightforward means of dividing the different pieces of a large application into a series of separate Java Classes. The functionality of these Java classes can be accessed through abstracted application programming interfaces (API's), which allow for easy utilization of the underlying programmatic logic without knowledge of their implementation.

Magellan is designed to be deployed on a server that is running the Windows 2000 or XP Operating System (http://www.microsoft.com/), and the Apache Tomcat web server and servlet container (http://tomcat.apache.org/). In addition, the Java component of Magellan was developed using the Eclipse IDE (http://www.eclipse.org/) and the Sysdeo Tomcat plugin (http://www.sysdeo.com/eclipse/tomcatplugin).

7.2.2. Centralized Oracle Database Utilizing an Open Architecture

All information uploaded by the end user to Magellan is stored in a relational database rather than in flat text files. While incorporation of a database component added complexity to the project, storage in a relational database provides for sophisticated querying of data and annotations using Structured Query Language (SQL). Magellan was prototyped on the Oracle database platform (http://www.oracle.com/index.html), although a port to the open source database MySQL (http://www.mysql.com/) has been initiated with collaborators at UPENN.

The use of a server sided Oracle relational database allows for centralization of all data and annotation information into one physical location. This allows for easy sharing of information between different users, with a security layer to ensure that the owner of said information has granted permission for access.

Because the data types generated by biological researchers are heterogeneous and likely to change over time, it was decided to represent both data and annotations as generally as possible. Rather than hard coding the database schema to store only certain data types, data and annotations are represented as type-value pairs. As seen in a simplified representation of the Magellan database schema (Figure 11), separate tables are utilized to store data types and annotation types as user defined strings. This method of representation is analogous to the Entity-Value-Relationship model, which also employs type-value representation of data.

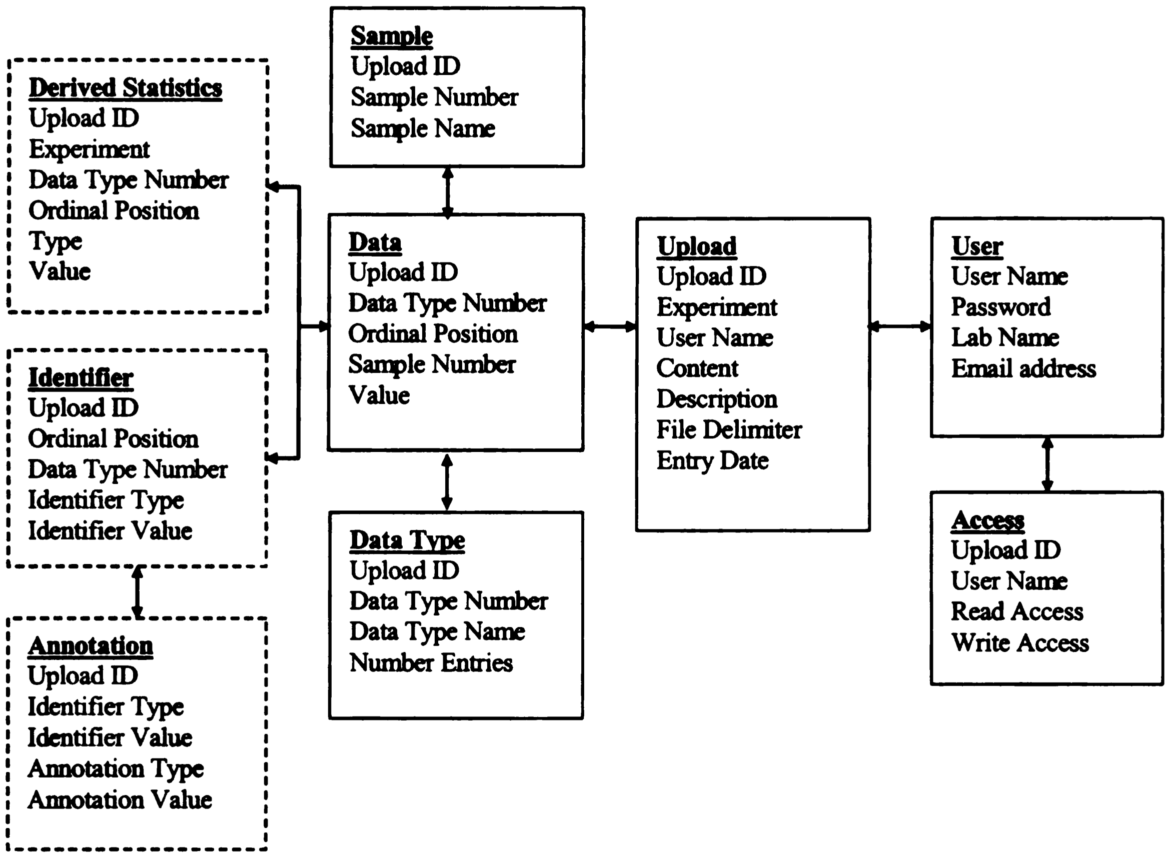


Figure 11 – A simplified representation of the table structure of the Magellan database schema. Tables with hatched outlines are used to store annotation information.

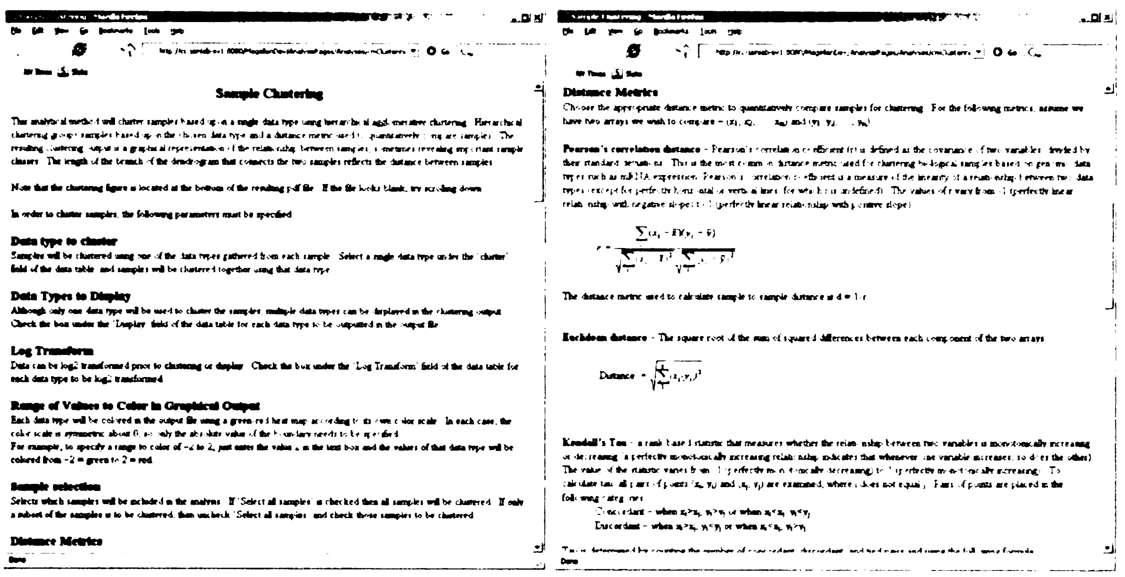
7.2.3. Analytical Applications

Although Java is a very powerful language for building server sided applications, it is not necessarily the best solution for developing the analytical applications that perform computations on data once it has been retrieved from the database. Several of the analytical applications, particularly those that make use of resampling and permutation, are extremely computationally intensive and are best suited for development using the C programming language. Compiled C programs tend to run faster than their Java based counterparts by a factor of at least two or three, a difference that becomes particularly significant with large genomic data sets. Other applications deployed in Magellan that make extensive use of statistical functions and/or graphical output have been developed using the R statistical package (http://www.r-project.org/). R is an open source application used extensively by the statistics community, and is useful because its many publicly available code libraries allow for rapid development and deployment of algorithms that make heavy use of statistical functions

External applications written in C or Java are executed from within the JSP pages by making use of Java's Runtime.exec() method. This method executes the appropriate analytical program on the server, and monitors the termination of the process. If the process terminates without error, then the user is provided with a hyperlink to all result files. If the process terminates abnormally, then the user is provided with a list of the program output and error messages, which can be provided to the system administrator for remedy.

7.3. Documentation

In an application such as Magellan, it is important that the analytical functions not be regarded as a black box by end users. To this end, several web pages have been developed to document both the core functionality of the system as well as the analytical functions deployed within. This documentation serves to explain the basic purpose of each analysis method, descriptions of the parameters that the user must specify, and definitions of statistics and distance metrics used by the algorithm. In addition, the documentation for analytical functions includes an example use case and output.



Sample Clustering

This analytical method will cluster samples based upon a single data type using hierarchical agglomerative clustering. Hierarchical clustering groups samples based upon the chosen data type and a distance metric used to quantitatively compare samples. The resulting clustering output is a graphical representation of the relationship between samples, sometimes revealing important sample classes. The length of the branch of the dendrogram that connects the two samples reflects the distance between samples.

Note that the clustering figure is located at the bottom of the resulting pdf file. If the file looks blank, try scrolling down.

In order to cluster samples, the following parameters must be specified:

Data type to cluster
Samples will be clustered using one of the data types gathered from each sample. Select a single data type under the 'cluster' field of the data table, and samples will be clustered together using that data type.

Data Types to Display
Although only one data type will be used to cluster the samples, multiple data types can be displayed in the clustering output. Check the box under the 'Display' field of the data table for each data type to be outputted in the output file.

Log Transform
Data can be log2 transformed prior to clustering or display. Check the box under the 'Log Transform' field of the data table for each data type to be log2 transformed.

Range of Values to Color in Graphical Output
Each data type will be colored in the output file using a green-red heat map according to its own color scale. In each case, the color scale is symmetric about 0, so only the absolute value of the boundary needs to be specified.
For example, to specify a range to color of -2 to 2, just enter the value 2 in the text box and the values of that data type will be colored from -2 = green to 2 = red.

Sample selection
Selects which samples will be included in the analysis. If 'Select all samples' is checked then all samples will be clustered. If only a subset of the samples is to be clustered, then uncheck 'Select all samples' and check those samples to be clustered.

Distance Metrics

Distance Metrics
Choose the appropriate distance metric to quantitatively compare samples for clustering. For the following metrics, assume we have two arrays we wish to compare – $(x_1, x_2, \ldots x_n)$ and $(y_1, y_2, \ldots y_n)$.

Pearson's correlation distance – Pearson's correlation coefficient (r) is defined as the covariance of two variables divided by their standard deviations. This is the most common distance metric used for clustering biological samples based on genomic data types such as mRNA expression. Pearson's correlation coefficient is a measure of the linearity of a relationship between two data types (except for perfectly horizontal or vertical lines, for which r is undefined). The values of r vary from -1 (perfectly linear relationship with negative slope) to 1 (perfectly linear relationship with positive slope).

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2}\sqrt{\sum (y_i - \bar{y})^2}}$$

The distance metric used to calculate sample to sample distance is d = 1-r

Euclidean distance – The square root of the sum of squared differences between each component of the two arrays.

$$\text{Distance} = \sqrt{\sum (x_i - y_i)^2}$$

Kendall's Tau – a rank based statistic that measures whether the relationship between two variables is monotonically increasing or decreasing (a perfectly monotonically increasing relationship indicates that whenever one variable increases, so does the other). The value of the statistic varies from -1 (perfectly monotonically decreasing) to 1 (perfectly monotonically increasing). To calculate tau, all pairs of points (x_i, y_i) and (x_j, y_j) are examined, where i does not equal j. Pairs of points are placed in the following categories:

Concordant – when $x_i > x_j$, $y_i > y_j$ or when $x_i < x_j$, $y_i < y_j$
Discordant – when $x_i > x_j$, $y_i < y_j$ or when $x_i < x_j$, $y_i > y_j$

Tau is determined by counting the number of concordant, discordant, and tied pairs and using the following formula

Figure 12 – Example documentation of analytical methods in Magellan

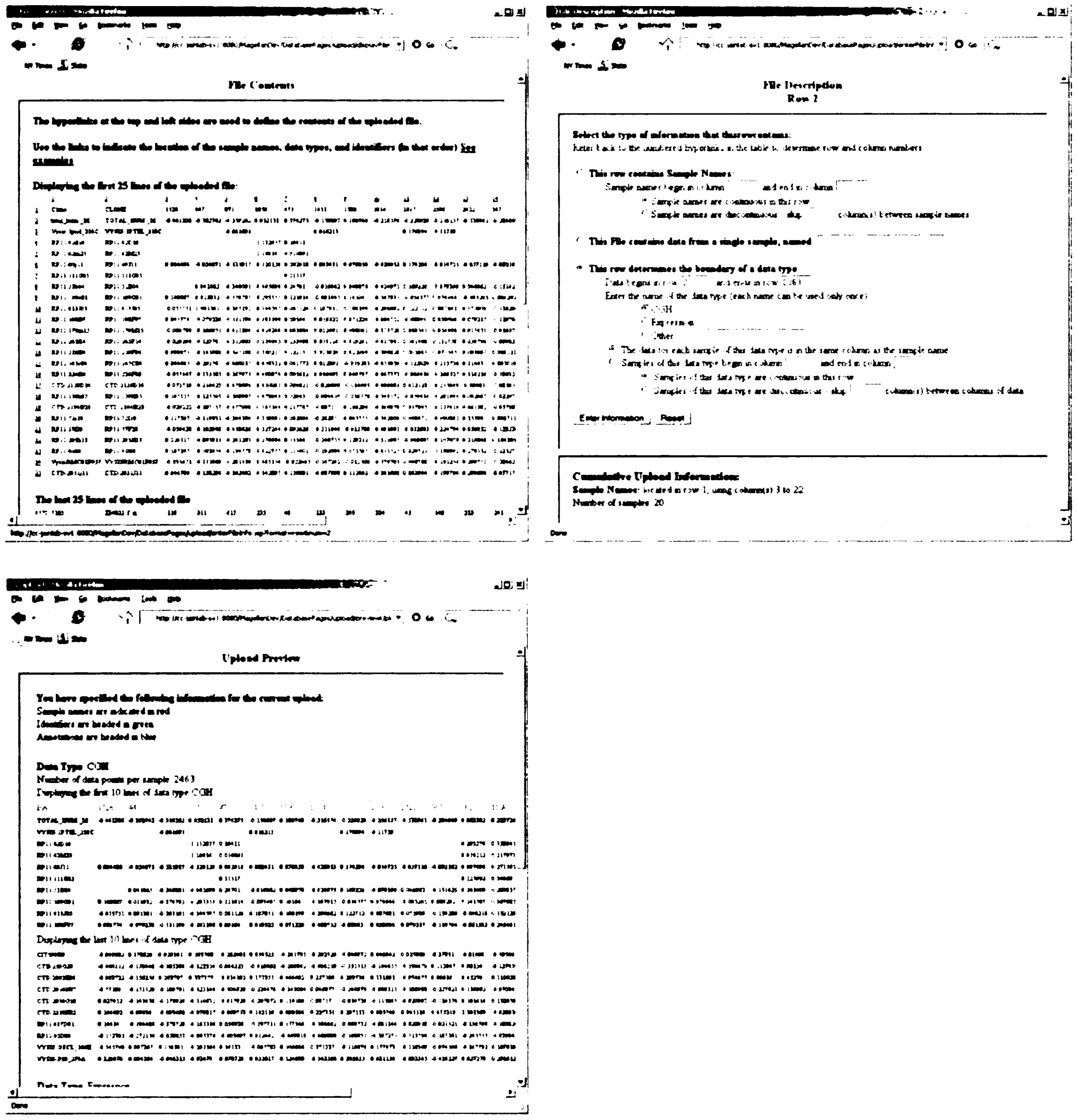
7.4. System Security

All data and annotations stored in Magellan are password protected. Each user chooses a login and password when they register to use Magellan online and must enter the login and password each time they use the system. All information uploaded by a user belongs to that individual, and is only accessible by other users with the permission of the researcher who uploaded it. Magellan was not designed to store any identifiable patient data, so additional system security measures besides the password protection described above have not been employed.

7.5. User Interfaces to Magellan Features

7.5.1. *File Format Independent Method of Data Entry*

One of the difficulties in using existing analytical applications is their dependence on particular file formats. For this reason, it was decided that Magellan would provide a generalized means of uploading data from the user to the system. Rather than forcing end users to reformat their data, the user can specify the content and location of the information to be stored. This is achieved by generating a representation of the uploaded data file in the browser complete with hyperlinks along the rows and columns. When a particular hyperlink is activated, a web page pops up that contains fields into which a user can specify the content contained within that region of the file (Figure 13). By repeating the process for the sample names, data types, identifiers and annotations, the user tells the system exactly what information to extract from the uploaded file, and how it should be represented in the database.


File Contents

The hyperlinks at the top and left sides are used to define the contents of the uploaded file.

Use the links to indicate the location of the sample names, data types, and identifiers (in that order) See examples

Displaying the first 25 lines of the uploaded file:

[illegible]

The last 25 lines of the uploaded file

[illegible]

File Description
Row 2

Select the type of information that this row contains:
Refer back to the numbered hyperlinks in the table to determine row and column numbers

This row contains Sample Names:
Sample names begin in column and end in column
Sample names are continuous in this row
Sample names are discontinuous - skip column(s) between sample names

This File contains data from a single sample, named

This row determines the boundary of a data type
Data begins in row 2 and ends in row 2463
Enter the name of the data type (each name can be used only once)
CGH
Expression
Other
The data for each sample of this data type is in the same column as the sample name
Samples of this data type begin in column and end in column
Samples of this data type are continuous in this row
Samples of this data type are discontinuous - skip column(s) between columns of data

Enter Information Reset

Cumulative Upload Information:
Sample Names: located in row 1, using column(s) 3 to 22
Number of samples 20

Upload Preview

You have specified the following information for the current upload:
Sample names are indicated in red
Identifiers are headed in green
Annotations are headed in blue

Data Type CGH
Number of data points per sample 2463
Displaying the first 10 lines of data type CGH

[illegible]

Displaying the last 10 lines of data type CGH

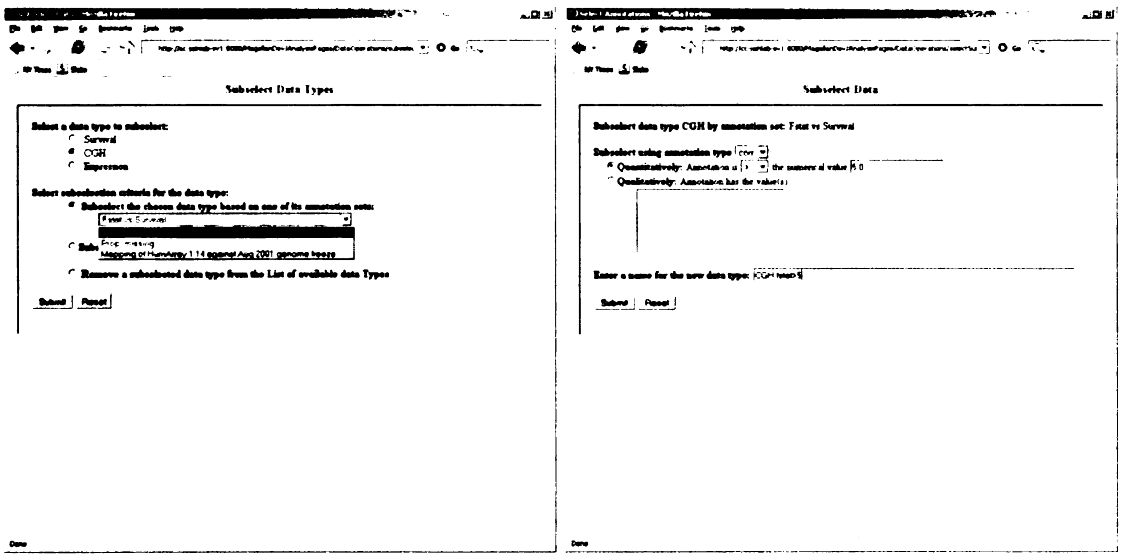
[illegible]

Figure 13 – Web pages used to upload unformatted biological data to the Magellan database. The file contents, content specification, and upload preview pages are shown.

7.5.2. Variable Selection

The user interface used to perform a variable selection is detailed in Figure 14. A user chooses a data type, and a set of annotations to use in performing variable selection. The user then specifies whether to perform a quantitative or qualitative (i.e. categorical) selection on the annotations. Figure 15 illustrates an instance in which CGH data is variable selected for only those loci whose F statistics against patient survival exceed the numerical value 5. As a result of navigation through these pages, a new data type becomes available to the user in which only the variables whose annotations satisfy the selection criteria are returned for analysis.

Magellan utilizes this information at the level of the interaction between the business logic (residing in compiled Java classes) and the Oracle database. When the user chooses to perform an analysis on the selected data type, the system first performs a query to determine which annotations satisfy the user specified criteria, and links those annotations to specific rows of data through identifiers. Only those rows of data are returned to the system for analysis.


Subselect Data Types

Select a data type to subselect:
- Survival
- CGH
- Expression

Select subselection criteria for the data type:
- Subselect the chosen data type based on one of its annotation sets: Fatal vs Survival
  - Fatal vs Survival
  - Prop. missing
  - Mapping of HumArray 1.14 against Aug 2001 genome freeze
- Remove a subselected data type from the List of available data Types

Subselect Data

Subselect data type CGH by annotation set: Fatal vs Survival

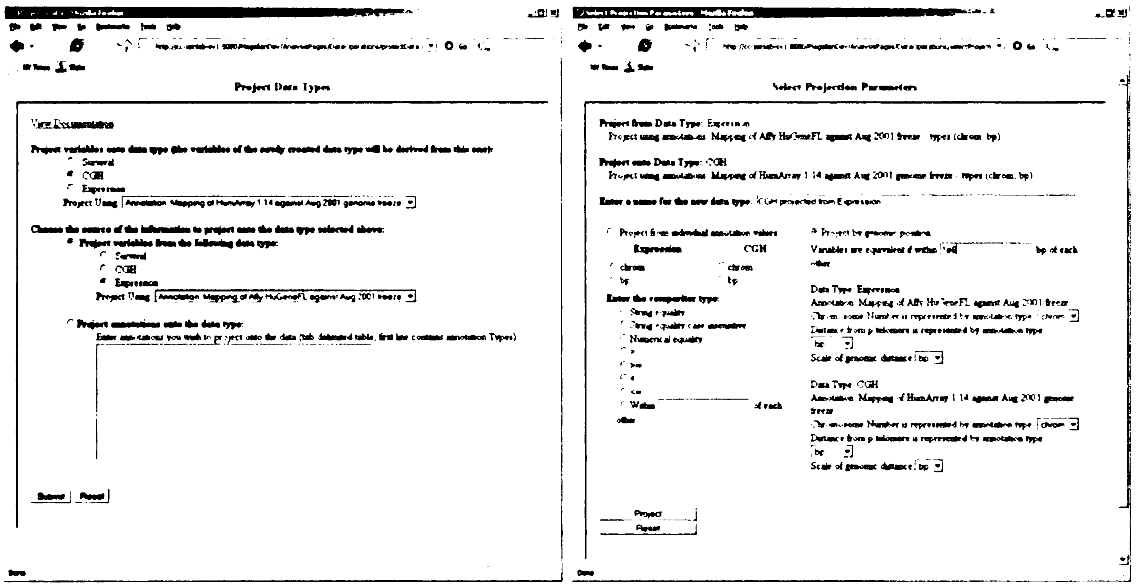
Subselect using annotation type: corr
- Quantitatively: Annotation is > the numerical value 5.0
- Qualitatively: Annotation has the value(s)

Enter a name for the new data type: CGH test 5

Figure 14 – User interface to perform variable selection on a data type.

7.5.3. *Variable Projection*

As stated in the previous chapter 'projection' is a term used to describe the process of finding equivalent variables between data types or data sets. The user interface for performing projections is illustrated in Figure 15. A user selects a data type that serves as the source of the projected variables, a second data type onto which the variables will be projected, and the annotations that will be used to determine equivalence. In the succeeding web page, the user defines the criteria for equivalence of annotations; in the example shown, projection by genomic position has been chosen such that only those BACs within 1Mb of the genes will be retrieved to make up the new data type.


File Edit View Go Bookmarks Tools Help
Go
Project Data Types
View Documentation
Submit
Reset
Done
Select Projection Parameters
Project
Reset

Project variables onto data type (the variables of the newly created data type will be derived from this one):
- ( ) Survival
- (●) CGH
- ( ) Expression

Project Using: Annotation Mapping of HumArray 1.14 against Aug 2001 genome freeze

Choose the source of the information to project onto the data type selected above:
- (●) Project variables from the following data type:
  - ( ) Survival
  - ( ) CGH
  - (●) Expression

  Project Using: Annotation Mapping of Affy HuGeneFL against Aug 2001 freeze
- ( ) Project annotations onto the data type:

  Enter annotations you wish to project onto the data (tab delimited table, first line contains annotation Types)

Project from Data Type: Expression
Project using annotations: Mapping of Affy HuGeneFL against Aug 2001 freeze - types (chrom, bp)

Project onto Data Type: CGH
Project using annotations: Mapping of HumArray 1.14 against Aug 2001 genome freeze - types (chrom, bp)

Enter a name for the new data type: CGH projected from Expression

( ) Project from individual annotation values

Expression	CGH
( ) chrom	( ) chrom
( ) bp	( ) bp

Enter the comparison type:
- ( ) String equality
- ( ) String equality case insensitive
- ( ) Numerical equality
- ( ) >
- ( ) >=
- ( ) <
- ( ) <=
- ( ) Within ____ of each other

(●) Project by genomic position

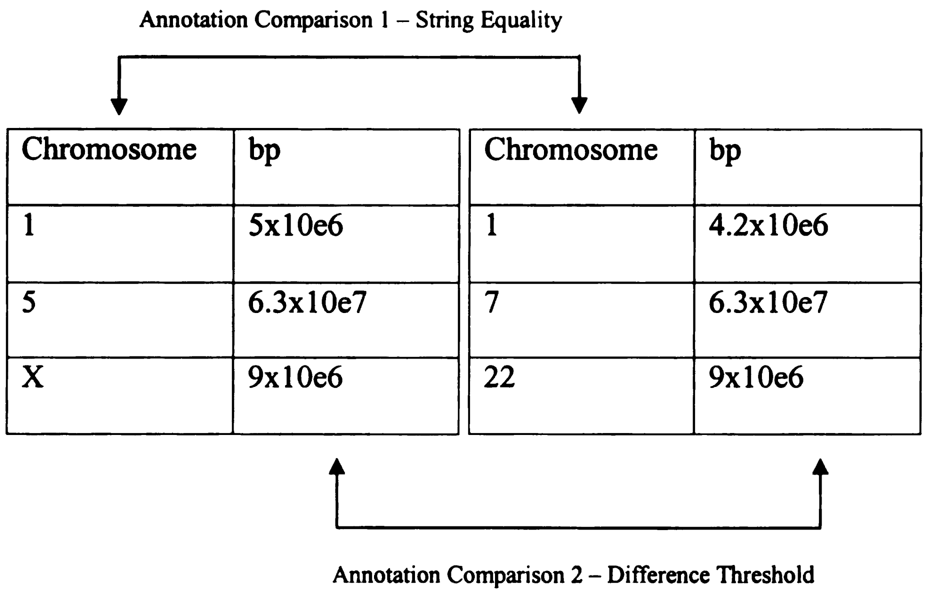
Variables are equivalent if within 1e6 bp of each other

Data Type: Expression
Annotation: Mapping of Affy HuGeneFL against Aug 2001 freeze
Chromosome Number is represented by annotation type: chrom
Distance from p telomere is represented by annotation type: bp
Scale of genomic distance: bp

Data Type: CGH
Annotation: Mapping of HumArray 1.14 against Aug 2001 genome freeze
Chromosome Number is represented by annotation type: chrom
Distance from p telomere is represented by annotation type: bp
Scale of genomic distance: bp

Figure 15 – User interface to perform variable projection between two data types.

The concepts of projection and annotation comparison are implemented as Java classes. The projection class stores pointers to the data types that serve as the source and destination of variable projection, as well as the annotations to be used in determining variable equivalence. The annotation comparison object stores the information the user has specified in determining equivalence, i.e. whether the comparison is to be based on string equality or quantitative relationships, and the comparison operator and threshold involved. By stringing together multiple annotations with multiple annotation comparison objects, complex projection events involving multiple annotation types can be made. For example, comparison of genomic position involves a two part comparison including string equality of chromosome name and a thresholded difference in distance from p-telomere. The overall comparison can be made by instantiating multiple annotation comparison objects, one for each annotation type to be used in the determination of equivalence (Figure 16).


Annotation Comparison 1 – String Equality
Annotation Comparison 2 – Difference Threshold

Chromosome	bp
1	5x10e6
5	6.3x10e7
X	9x10e6

Chromosome	bp
1	4.2x10e6
7	6.3x10e7
22	9x10e6

Figure 16 - Projection involving multiple annotation comparisons

7.6. Analytical Processes

7.6.1. UML Representation of the Analytical Process

The process by which an analysis on data and annotations is performed within Magellan is depicted in the Unified Modeling Language sequence diagram in Figure 17. Briefly, the user selects a data set in her browser, and a list of data types are retrieved from the database and displayed in the browser window. The user then chooses an analytical method and parameters for that method including the data and annotations (if needed) to be analyzed. The analysis is initiated, which consists of retrieving the data as a flat file from the database, forking off the analytical process that operates on the data, and informing the user when the results are ready to be viewed and/or downloaded.

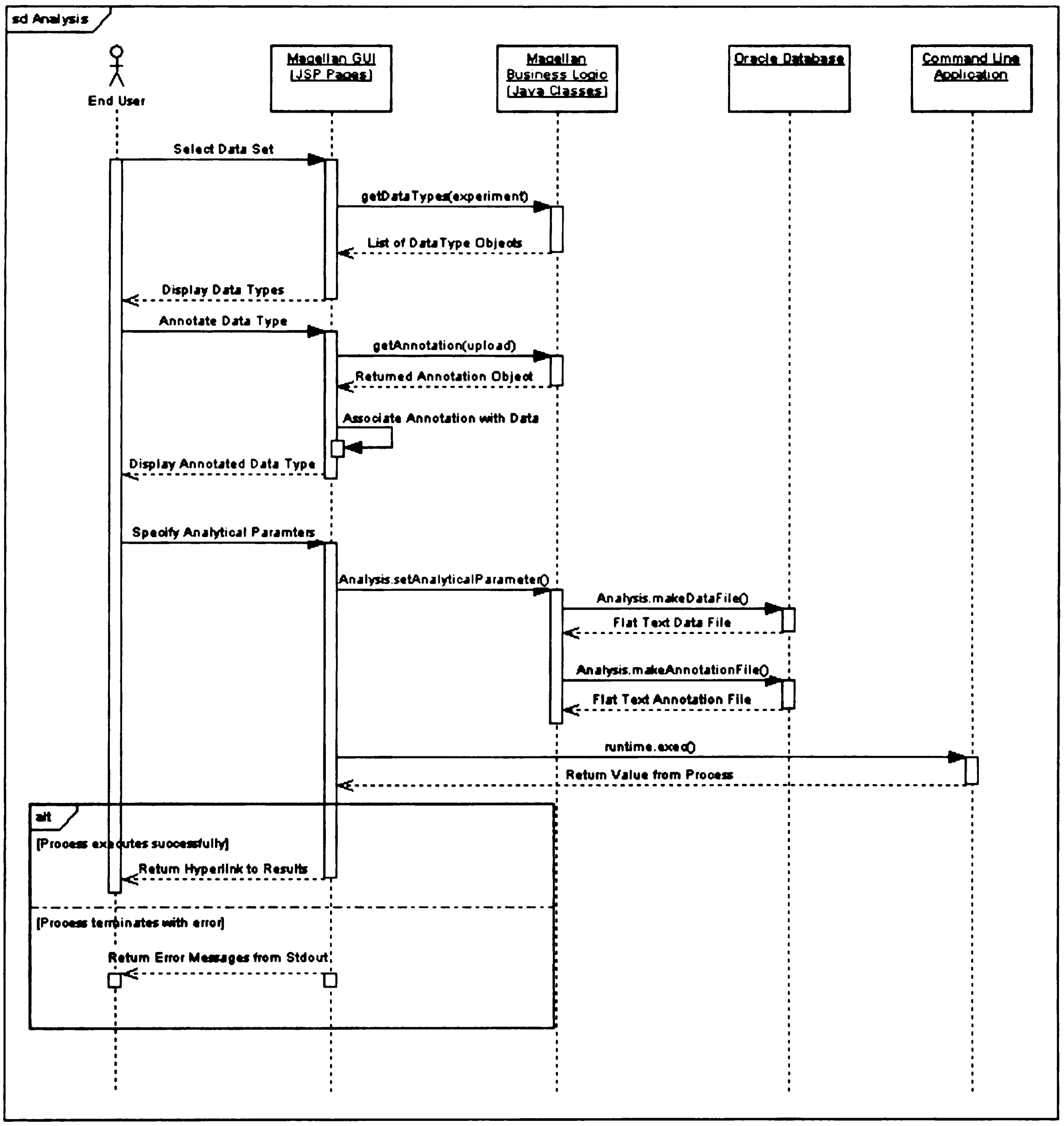
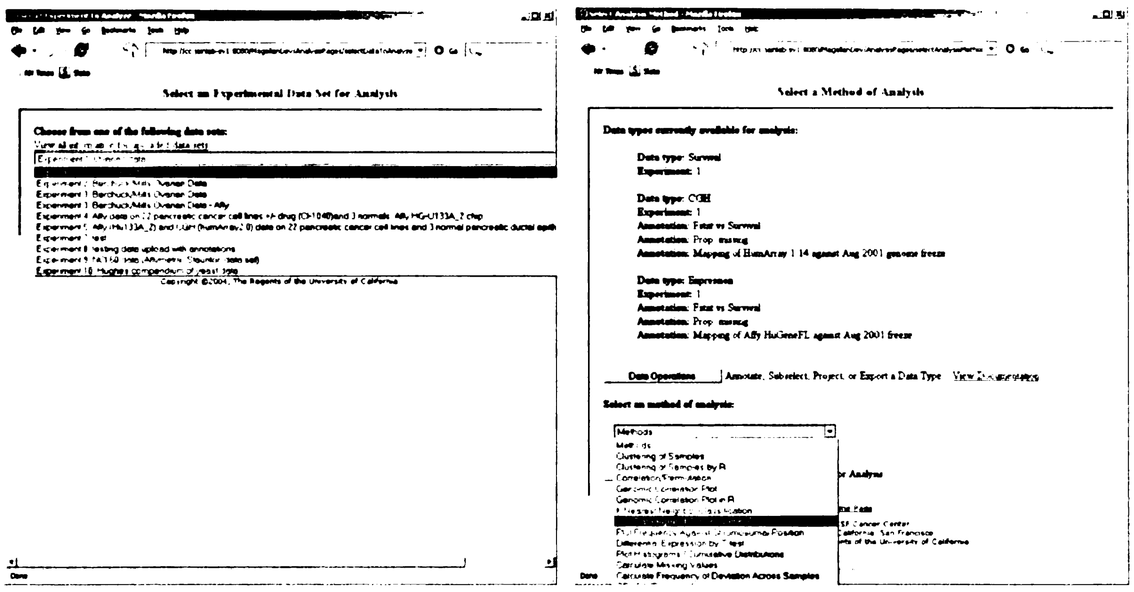

sd Analysis
End User
Magellan GUI (JSP Pages)
Magellan Business Logic (Java Classes)
Oracle Database
Command Line Application
Select Data Set
getDataTypes(experiment)
List of DataType Objects
Display Data Types
Annotate Data Type
getAnnotation(upload)
Returned Annotation Object
Associate Annotation with Data
Display Annotated Data Type
Specify Analytical Paramters
Analysis.setAnalyticalParameter()
Analysis.makeDataFile()
Flat Text Data File
Analysis.makeAnnotationFile()
Flat Text Annotation File
runtime.exec()
Return Value from Process
alt
[Process executes successfully]
Return Hyperlink to Results
[Process terminates with error]
Return Error Messages from Stdout

Figure 17 – UML sequence diagram of the execution of an analytical method on data and annotations within Magellan.

7.6.2. User Interface to the Analytical Functions Deployed in Magellan

The analytical processes deployed in Magellan to date utilize a similar user interface and flow of operation (Figure 18). Briefly, the user selects a data set from the list of data sets to which she has access. The subsequent page allows the user to select a method of analysis, while the page after that provides an HTML form into which the user specifies analysis parameters. Once the analysis parameters have been provided, the data is retrieved, the application executed, and the user is presented with a hyperlink to the results, which can be viewed in a web browser.


Select an Experimental Data Set for Analysis

Choose from one of the following data sets:

View all information for available data sets

Experiment 1 Ovarian Data

Experiment 2 Berchuck/Mills Ovarian Data
Experiment 1 Berchuck/Mills Ovarian Data
Experiment 3 Berchuck/Mills Ovarian Data - Affy
Experiment 4 Affy data on 22 pancreatic cancer cell lines +/- drug (CI-1040)and 3 normals Affy HG-U133A_2 chip
Experiment 5 Affy (Hu133A_2) and UCSF (humArray2.0) data on 22 pancreatic cancer cell lines and 3 normal pancreatic ductal epith
Experiment 7 test
Experiment 8 testing data upload with annotations
Experiment 9 NCI 60 data (Affymetrix Staunton data set)
Experiment 10 Hughes compendium of yeast data

Copyright ©2004, The Regents of the University of California

Select a Method of Analysis

Data types currently available for analysis:

Data type: Survival
Experiment: 1

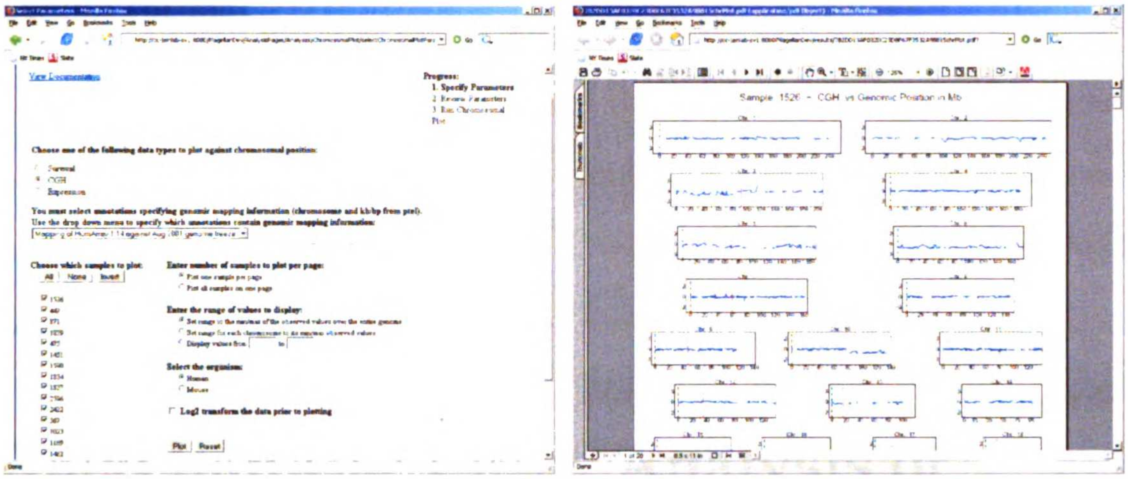
Data type: CGH
Experiment: 1
Annotation: Fatal vs Survival
Annotation: Prop. missing
Annotation: Mapping of HumArray 1.14 against Aug 2001 genome freeze

Data type: Expression
Experiment: 1
Annotation: Fatal vs Survival
Annotation: Prop. missing
Annotation: Mapping of Affy HuGeneFL against Aug 2001 freeze

Data Operations Annotate, Subselect, Project, or Export a Data Type View Documentation

Select an method of analysis:

Methods
Methods
Clustering of Samples
Clustering of Samples by R
Correlation Form Aneca
Genomic Correlation Plot
Genomic Correlation Plot in R
K Nearest Neighbor Classification
[illegible]
Plot Frequency Against Chromosomal Position
Differential Expression by T-test
Plot Histograms / Cumulative Distributions
Calculate Missing Values
Calculate Frequency of Deviation Across Samples



View Documentation

Progress:
1. Specify Parameters
2. Review Parameters
3. Run Chromosomal Plot

Choose one of the following data types to plot against chromosomal position:

- Survival
- [✓ CGH]
- Expression

You must select annotations specifying genomic mapping information (chromosome and kb/bp from pter).
Use the drop down menu to specify which annotations contain genomic mapping information:
Mapping of HumArray 1.14 against Aug 2001 genome freeze

Choose which samples to plot:

Enter number of samples to plot per page:
- [✓ Plot one sample per page]
- Plot all samples on one page

Enter the range of values to display:
- [✓ Set range to the maximum of the observed values over the entire genome]
- Set range for each chromosome to the maximum observed values
- Display values from ___ to ___

Select the organism:
- [✓ Human]
- Mouse

Log2 transform the data prior to plotting

Sample 1526 – CGH vs Genomic Position in Mb

Figure 18 – User interface for execution of an analytical function in Magellan. The data selection, analytical method selection, parameter specification and results pages are shown.

7.7. Conclusions

Magellan was designed as a client-server based web application with a centralized Oracle database, Java business logic, and JSP based presentation logic. The executables that comprise the analytical layer of the system are separate programs that are modularly deployed within Magellan and run as multithreaded applications. The design choices made in the development of Magellan reflect a desire for generality and extensibility. The abstract, type-value based method of representing data and annotations allows for multiple, arbitrary types of information to be stored and retrieved for analysis. The use of the Java programming language in the implementation of Magellan insures an ease of extensibility of the application, such that developers can add new functionality to the application by utilizing and extending the Java API.

Chapter 8
Magellan Use Case: Analysis of Genomic Data in Ovarian Cancer

8.1. Introduction

The previous two chapters covered design and implementation decisions in the development of Magellan. This chapter discusses in detail an example of the use of Magellan on genomic and clinical data derived from ovarian tumors (Kingsley, Kuo et al. 2005).

8.2. Description of the Data Set

Twenty primary human ovarian tumor samples were analyzed in this study. Ten were tumors from patients that survived more than 7 years (*long* survivors), and 10 were from patients that survived less than 3 years (*short* survivors).

The tumors were analyzed by array CGH using experimental procedures previously described (Hodgson, Hager et al. 2001; Snijders, Nowak et al. 2001). Briefly, DNA samples from the tumors and from normal tissues were labeled with CY3 and CY5, respectively, and hybridized to an array comprised of replicate BAC clones distributed at

~ megabase intervals along the genome. Hybridized arrays were counterstained with DAPI to facilitate array element segmentation. CY3/CY5/DAPI images were analyzed to determine CY3/CY5 intensity ratios for each element in the array using custom software (Jain, Tokuyasu et al. 2002). Measurements of individual array elements were discarded if the within-spot Cy3:Cy5 pixel intensity correlation was <0.81, or if the number of pixels per array element was <25 (the average spot size was 65 pixels) or if the CY3:CY5 ratio was >20% from the mean of 4 replicate measurements). This resulted in CGH measurements from 2309 genomic loci after quality control and elimination of loci where more than 50% of samples had missing values. Note that missing values are typically caused by rejection of spots based on statistical concerns such as high replicate spot variance. Regions of the genome that have been deleted in the tumors should still be represented in the data set.

Expression data was collected as reported (Lancaster, Dressman et al. 2004), using the Affymetrix HuGeneFLGeneChip and Microarray Analysis Suite (http://www.affymetrix.com). This resulted in 7129 gene expression values for each sample.

As described above, CGH and mRNA expression data were obtained from twenty ovarian tumors, comprising ten long and ten short survivors. Magellan was used to explore the relationship between these two types of genomic data, and between the genomic data and accompanying clinical information. Several standard analyses were performed, which made use of a single data type at a time, with no use of annotation information. I also performed multiple analyses that required annotation information, multiple data types, or both.

8.3. Single-Mode Analysis

Figure 19 shows the frequency of genomic copy number alteration for the twenty tumor samples. Gains and losses previously shown to be common in ovarian tumors using chromosomal CGH (Shayesteh, Lu et al. 1999) are present. The resolution of array-based CGH further sharpens the structure of the abnormalities.

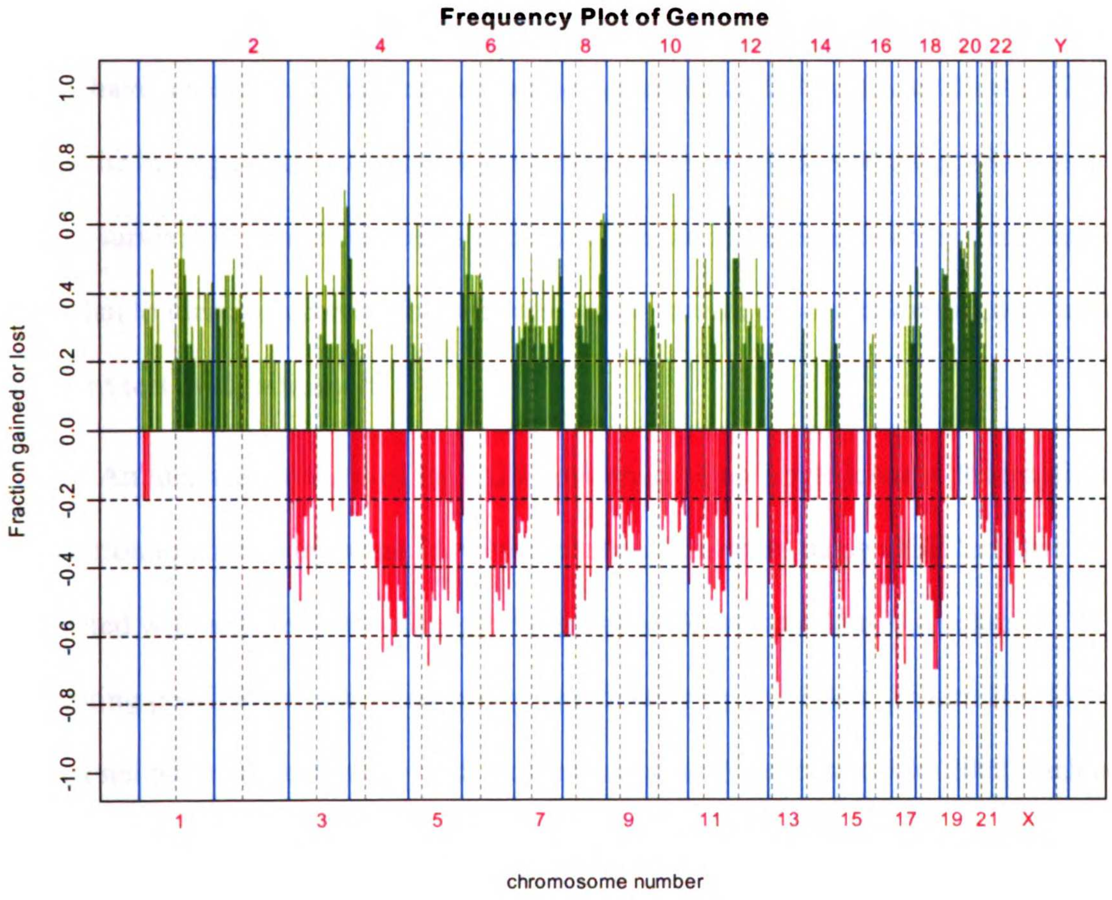


Figure 19 - Graphical representation of the frequency of chromosomal gains/losses in 20 ovarian tumors

Correlation analysis was used to determine the relationship between genomic and clinical data, that is whether the behavior of any single gene or genomic locus significantly correlated with patient survival class by t-test. The significance threshold for the t-statistic was determined by a maxT permutation-based approach, as described previously (Jain, Chin et al. 2001) using the 95th percentile of the max permutation distribution as a significance cutoff. A significant correlation between the expression of the spermidine/spermine N1-acetyltransferase (SSAT) gene and patient survival at $p<0.05$ was observed. This is a conservative approach, since the null distribution is calculated from the maximum observed t-statistic from each round of permutation. The SSAT gene has been previously observed to be elevated in human prostate cancer, where it may have a role in maintaining polyamine homeostasis (Bettuzzi, Davalli et al. 2000). While the biological and clinical interpretation of the correlation between SSAT and patient survival in ovarian cancer is unclear at this point, the result demonstrates use of Magellan to identify a correlation between gene expression and clinical outcome under a stringent test of significance.

An identical analysis of the CGH data revealed no single locus with significant association to clinical outcome. However, clustering samples using CGH loci that correlated with outcome showed a good segregation of samples based on class, suggesting possible success of outcome classification by CGH data (Figure 20). K-nearest neighbors classification with variable selection (Olshen and Jain 2002) yielded reasonable classification performance using a leave one out cross-validation. Variable selection was performed for each round of cross validation, using the t-statistic vs. outcome to select the top 50 variables for each sample holdout. Using the Euclidean distance metric, I consistently observed a better than random fraction of correctly classified tumors using a number of different values for parameters such as the number of holdouts and number of neighbors (Figure 21A). Due to the very small sample size, cross-validation estimates of classification success were better for leave-one-out testing than for tests run with larger holdout sets. For the single-holdout cases, classification performance was as high as 80 to 85%. Figure 21B shows the effect of varying the number of selected variables under different values of K, using a single holdout. With a very small number of variables, performance was poor, but there is a broad peak of performance from 25 to 100 variables, centered at 50. Classifiers based on mRNA expression data underperformed classifiers based on CGH data, despite the converse result in the univariate analyses.

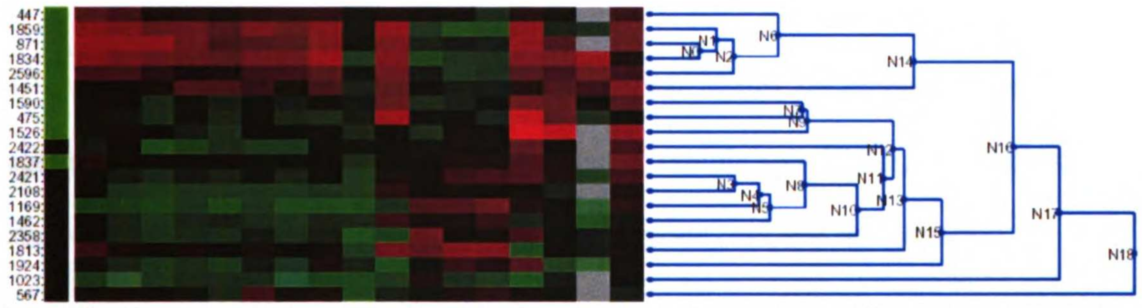


Figure 20 – Hierarchical clustering of samples based on loci whose CGH profile correlated strongly with outcome

A.

			K		
		1	**3**	**5**	**7**
	1	.85	.85	.80	.80
	2	.75	.70	.65	.50
Holdouts	**3**	.80	.75	.70	.60
	4	.60	.80	.70	.70
	5	.55	.70	.55	.55

B.

		K			
		1	3	5	7
	5	.50	.45	.40	.65
	10	.70	.50	.55	.60
Loci	**25**	.70	.80	.70	.60
	50	.85	.85	.80	.80
	100	.70	.75	.65	.55

Figure 21 - Fraction of tumors correctly classified using CGH data under cross validation. A – Fraction of tumors correctly classified using 50 genomic loci but varying the number of holdouts and the value of K. B - Fraction of tumors correctly classified using one holdout but varying the number of genomic loci and the value of K

Given the very small number of samples, these results are encouraging to a degree, but require prospective validation on larger sample sets. The signal within the data is not overwhelmingly strong, and the single-mode analyses do not reveal compelling results with respect to patient outcome.

8.4. Annotation Based Correlation Methods

The preceding analyses were performed independently on the mRNA expression and CGH data sets. In order to *relate* the two data sets, annotations associated with the data were utilized. In particular, genomic mapping data to relate genes to genomic loci was used. Two types of analysis are presented here: 1) where a statistical question relies directly upon annotation information, and 2) where a statistical computation is used as an annotation of genes in order to constrain a question about genomic loci. As previously discussed, the term 'projection' is used to describe the process of finding equivalent

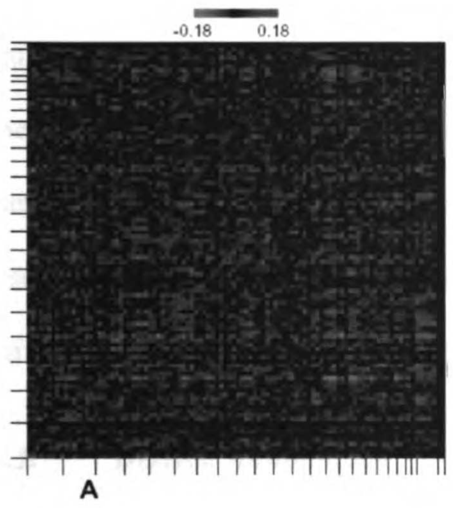
variables between two data types by examining the relationship between the annotations associated with those variables.

8.4.1. Gene Dosage Effect

The presence of frequent alterations in copy number across the genome in this collection of ovarian tumors (Figure 19) suggests that many genes could potentially be affected by gains or losses of genomic DNA. In order to determine the relationship between CGH and expression data (whether the sample-to-sample copy number of any genomic locus was correlated with the mRNA expression of any gene), a correlation analysis similar to that between genomic data and clinical variables was performed. Correlation coefficients were calculated for each pairwise combination of loci and genes, and significance was determined by maxT based permutation analysis. No gene-locus correlations exceeded the 95th percentile cutoff of the permutation analysis, indicating the inherent difficulty of finding relationships between high-dimensional data types with a relatively small number of samples.

We considered a more direct relationship between gene copy number and gene expression by incorporating biological annotations that used the genomic mapping of the BACs of the CGH array and the genes on the Affymetrix chip. The annotations were used to create bins of genes and loci based on common values. In this analysis, the genome was divided into 100 bins of equal length. Each bin of a two-dimensional matrix contained the genes (x-axis) and loci (y-axis) that mapped to those genomic regions indicated on the respective axes. The color of each bin was the average of all cross correlations of the CGH data for each genomic locus vs. the mRNA expression of each gene within the bin.

Figure 22 shows a graphical depiction of the result computed and rendered from Magellan. An area of higher than background correlation is apparent along the diagonal of the graph, suggesting that expression of genes correlated with the amplification or deletion of loci that map nearby in the genome (Figure 22A). The significance of off diagonal correlation values is uncertain, but the correlation values for genes and loci that map to within 1Mb is significantly different from the correlation values for genes and loci that are separated by at least 50Mb (p << 0.001 by t-test). This is evidenced by the right-shift of the cumulative histogram of correlation values in the close-mapping case (Figure 22B).



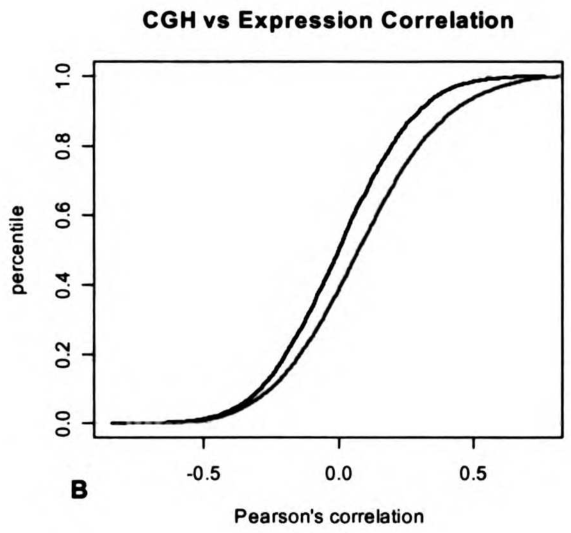


Figure 22 - Correlation of CGH and mRNA expression data, binned by genomic position. A - All by all Pearson's correlation of CGH and expression data, binned by genomic position as described in the text. B - Cumulative distributions of gene/locus correlations for pairs that are within 1Mb of each other (green) or at least 50Mb away from each other (black).

Genome-wide, on average, amplification of a genomic region tends to up-regulate the expression of genes in that region while genomic deletion tends to down-regulate gene expression, a result that has also been observed in breast cancer (Pollack, Sorlie et

al. 2002). The result confirms expectations, but there are two aspects of this analysis with respect to the Magellan platform that are important. First, the explicit use of annotation information (genomic mapping data) was required to make the correspondence between data types. Second, integration of the annotations with the data for the analysis was accomplished using a procedure easily accessible to a naïve user.

8.4.2. Gene Annotations

Gene ontology annotations were available for most of the genes represented in the mRNA expression data. These were loaded in Magellan as curated annotations of the expression data type. As a demonstration of textual annotation use, 394 genes were selected with any of 13 cell cycle related gene annotations (features such as cell cycle, regulation of cell cycle, …). The distributions of gene correlations with survival for all genes versus those of the cell cycle were compared, resulting in a rightward shift of the latter set. The shift was not quite significant (p = 0.07 by Wilcoxon rank sum) and is presented here as an illustration, but the following results using *derived* annotations exhibited significant enrichment of extreme statistics.

8.4.3. Derived Annotations

In addition to curated annotations such as genomic position or gene ontologies, data can be selected based upon quantitative annotations that are derived from the data itself (such as correlation with a clinical outcome) prior to analysis. Combining selections such as these with data projection can be useful in moving among data subsets within an experiment as well as moving between different experiments.

We computed the t-test for the mRNA expression data with respect to patient survival for each gene and selected the option within Magellan to store the values as a derived annotation for the expression data type. The gene expression data set was then variable selected such that only those genes with statistic values in the top 1% were considered. Using the genomic mapping of the genes, those loci in the CGH data set that were in close physical proximity to the selected genes were selected. The cumulative histograms from the distribution of t-statistics for the 95 selected loci and the corresponding distribution for all loci are shown in Figure 23. The cumulative histogram for the selected loci is shifted to the right of that of the unselected loci, indicating that genomic aberrations that are located near genes whose expression correlates with outcome are themselves more strongly correlated with outcome ($p < 0.05$ by Wilcoxon rank-sum test).

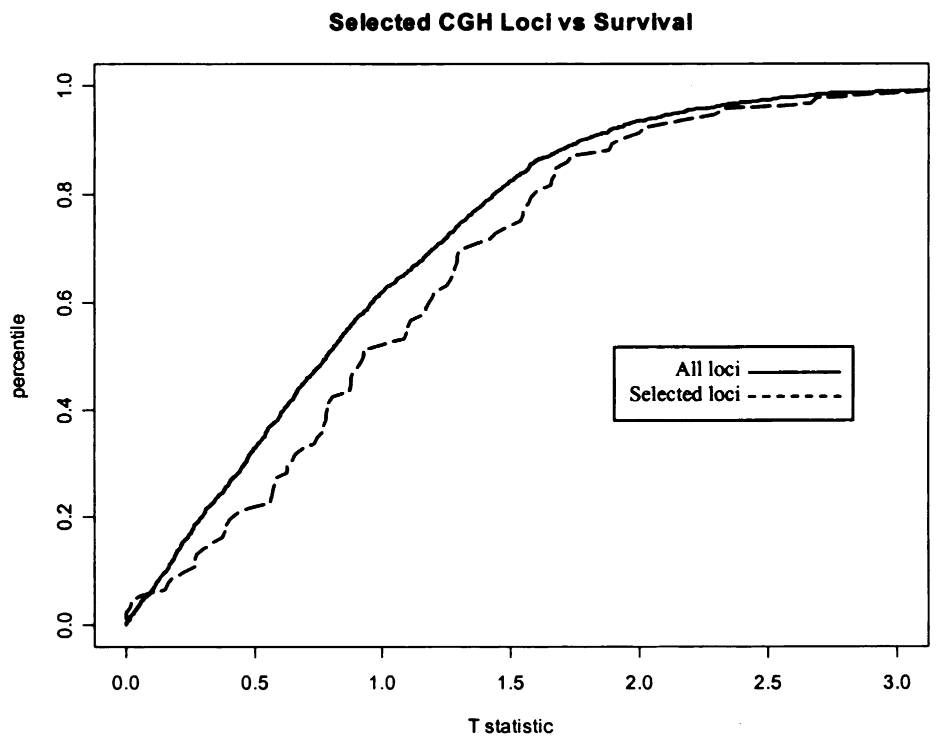


Figure 23 – Projection of variables from mRNA expression to CGH. Cumulative distribution of t-statistics of CGH loci vs. survival for all loci (solid line) or loci that are within 1MB of one of the top 1% of genes that correlated with survival (dashed line).

In data sets with limited numbers of samples, such as the one presented here, it is often easier to observe significant differences in distributions than in individual genes or loci. In the example just presented, variables were projected from gene expression to gene copy number, but outcome information was used in the process. Since a direct relationship between genome copy number and expression has been shown, the effect may be a simple manifestation of the same process.

However, since genome copy number has an expected normal value (a relative log ratio of 0 compared with normal), genomic loci that are involved with the disease process can be identified *without* reference to outcome. Genomic loci were identified that met progressively more stringent tests of frequency of aberration, and the relationship to patient outcome of the genes that mapped close by was examined. Figure 24 shows cumulative histograms of the t-statistics of gene expression for multiple subsets of genes. The leftmost distribution in black was from all genes. Each succeeding distribution was computed from progressively more stringent criteria. The shift in distributions is monotonic, and the difference between the distributions for all genes compared with those genes that map to loci that are most frequently altered is significant ($p < 0.05$ by Wilcoxon rank-sum test).

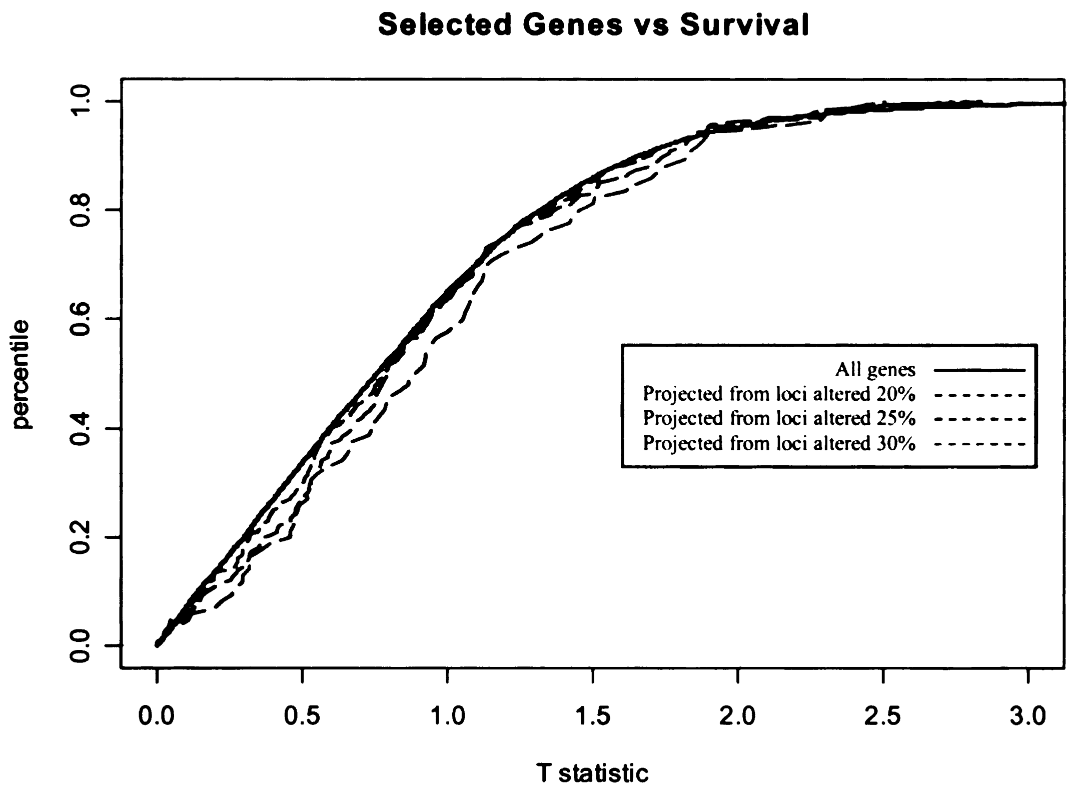


Figure 24 - Projection of Frequently Aberrant CGH loci onto mRNA Expression Data. Cumulative distributions of t-statistics of mRNA expression vs. survival for all genes (solid black line) or genes that are within 1MB of CGH loci whose log2 deviation from normal is at least 0.5 at a frequency of at least 20% (dashed green line), 25% (dashed blue line), or 30% (dashed red line) of samples.

8.5. Conclusions

We demonstrated Magellan's application on a specific genomic data set containing CGH, mRNA gene expression, and clinical outcome data from ovarian tumors. Using this system, I was able to identify a significant correlation between expression of the SSAT gene and patient survival, and was able to build a CGH based classifier that correctly predicts survival up to 85% of the time in cross validation. It is important to note that these results are derived from a small data set comprising a selected group of patients, and any conclusions need to be reproduced on larger data sets.

b

a

is

d

g

i

k

By making use of curated annotations, stronger conclusions about the relationship between genome copy number and gene expression could be made. In particular, on average, genome-wide, there is a significant component of gene expression variation that is explained by changes in genomic copy number. Using combinations of curated and derived annotations, I showed enrichment for relationship to survival in moving from gene expression data to genome copy number data and *vice versa*. The power of integrated analysis with the combination of annotations, both curated and derived, is a key distinguishing feature of Magellan.

Chapter 9
Integration of Magellan with the caBIG Project

9.1. Introduction

While Magellan was under development, a multi site cancer bioinformatics project was initiated which shared some of the same goals as Magellan. This project was sponsored by the National Cancer Institute's Center for Bioinformatics, and was known as the cancer bioinformatics grid, or caBIG initiative (http://cabig.nci.nih.gov). The stated goal of caBIG is to create a 'grid' or network that enables individuals at different institutions to easily share data and analysis tools. caBIG is designed to provide a 'world wide web' to the cancer research community, to facilitate the delivery of new analytical approaches, and speed the pace of discovery. Magellan became a funded development project within the caBIG initiative.

The caBIG initiative currently includes nearly 500 participants from roughly 50 Cancer Centers and other organizations. Approximately 70 projects are being funded in a three year pilot project, with about 10 development projects within the Integrative Cancer Research Workspace, of which Magellan is a part. Our funding was initiated during the

second year of the caBIG pilot project (Figure 25), and the specifics of the project centered on the integration of Magellan with the standards and tools within caBIG to facilitate its broad adoption by the cancer research community. One of the unique aspects of caBIG participation is that developers are matched with *adopters* who are funded to collaborate in testing and using tools. Magellan's adopters were chosen from David Fenstermacher's group at the University of Pennsylvania. This chapter describes progress in integrating Magellan with the tools and information stores made public by caBIG.

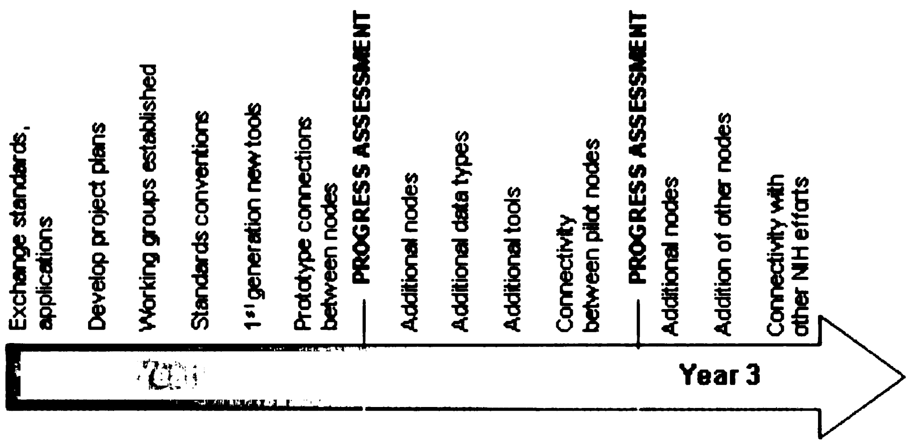


Figure 25 – The project milestones for the three year caBIG pilot project

9.2. Goals of caBIG

The head of the National Cancer Institute, Andrew von Eschenbach, recently stated that the goal of the NCI should be to eliminate the suffering and death associated with cancer by the year 2015. Whether this timetable is feasible or not has been the subject of debate, but certainly the goal is a worthy one and the caBIG project is designed to be one instrument in achieving that goal.

The current paradigm of cancer genomics data analysis has been one in which scientists at remote institutions generate data, perform analyses using publicly available or homegrown analytical methods, and share their data and results through publications, meetings, and web sites. caBIG hopes to change this paradigm by centrally storing large numbers of queryable data sets and providing a large suite of analytical tools that can be accessed over a grid. In addition, caBIG proposes to make these tools interoperable, such that the functionality of one analytical tool can be accessed by another analytical tool through simple API's.

A workflow supported by caBIG could, therefore, include one in which multiple applications can be sequentially called upon to perform operations on biological information. As such, a workflow could consist of several steps: retrieval of several data sets from the database, normalization of the different data sets and subsequent merging, feature selection of variables, followed by the application of an analytical function or visualization (Figure 26). In order for this interoperability to be achieved, each of the applications must communicate with each other through well documented API's. In addition, a standard for the representation of data and for preliminary analytical results will be required.

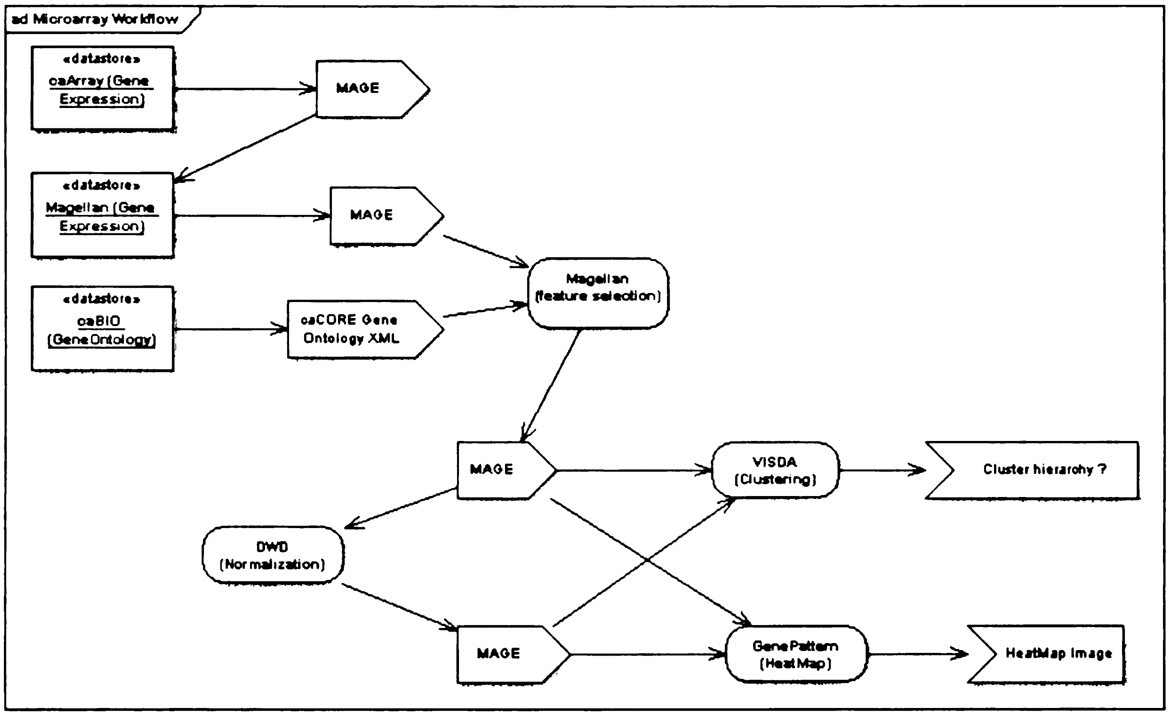


Figure 26 – A workflow example showing interoperability of applications in caBIG

9.3. Compatibility Standards Imposed by caBIG

One of the goals of caBIG is to assemble tools that are interoperable with one another, such that one application can access and use the components of another application. In this case, 'access' refers to programmatic access to data and tools from within other software applications, not just interactive access from user interfaces. Such interoperability requires a common standard for the information to be shared between applications. In the absence of such standards, a bioinformatics 'Tower of Babel' will exist in which each application provides information using its own data representation. In order for any two applications to communicate under such circumstances a 'translator' must be written for each application pair, an untenable situation as the number of applications becomes large.

For the immediate future, caBIG has adopted an approach where every data type to be exposed by public API's available over the grid will be registered in a repository known as the caDSR (cancer data standards repository). Under this scenario, developers who intend to programmatically access the functionality of other grid-enabled applications will at least have a means of determining the representation of information that particular applications require. While it is hoped that future caBIG developers will adopt data types already curated in the caDSR, this is not a formal requirement for current projects funded by caBIG.

While the imposition of standards is important for the exchange of information between applications, such standards can place limits on the types of information that can be represented. The planning committees of caBIG have mandated that new types of data that emerge from the experimental community must be registered in the caDSR, and it is expected that standards will be adopted in conjunction with the vocabulary and common data element workgroup of caBIG. To some extent, there is a tradeoff between ease of exchange vs. extensibility in the adoption of standards, such that the appropriate level of specificity becomes very important. While these standards will be useful for exchange between applications, it will be important to make these standards flexible enough that their adoption does not place an undue burden on application developers.

9.4. NCI Data Sources Accessible through caBIG

The caBIG project is not simply a grid framework under which applications can communicate. The NCI has also assembled a large amount of bioinformatics information to be accessed by the various applications developed under the caBIG umbrella.

9.4.1. caBIO

Cancer Bioinformatics Infrastructure Objects (caBIO) is an object model that allows for programmatic access to a series of clinical and genomic data sources. As such, caBIO provides a Java API that allows software applications to access sources of information that are aggregated and stored at the NCI. The data sources currently available through the caBIO API include:

- SAGE Data (CGAP) – NCI and Duke university SAGE experiment data
- Expression Measurements (NCICB GEDP) - Probe sets
- Sequence Trace Files (GAI) - EST traces and full-length mRNA clone traces
- Genetic Annotation Initiative (GAI) - SNPs
- Sequence Verified Clones - Human and mouse sequence-verified clone information
- Cancer Clinical Trials (NCI CTEP and PDQ) - Trials and drug agent information
- CMAP Annotation Data (CMAP) - Drug targets, anomalies
- Cancer Vocabulary (NCI) - Cancer related terminology and concepts
- Unigene (NCBI) - Human and mouse genes, sequences, map locations, clones, proteins
- Homologene (NCBI) - Human and mouse gene homologs
- LocusLink (NCBI) - Genes, gene ontologies, gene aliases, taxons
- RefSeq (NCBI) - Reference sequences
- EST Data (NCICB) - Tissue-specific expression level ESTs
- cDNA library information (NCICB) - cDNA libraries for disease and tissue
- Human Genome via UCSC - Genomic sequences, annotations, and map coordinates
- BioCarta (BioCarta) - Pathways
- Gene Ontology - Hierarchy of gene functions

The interaction of Magellan with caBIO is designed to facilitate the transfer of curated annotation information from these NCI data sources for local storage in the Magellan database. For example, gene ontology terms that annotate the genes in a

particular mRNA expression data set can be downloaded via caBIO and used for variable selection to focus on user specified gene subsets.

9.4.2. caArray

caArray is an open source, standards based repository for storing and retrieving microarray data. caArray is compliant with the MIAME 1.1 standard (Brazma, Hingamp et al. 2001) for experimental annotation, such that metadata including array design, experimental protocols, and biomaterials can be stored in association with the data itself. In addition, caArray is designed to be compliant with a number of different file/exhange formats for the input and output of microarray data, including MAGE-ML (Spellman, Miller et al. 2002), Affymetrix .cel files (http://www.affymetrix.com), and GenePix (http://www.moleculardevices.com). For the software development community, caArray provides APIs for programmatic access to microarray data, such that entire microarray data sets can be retrieved from within applications.

While it is hoped that Magellan will be able to retrieve data from caArray, this is currently a secondary objective in the Magellan-caBIG collaboration. caArray is currently under development and the performance of data downloads is still somewhat slow. Application developers have cited times of 30 minutes to an hour to retrieve moderate sized data sets, which are not currently practical for on the fly analysis in Magellan. Because of the current performance issues with caArray, it is anticipated that Magellan will mirror individual data sets that are selected for download by end users. Once a data set is downloaded from caArray and stored locally in the Magellan database, it will be available for analysis by Magellan's analytical applications.

9.5. Interaction of Magellan with caBIO

The consumption of annotation information from caBIO is the first implementation of an interaction between Magellan and the data sources at caBIG. The model for this interaction is one in which large numbers of identifiers will be passed to the a public API, and identifier-annotation pairs will be returned and stored locally in the Magellan database. The various data sources curated by caBIG are accessible through an application programming interface called the caCORE API. The following two sections illustrate how users would use Magellan to incorporate caBIO based annotations into an analysis of data. The first section provides a high level illustration of how a user would use Magellan to retrieve annotations from caBIO and then use those annotations for variable selection. The second section provides a more detailed examination of how Magellan interacts with the caCORE API to retrieve annotation information.

9.5.1. UML Use Case Model

The means by which an end user would incorporate caBIO based annotations into a specific analysis in Magellan are illustrated in a UML use case diagram in Figure 27. This diagram details the steps an end user would take to perform an annotation based variable selection on mRNA expression data, followed by a correlation analysis on the resulting data set. The user chooses a particular mRNA expression data set for analysis, and selects gene ontology biochemical pathway terms to annotate the genes. The GO annotations are retrieved from caBIO and merged with the mRNA expression data based on common identifiers. The user then selects a particular GO term (or terms) to select variables from the data set; for example, 'GPCR' to select only those genes that are annotated as G-protein coupled receptors. Once this selection has been made, the user

[illegible]

can choose an analytical method such as correlating the variable selected data type against a clinical variable such as patient outcome. The user specifies the parameters for this analysis and retrieves the results.

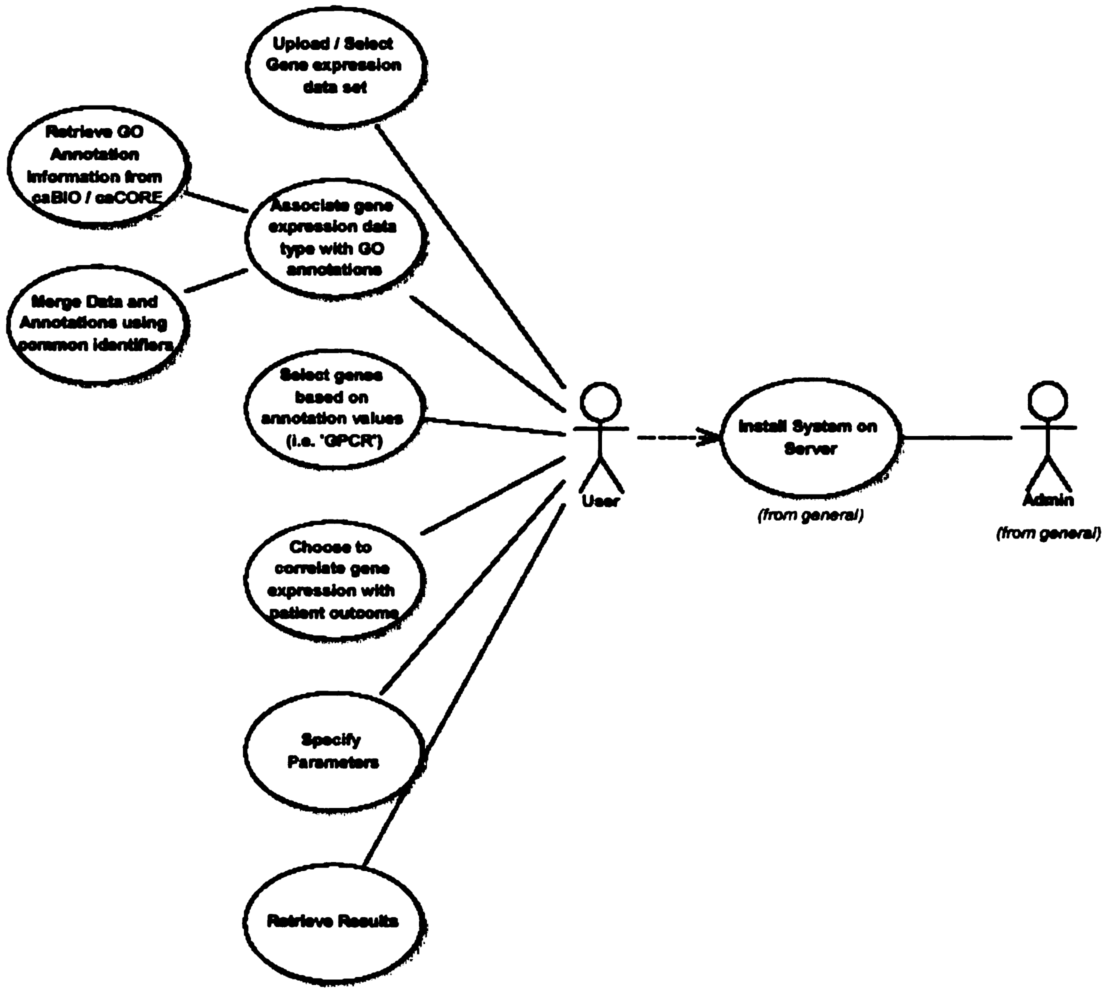
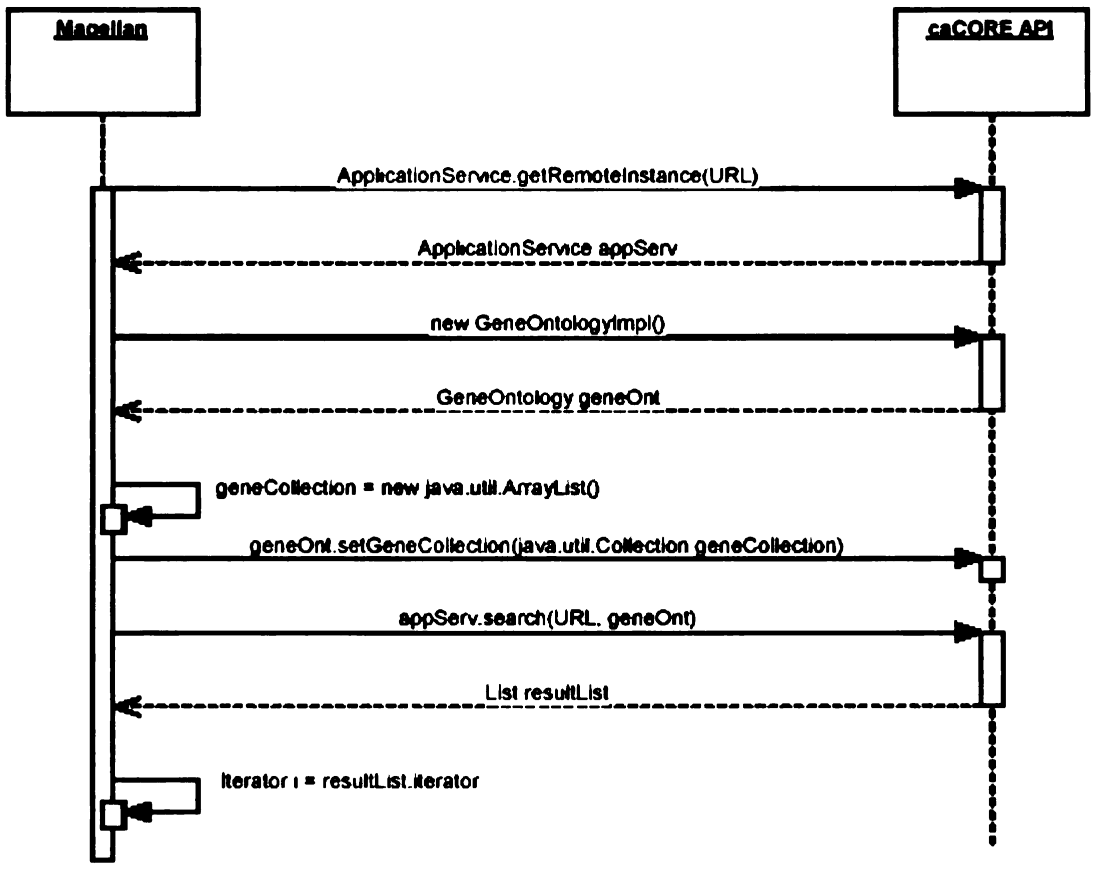

Upload / Select Gene expression data set
Retrieve GO Annotation information from caBIO / caCORE
Associate gene expression data type with GO annotations
Merge Data and Annotations using common identifiers
Select genes based on annotation values (i.e. 'GPCR')
Choose to correlate gene expression with patient outcome
Specify Parameters
Retrieve Results
User
Install System on Server
(from general)
Admin
(from general)

Figure 27 – UML use case diagram of caBIO derived annotation based analysis in Magellan

9.5.2. UML Sequence Diagram

The mechanism by which Magellan will interact with the caCORE API to retrieve annotations from caBIO is detailed in a UML sequence diagram in Figure 28. This

diagram illustrates the steps taken by the Magellan Java business logic to retrieve the gene ontology annotations used in the use case diagram of Figure 27. The caCORE API allows LocusLink or OMIM identifiers to be queried against the caBIO annotation sources, so that collections of identifiers can be passed to Java objects representing different annotation classes, such as the gene ontology example below.



Magellan
caCORE API
ApplicationService.getRemoteInstance(URL)
ApplicationService appServ
new GeneOntologyImpl()
GeneOntology geneOnt
geneCollection = new java.util.ArrayList()
geneOnt.setGeneCollection(java.util.Collection geneCollection)
appServ.search(URL, geneOnt)
List resultList
Iterator i = resultList.iterator

Figure 28 – Access of gene ontology terms by Magellan using the caCORE API

9.5.3. Performance

Currently, the caCORE API does not provide for batch queries in which several thousand identifiers-annotation pairs can be retrieved. As such, only one identifer can be

passed to the caCORE API at a time, and one annotation set retrieved. While this does allow for retrieval of annotation information, the amount of time required to access several thousand annotations would be measured in hours, a time frame that is impractical at present. Fortunately, the caCORE API is being updated with respect to its querying capabilities and it is anticipated that the performance will improve shortly.

9.6. Conclusions

The caBIG project has the stated goal of linking together information sources and analytical tools in one web portal. The centralization of large amounts of cancer genomics information and large numbers of applications that operate on that information has great potential to improve the productivity of cancer researchers around the world. For this promise to be realized however, several criteria must be met. First, reasonable exchange standards for data, annotations, and the results of computations must be defined. These standards must be sufficiently detailed such that information can be exchanged across applications deployed on the grid. At the same time, they must be flexible enough to accommodate new types of information as they arise. In addition, the performance of querying and retrieval for the information stores at caBIG (caArray and caBIO) must be improved such that both data and annotations can be accessed by analytical applications in a timely fashion.

Magellan's role in the caBIG project is to serve as a test case for integration of an analysis tool within the larger analytical grid. Because Magellan is currently deployed as a web-based application as opposed to a grid enabled API, Magellan does not have to meet as many stringent data and vocabulary standards as some other analytical methods. Our primary responsibility in the integration of Magellan with caBIG is to create an

interface to caBIO and caArray. While neither API currently has a high enough performance to allow smooth and timely access of data and annotations, it is expected that this will occur in the future.

Since caBIG is structured as an open-source and highly collaborative project, many of Magellan's future improvements will occur with the effort of other groups within the caBIG consortium. In particular, the caBIG group at UPenn directed by David Fenstenmacher is actively porting Magellan's database layer to MySQL, an open-source database management system. The UPenn group has a large set of users within their cancer research community who have expressed interest in making use of Magellan, and it appears that UPenn will represent a major center of further development of the system.

[illegible]

Chapter 10 Conclusions

This dissertation covered both the development and use of the Magellan application. As discussed in the second chapter of this thesis document, cancer is a disease of alterations in DNA, and these alterations are increasingly measurable using the microarray technology discussed in Chapter 3. While this technology has greatly increased the productivity of researchers with respect to the amount of information generated per experiment, it presents statistical challenges that are detailed in Chapter 4. Chapters 4 and 5 discussed a series of statistical and machine learning techniques that were used to explore the relationship between genomic data types, and their limitations. The remainder of this dissertation was devoted to the development and use of the Magellan application.

Magellan was developed to allow researchers at the UCSF Cancer Center and elsewhere to analyze the varied data types that they are currently generating. As discussed in Chapter 6, the Magellan application was designed to store and analyze heterogeneous biological data and annotations. The generalized means of representing data and annotations allows for a broad utility, while the modular deployment of the analytical methods insures extensibility of the system. Chapter 7 covered the

[illegible]

implementation of the Magellan application as a server sided web application using JSP technology to generated dynamic web content, an Oracle database to store all information, and varied analytical applications to perform the actual computation. Chapter 8 provided an example of the use of Magellan on an ovarian tumor data set containing comparative genomic hybridization, mRNA gene expression, and clinical outcomes. This dissertation concluded with a discussion on the integration of Magellan with the Cancer Bioinformatic Grid (caBIG) project at the National Cancer Institute.

The technological advances discussed in this document have made the field of molecular biology much more quantitative over the last ten years, and this trend is expected to continue. It is clear that the human brain quickly becomes overwhelmed by the amount of information currently being generated, so applications such as Magellan will be increasingly important in order for users to draw meaningful conclusions from their data. It is hoped that such advances in data generation and analysis will not only increase our understanding of biological systems, but also help to alleviate the suffering caused by diseases such as cancer.

Because of the integration of Magellan within the caBIG project, its future development and impact on cancer research is reasonably well assured. The dissertation work reported here has delivered a functional, scalable, documented, and demonstrably useful system for the analysis of complex, very high-dimensional data of multiple types that represent the future of cancer research and broader biomedical inquiry.

Bibliography

Alberts, B., A. Johnson, et al. (2002). Molecular Biology of the Cell. New York, Garland Science.

Albertson, D. G., C. Collins, et al. (2003). "Chromosome aberrations in solid tumors." Nat Genet **34**(4): 369-76.

Ashburner, M., C. A. Ball, et al. (2000). "Gene ontology: tool for the unification of biology. The Gene Ontology Consortium." Nat Genet **25**(1): 25-9.

Baggerly, K. and K. Coombes (2004). Introduction to Microarrays.

Benjamini, Y. and Y. Hochberg (1995). "Controlling the false discovery rate - a practical and powerful approach to multiple testing." J.R. Stat. Soc. Ser. B **57**: 289-300.

Bettuzzi, S., P. Davalli, et al. (2000). "Tumor progression is accompanied by significant changes in the levels of expression of polyamine metabolism regulatory genes and clusterin (sulfated glycoprotein 2) in human prostate cancer specimens." Cancer Res **60**(1): 28-34.

Brazma, A., P. Hingamp, et al. (2001). "Minimum information about a microarray experiment (MIAME)-toward standards for microarray data." Nat Genet **29**(4): 365-71.

Buck, M. J. and J. D. Lieb (2004). "ChIP-chip: considerations for the design, analysis, and application of genome-wide chromatin immunoprecipitation experiments." Genomics **83**(3): 349-60.

Cahill, D. P., C. Lengauer, et al. (1998). "Mutations of mitotic checkpoint genes in human cancers." Nature **392**(6673): 300-3.

Ciardiello, F., R. Caputo, et al. (2000). "Antitumor effect and potentiation of cytotoxic drugs activity in human cancer cells by ZD-1839 (Iressa), an epidermal growth factor receptor-selective tyrosine kinase inhibitor." Clin Cancer Res **6**(5): 2053-63.

DeRisi, J. L., V. R. Iyer, et al. (1997). "Exploring the metabolic and genetic control of gene expression on a genomic scale." Science **278**(5338): 680-6.

Edwards, B. K., M. L. Brown, et al. (2005). "Annual report to the nation on the status of cancer, 1975-2002, featuring population-based trends in cancer treatment." J Natl Cancer Inst **97**(19): 1407-27.

Goldenberg, M. M. (1999). "Trastuzumab, a recombinant DNA-derived humanized monoclonal antibody, a novel agent for the treatment of metastatic breast cancer." Clin Ther **21**(2): 309-18.

Golub, T. R., D. K. Slonim, et al. (1999). "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring." Science **286**(5439): 531-7.

Hahn, W. C., C. M. Counter, et al. (1999). "Creation of human tumour cells with defined genetic elements." Nature **400**(6743): 464-8.

Hanahan, D. and R. A. Weinberg (2000). "The hallmarks of cancer." Cell **100**(1): 57-70.

Harris, C. C. (1996). "p53 tumor suppressor gene: from the basic research laboratory to the clinic--an abridged historical perspective." Carcinogenesis **17**(6): 1187-98.

Hartwell, L. H., J. Culotti, et al. (1974). "Genetic control of the cell division cycle in yeast." Science **183**(120): 46-51.

Hodgson, G., J. H. Hager, et al. (2001). "Genome scanning with array CGH delineates regional alterations in mouse islet carcinomas." Nat Genet **29**(4): 459-64.

Jain, A. N., K. Chin, et al. (2001). "Quantitative analysis of chromosomal CGH in human breast tumors associates copy number abnormalities with p53 status and patient survival." Proc Natl Acad Sci U S A **98**(14): 7952-7.

Jain, A. N., T. A. Tokuyasu, et al. (2002). "Fully automatic quantification of microarray image data." Genome Res **12**(2): 325-32.

Kingsley, C. B., W. L. Kuo, et al. (2005). "Magellan: A Web Based System for the Integrated Analysis of Heterogeneous Biological Data and Annotations; Application to DNA Copy Number and Expression Data in Ovarian Cancer." Cancer Informatics **In Press**.

Kinzler, K. W. and B. Vogelstein (1996). "Lessons from hereditary colorectal cancer." Cell **87**(2): 159-70.

Knudson, A. G., Jr., H. W. Hethcote, et al. (1975). "Mutation and childhood cancer: a probabilistic model for the incidence of retinoblastoma." Proc Natl Acad Sci U S A **72**(12): 5116-20.

Lancaster, J. M., H. K. Dressman, et al. (2004). "Gene expression patterns that characterize advanced stage serous ovarian cancers." J Soc Gynecol Investig **11**(1): 51-9.

Lengauer, C., K. W. Kinzler, et al. (1997). "Genetic instability in colorectal cancers." Nature **386**(6625): 623-7.

LLC, V. L. (2005). GeneSifter, VizX Labs LLC.

Miki, Y., J. Swensen, et al. (1994). "A strong candidate for the breast and ovarian cancer susceptibility gene BRCA1." Science **266**(5182): 66-71.

Morgan, D. O. (1997). "Cyclin-dependent kinases: engines, clocks, and microprocessors." Annu Rev Cell Dev Biol **13**: 261-91.

Nadkarni, P. M., L. Marenco, et al. (1999). "Organization of heterogeneous scientific data using the EAV/CR representation." J Am Med Inform Assoc **6**(6): 478-93.

Nannya, Y., M. Sanada, et al. (2005). "A robust algorithm for copy number detection using high-density oligonucleotide single nucleotide polymorphism genotyping arrays." Cancer Res **65**(14): 6071-9.

Nowell, P. C. (1976). "The clonal evolution of tumor cell populations." Science **194**(4260): 23-8.

Olshen, A. B. and A. N. Jain (2002). "Deriving quantitative conclusions from microarray expression data." Bioinformatics **18**(7): 961-70.

Parada, L. F., C. J. Tabin, et al. (1982). "Human EJ bladder carcinoma oncogene is homologue of Harvey sarcoma virus ras gene." Nature **297**(5866): 474-8.

Pinkel, D., R. Segraves, et al. (1998). "High resolution analysis of DNA copy number variation using comparative genomic hybridization to microarrays." Nat Genet **20**(2): 207-11.

Pollack, J. R., T. Sorlie, et al. (2002). "Microarray analysis reveals a major direct role of DNA copy number alteration in the transcriptional program of human breast tumors." Proc Natl Acad Sci U S A **99**(20): 12963-8.

Rajagopalan, H. and C. Lengauer (2004). "Aneuploidy and cancer." Nature **432**(7015): 338-41.

Ross, D. T., U. Scherf, et al. (2000). "Systematic variation in gene expression patterns in human cancer cell lines." Nat Genet **24**(3): 227-35.

Segal, M. R., K. D. Dahlquist, et al. (2003). "Regression approaches for microarray data analysis." J Comput Biol **10**(6): 961-80.

Shayesteh, L., Y. Lu, et al. (1999). "PIK3CA is implicated as an oncogene in ovarian cancer." Nat Genet **21**(1): 99-102.

Shen-Ong, G. L., E. J. Keath, et al. (1982). "Novel myc oncogene RNA from abortive immunoglobulin-gene recombination in mouse plasmacytomas." Cell **31**(2 Pt 1): 443-52.

Snijders, A. M., N. Nowak, et al. (2001). "Assembly of microarrays for genome-wide measurement of DNA copy number." Nat Genet **29**(3): 263-4.

Spector, D. H., K. Smith, et al. (1978). "Uninfected avian cells contain RNA related to the transforming gene of avian sarcoma viruses." Cell **13**(2): 371-9.

Spellman, P. T., M. Miller, et al. (2002). "Design and implementation of microarray gene expression markup language (MAGE-ML)." Genome Biol **3**(9): RESEARCH0046.

Storey, J. D. and R. Tibshirani (2003). "Statistical significance for genomewide studies." Proc Natl Acad Sci U S A **100**(16): 9440-5.

Tibshirani, R., T. Hastie, et al. (2002). "Diagnosis of multiple cancer types by shrunken centroids of gene expression." Proc Natl Acad Sci U S A **99**(10): 6567-72.

Tusher, V. G., R. Tibshirani, et al. (2001). "Significance analysis of microarrays applied to the ionizing radiation response." Proc Natl Acad Sci U S A **98**(9): 5116-21.

van 't Veer, L. J., H. Dai, et al. (2002). "Gene expression profiling predicts clinical outcome of breast cancer." Nature **415**(6871): 530-6.

Wang, W. L., M. E. Healy, et al. (2000). "Growth inhibition and modulation of kinase pathways of small cell lung cancer cell lines by the novel tyrosine kinase inhibitor STI 571." Oncogene **19**(31): 3521-8.

Westfall, P. H. and S. S. Young (1993). Resampling-based Multiple Testing, Wiley, New York.

Appendix

10.1. Distribution of Data and Source Code

The distribution of the Magellan application contains several parts:

1. The Magellan source code as an Eclipse project
2. The ovarian tumor data set described in this thesis
3. Installation instructions

In the Magellan archive to be publicly distributed, these three components are located in the 'magellan', 'data', and 'intallation' directories, respectively.

10.2. Javadocs

The following javadocs document the Java business logic contained in the compiled Java classes of the Magellan system. The API described in these javadocs should be sufficient to allow external developers to add analytical components to the Magellan system.

edu.ucsf.Magellan
Class AnalysisInfo

```
java.lang.Object
  └─ edu.ucsf.Magellan.AnalysisInfo
```

```
public class AnalysisInfo
extends java.lang.Object
```

The AnalysisInfo object provides the functionality for the analytical portion of Magellan. The object stores a number of parameters neccessary for analysis, including pointers to the dataType objects that are available for analysis from the currently selected experiment, as well as pointers to the dataType objects that have been selected and are to be outputted to a text file prior to analysis. It also provides methods to output the flat files containing the data and annotations needed for a particular analysis. Two important vectors of dataType objects are stored: availableDataTypes - data types that are available to be analyzed. For example, if a user selects a particular experiment for analysis, this vector would contain pointers to all the dataType objects that make up that experiment. outputDataTypes - the data types to be outputted for the current analysis. For example, if two data types are to be correlated then this vector would contain pointers to only those two dataType objects.

Field Summary	
private java.util.Vector	**availableDataTypes**
private java.lang.String	**dataFileAppendString**
private java.util.Vector	**execStatements**
private int	**experimentNumber**
private java.util.Vector	**outputDataTypes**
private java.util.Hashtable	**parameters**
private java.util.Vector	**resultFiles**
private java.util.Vector	**sampleList**
private java.util.Date	**startTime**

Constructor Summary
AnalysisInfo()   Contructor

Method Summary	
void	**addExperimentToAvailableDataTypes**(int experiment) Adds data types from a specified experiment to the list of data types availble for analysis
void	**addResultFile**(java.lang.String fileName) Adds a file name to the vector containing the file names that will hold the results of the current analysis.
void	**addSamplesFromExperimentToList**(int experiment) Adds Sample objects for all samples in the passed experiment number to the list of samples for the current analysis
void	**addSamplesFromUploadToList**(int uploadID) Adds Sample objects for all samples in the passed upload number to the list of samples for the current analysis

void	**addToAvailableDataTypes**(DataType dt) Adds a data type object to the list of data types available for analysis.
void	**addToOutputDataTypes**(DataType dt) Adds a data type object to the list of data types to be outputted for analysis.
void	**addToOutputDataTypes**(java.lang.String[] indices) Adds the dataType objects from the list of indices to the list of data types to be outputted for analysis.
boolean	**annotationsToOutput**() Returns true if any annotations associated with any of the data types to be outputted are to be outputted themselves.
void	**clearAnnotationOutput**() No annotations associated with any data type will be outputted as part of the current analysis.
void	**clearAvailableDataTypes**() Clears the list of dataType objects available for analysis.
void	**clearOutputDataTypes**() Clears the list of dataType objects to be outputted for analysis.
void	**clearResultFiles**() Clears the vector containing the file names that will hold the results of the current analysis.
void	**clearSampleList**() Removes all the sample objects from the list of samples for the current analysis
void	**clearSampleOutputToDataFile**() No Sample objects in the list of samples for the current analysis will be outputted.
java.lang.Object	**clone**() Clones an AnalysisInfo object (deep copy).
java.util.Vector	**dataTypeNames**() Returns a vector of data type names of the dataType objects available for analysis
java.lang.String	**dataTypeNumberString**() Returns a string containing a comma delimited list of data type numbers of the dataType objects to be outputted for the current analysis.
java.lang.String	**dataTypeNumberString**(int experimentNumber) Returns a string containing a comma delimited list of data type numbers of the dataType objects to be outputted from the specified experiment number.
boolean	**dataTypesFromSameExperiment**() Returns a true if all data types to be outputted are from the same experiment.
void	**deleteOldResultFiles**(java.lang.String filePath) Deletes the result files from the previous round of analysis.
DataType	**getAvailableDataType**(int position) Returns a data type object at a particular position in the list of available data types.
DataType	**getAvailableDataType**(java.lang.String position) Returns a data type object at a particular position in the list of available data types.
DataType	**getAvailableDataTypeFromExperiment**(java.lang.String name, int experiment) Returns a data type object with the name and experiment number specified
DataType	**getAvailableDataTypeFromUpload**(java.lang.String name, int uploadID) Returns a data type object with the name and upload number specified
java.util.Vector	**getAvailableDataTypes**() Returns a vector of dataType objects that are currently available for analysis.
int	**getExperimentNumber**() Returns the experiment number of the data set currently chosen for analysis.
DataType	**getOutputDataType**(int position) Returns a dataType object at the position in the list of dataType objects to be outputted for the current analysis.
DataType	**getOutputDataType**(java.lang.String position) Returns a dataType object at the position in the list of dataType objects to be outputted for the

	current analysis.
`java.util.Vector`	**`getOutputDataTypes`**`()` Returns a vector of dataType objects that are to be outputted to a flat text file for the current analysis.
`java.lang.String`	**`getParameterValue`**`(java.lang.String parameter)` Returns the analytical parameter whose value has been previously set.
`java.util.Vector`	**`getResultFiles`**`()` Returns the vector containing the file names that will hold the results of the current analysis.
`Sample`	**`getSampleFromUploadInList`**`(java.lang.String sampleName, int uploadID)` Returns a Sample object from the list of samples for the current analysis with the passed name and upload number
`java.util.Vector`	**`getSampleList`**`()` Returns a vector of sample objects corresponding to those samples used in the current analysis
`java.util.Date`	**`getStartTime`**`()` Gets the Date object corresponding to the time at which the current analysis started.
`java.lang.String`	**`javascriptAnnotationArray`**`()` Returns a string containing javascript commands that define a 2D array containing annotation information associated with the data types that are currently available for analysis.
`java.lang.String`	**`javascriptSubselectionArray`**`()` Returns a string containing javascript commands that define a 2D array containing subselection information associated with the data types that are currently available for analysis.
`void`	**`makeAnnotationFile`**`(DataType dt, java.io.FileWriter annotFile)` Makes a flat text file containing the annotations for the passed data type.
`void`	**`makeAnnotationFile`**`(java.io.FileWriter annotFile)` Makes a flat text file containing the annotations for all the data types of the current analysis.
`void`	**`makeCmAnnotationFile`**`(java.io.FileWriter annotFile)` Makes a flat text file containing annotations in cm file formate for the current analysis.
`void`	**`makeCmDataFile`**`(java.lang.String path, java.lang.String sessionID)` Makes a flat text file containing the data for the current analysis.
`void`	**`makeDataFile`**`(java.lang.String path, java.lang.String sessionID)` Makes a flat text file containing the data for the current analysis.
`void`	**`makeDataFile`**`(java.lang.String path, java.lang.String sessionID, java.lang.String dataFileAppendString)` Makes a flat text file containing the data for the current analysis.
`private void`	**`makeDataFile`**`(java.lang.String path, java.lang.String sessionID, java.lang.String dataFileAppendString, boolean cmFlag)` Makes a flat text file containing the data for the current analysis.
`private void`	**`makeDataFileWithSameExpts`**`(java.io.FileWriter dataFile)` Makes a flat text file containing the data for the current analysis.
`int`	**`numSamplesToOuput`**`()` Returns the number of samples to be outputted for the current analysis
`void`	**`outputAllSamplesToDataFile`**`()` All Sample objects in the list of samples for the current analysis will be outputted.
`void`	**`outputSamplesToDataFile`**`(java.lang.String[] indices)` Selected Sample objects in the list of samples for the current analysis will be outputted.
`private void`	**`printAnnotInfo`**`(DataType dt, Annotation annot, java.io.FileWriter annotFile)` Makes a flat text file containing the passed annotations for the passed data type.
`private void`	**`printStatsInfo`**`(DataType dt, Annotation annot, java.io.FileWriter annotFile)` Makes a flat text file containing the passed derived annotations for the passed data type.
`java.lang.String`	**`RfileHeader`**`(java.lang.String path, java.lang.String sessionID)` Returns a string that contains the R commands to specify the location of data types and annotations in the data files.
`java.lang.String`	**`sampleNameString`**`()` Returns a tab delimited list of the names of samples to be outputted for the current analysis

int[]	**sampleNumberArray**() Returns an array of integers corresponding to the indices of samples to be outputted for the current analysis
java.lang.String	**sampleNumberString**() Returns a comma delimited list of the integers corresponding to the indices of samples to be outputted for the current analysis
java.lang.String	**sampleNumberString**(int experimentNum) Returns a comma delimited list of the integers corresponding to the indices of samples from the passed experiment number to be outputted for the current analysis
java.util.Vector	**sampleNumberVector**() Returns a vector of integers corresponding to the indices of samples to be outputted for the current analysis
java.util.Vector	**sampleNumberVector**(int experimentNum) Returns a vector of integers corresponding to the indices of samples from the passed experiment number to be outputted for the current analysis
void	**setDataFileAppendString**(java.lang.String appendString) Sets a string whose contents will be appended to the data file of the current analysis.
void	**setExperimentNumber**(int experimentNum) Sets the experiment number for the current analysis.
void	**setParameterValue**(java.lang.String parameter, java.lang.String value) Sets an analytical parameter for the current round of analysis.
void	**setStartTime**() Creates and stores a Date object corresponding to the time at which the current analysis started.

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

experimentNumber

```
private int experimentNumber
```

dataFileAppendString

```
private java.lang.String dataFileAppendString
```

availableDataTypes

```
private java.util.Vector availableDataTypes
```

outputDataTypes

```
private java.util.Vector outputDataTypes
```

sampleList

```
private java.util.Vector sampleList
```

resultFiles

```
private java.util.Vector resultFiles
```

---

execStatements

```
private java.util.Vector execStatements
```

---

parameters

```
private java.util.Hashtable parameters
```

---

startTime

```
private java.util.Date startTime
```

Constructor Detail

AnalysisInfo

```
public AnalysisInfo()
```

Contructor

Method Detail

clone

```
public java.lang.Object clone()
```

Clones an AnalysisInfo object (deep copy).
Overrides:
`clone` in class `java.lang.Object`
Returns:
a deep copy of an AnalysisInfo object

---

setExperimentNumber

```
public void setExperimentNumber(int experimentNum)
```

Sets the experiment number for the current analysis.

---

getExperimentNumber

```
public int getExperimentNumber()
```

Returns the experiment number of the data set currently chosen for analysis.
Returns:
the experiment number of the current analysis

---

getResultFiles

```
public java.util.Vector getResultFiles()
```

Returns the vector containing the file names that will hold the results of the current analysis.
Returns:
a vector of strings, each containing the name of a results file for the current analysis.

---

clearResultFiles

```
public void clearResultFiles()
```

Clears the vector containing the file names that will hold the results of the current analysis.

---

addResultFile

```
public void addResultFile(java.lang.String fileName)
```

Adds a file name to the vector containing the file names that will hold the results of the current analysis.
Parameters:
`fileName` - a string containing the name of a results file (not file path) for the current analysis

---

setDataFileAppendString

```
public void setDataFileAppendString(java.lang.String appendString)
```

Sets a string whose contents will be appended to the data file of the current analysis. This is useful is a user creates some data dynamically through the web pages of an analysis session - this dynamically generated data would be appended to the end of the data file prior to its analysis.
Parameters:
`appendString` - the string to be appended to the data file for the current analysis.

---

getStartTime

```
public java.util.Date getStartTime()
```

Gets the Date object corresponding to the time at which the current analysis started.
Returns:
a Date object corresponding to the time the analysis started.

---

setStartTime

```
public void setStartTime()
```

Creates and stores a Date object corresponding to the time at which the current analysis started. The start time is set to the time when the method is called.

---

deleteOldResultFiles

```
public void deleteOldResultFiles(java.lang.String filePath)
                          throws java.lang.SecurityException
```

Deletes the result files from the previous round of analysis. This method should be called at the start of a new analysis such that the prior results are not detected and shown to the user as the results of the current round.
Parameters:
`filePath` - the full path of the folder containing the result files to be deleted prior to the next round of analysis.
Throws:
`java.lang.SecurityException`

---

setParameterValue

```
public void setParameterValue(java.lang.String parameter,
                              java.lang.String value)
```

Sets an analytical parameter for the current round of analysis. This list of parameters and values can be used to store information (such as a type of correlation statistic or a range of values to display in a graph) for a given analysis.
Parameters:
`parameter` - the type of parameter whose value will be set (example: distanceMetric)
`value` - the value of the parameter to be set (example: euclidean)

---

getParameterValue

```
public java.lang.String getParameterValue(java.lang.String parameter)
```

[illegible]

Returns the analytical parameter whose value has been previously set.
Parameters:
`parameter` - the type of parameter whose value will be retrieved
Returns:
the value of the parameter

---

clearSampleList

```
public void clearSampleList()
```
Removes all the sample objects from the list of samples for the current analysis

---

getSampleList

```
public java.util.Vector getSampleList()
```
Returns a vector of sample objects corresponding to those samples used in the current analysis
Returns:
a vector of sample objects

---

getSampleFromUploadInList

```
public Sample getSampleFromUploadInList(java.lang.String sampleName,
                                        int uploadID)
```
Returns a Sample object from the list of samples for the current analysis with the passed name and upload number
Returns:
a Sample object with the passed name from the passed uploadID

---

addSamplesFromExperimentToList

```
public void addSamplesFromExperimentToList(int experiment)
                                    throws java.sql.SQLException
```
Adds Sample objects for all samples in the passed experiment number to the list of samples for the current analysis
Parameters:
`experiment` - the experiment number
Throws:
`java.sql.SQLException`

---

addSamplesFromUploadToList

```
public void addSamplesFromUploadToList(int uploadID)
                                throws java.sql.SQLException
```
Adds Sample objects for all samples in the passed upload number to the list of samples for the current analysis
Parameters:
`uploadID` - the experiment number
Throws:
`java.sql.SQLException`

---

clearSampleOutputToDataFile

```
public void clearSampleOutputToDataFile()
```
No Sample objects in the list of samples for the current analysis will be outputted.

---

outputAllSamplesToDataFile

```
public void outputAllSamplesToDataFile()
```
All Sample objects in the list of samples for the current analysis will be outputted.

[illegible]

outputSamplesToDataFile

`public void` **`outputSamplesToDataFile`**`(java.lang.String[] indices)`

Selected Sample objects in the list of samples for the current analysis will be outputted. The array of strings is typically produced the the request.getParameter() method.

Parameters:

`indices` - an array of Strings whose values contain integers corresponding to the indices of those samples to be outputted.

---

numSamplesToOuput

`public int` **`numSamplesToOuput`**`()`

Returns the number of samples to be outputted for the current analysis

Returns:

the number of samples that will be outputted

---

sampleNumberArray

`public int[]` **`sampleNumberArray`**`()`

Returns an array of integers corresponding to the indices of samples to be outputted for the current analysis

Returns:

an array of integers containing sample indices to be outputted

---

sampleNumberVector

`public java.util.Vector` **`sampleNumberVector`**`()`

Returns a vector of integers corresponding to the indices of samples to be outputted for the current analysis

Returns:

a vector of integers containing sample indices to be outputted

---

sampleNumberVector

`public java.util.Vector` **`sampleNumberVector`**`(int experimentNum)`

Returns a vector of integers corresponding to the indices of samples from the passed experiment number to be outputted for the current analysis

Parameters:

`experimentNum` - the experiment number the samples must be from to be included in the list

Returns:

a vector of integers containing sample indices to be outputted

---

sampleNumberString

`public java.lang.String` **`sampleNumberString`**`()`

Returns a comma delimited list of the integers corresponding to the indices of samples to be outputted for the current analysis

Returns:

a comma delimited string containing the sample numbers

---

sampleNumberString

`public java.lang.String` **`sampleNumberString`**`(int experimentNum)`

Returns a comma delimited list of the integers corresponding to the indices of samples from the passed experiment number to be outputted for the current analysis

Parameters:

`experimentNum` - the experiment number from which the sample numbers are selected.

Returns:

a comma delimited string containing the sample numbers from the specified experiment

[illegible]

---

sampleNameString

```
public java.lang.String sampleNameString()
```

Returns a tab delimited list of the names of samples to be outputted for the current analysis
Returns:
a tab delimited string containing the sample names

---

addToAvailableDataTypes

```
public void addToAvailableDataTypes(DataType dt)
```

Adds a data type object to the list of data types available for analysis.
Parameters:
`dt` - a data type object to be made available for analysis

---

clearAvailableDataTypes

```
public void clearAvailableDataTypes()
```

Clears the list of dataType objects available for analysis.

---

getAvailableDataTypes

```
public java.util.Vector getAvailableDataTypes()
```

Returns a vector of dataType objects that are currently available for analysis.
Returns:
a vector of dataType objects

---

getAvailableDataType

```
public DataType getAvailableDataType(int position)
```

Returns a data type object at a particular position in the list of available data types.
Parameters:
`position` - the position in the list of the data type to be returned
Returns:
the data type object at the specified position in the list

---

getAvailableDataType

```
public DataType getAvailableDataType(java.lang.String position)
```

Returns a data type object at a particular position in the list of available data types.
Parameters:
`position` - a string containing the integer position in the list of the data type to be returned
Returns:
the data type object at the specified position in the list

---

getAvailableDataTypeFromUpload

```
public DataType getAvailableDataTypeFromUpload(java.lang.String name,
                                               int uploadID)
```

Returns a data type object with the name and upload number specified
Parameters:
`name` - the name of the data type
`uploadID` - the uploadID of the data type
Returns:
a data type object with the passed name and uploadID

[illegible]

getAvailableDataTypeFromExperiment

```
public DataType getAvailableDataTypeFromExperiment(java.lang.String name,
                                                   int experiment)
```

Returns a data type object with the name and experiment number specified
Parameters:
`name` - the name of the data type
`experiment` - the experiment number of the data type
Returns:
a data type object with the passed name and experiment number

---

addExperimentToAvailableDataTypes

```
public void addExperimentToAvailableDataTypes(int experiment)
                                       throws java.sql.SQLException
```

Adds data types from a specified experiment to the list of data types availble for analysis
Parameters:
`experiment` - the experiment number
Throws:
`java.sql.SQLException`

---

javascriptAnnotationArray

```
public java.lang.String javascriptAnnotationArray()
```

Returns a string containing javascript commands that define a 2D array containing annotation information associated with the data types that are currently available for analysis. The array is called 'annotations' in javascript - the first subscript corresponds to the position of the data type in the list of available data types, and the second corresponds to the position of the annotation in the list of annotations associated with the data type. The defined array is useful for filling drop down menus/lists of annotations when a user selects a data type in a web page.
Returns:
a string containing the javascript commands that specify the array of annotation information.

---

javascriptSubselectionArray

```
public java.lang.String javascriptSubselectionArray()
```

Returns a string containing javascript commands that define a 2D array containing subselection information associated with the data types that are currently available for analysis. The array is called 'subSelections' in javascript - the first subscript corresponds to the position of the data type in the list of available data types, and the second corresponds to the position of the subselection in the list of subselections associated with the data type. The defined array is useful for filling drop down menus/lists of subselections when a user selects a data type in a web page.
Returns:
a string containing the javascript commands that specify the array of subselection information.

---

clearOutputDataTypes

```
public void clearOutputDataTypes()
```

Clears the list of dataType objects to be outputted for analysis.

---

addToOutputDataTypes

```
public void addToOutputDataTypes(DataType dt)
```

Adds a data type object to the list of data types to be outputted for analysis.
Parameters:
`dt` - a data type object to be outputted for analysis

[illegible]

addToOutputDataTypes

`public void addToOutputDataTypes(java.lang.String[] indices)`

Adds the dataType objects from the list of indices to the list of data types to be outputted for analysis. The indices are stringified integers present in an array, such as the arrays returned by the request.getParameter() method. The integer value is the position of the data type in the list of available data types.

Parameters:

`indices` - an array of strings, each of contains an integer.

---

getOutputDataTypes

`public java.util.Vector getOutputDataTypes()`

Returns a vector of dataType objects that are to be outputted to a flat text file for the current analysis.

Returns:

a vector of dataType objects to be outputted.

---

getOutputDataType

`public DataType getOutputDataType(int position)`

Returns a dataType object at the position in the list of dataType objects to be outputted for the current analysis.

Parameters:

`position` - an integer corresponding to the position in the list of outputted data types

Returns:

a dataType object at the specified position

---

getOutputDataType

`public DataType getOutputDataType(java.lang.String position)`

Returns a dataType object at the position in the list of dataType objects to be outputted for the current analysis.

Parameters:

`position` - a stringified integer corresponding to the position in the list of outputted data types

Returns:

a dataType object at the specified position

---

clearAnnotationOutput

`public void clearAnnotationOutput()`

No annotations associated with any data type will be outputted as part of the current analysis.

---

annotationsToOutput

`public boolean annotationsToOutput()`

Returns true if any annotations associated with any of the data types to be outputted are to be outputted themselves.

Returns:

a boolean variable set to true if any annotations are to be outputted.

---

dataTypeNames

`public java.util.Vector dataTypeNames()`

Returns a vector of data type names of the dataType objects available for analysis

Returns:

a vector a strings containing data type names.

---

dataTypeNumberString

[illegible]

```
public java.lang.String dataTypeNumberString()
```
Returns a string containing a comma delimited list of data type numbers of the dataType objects to be outputted for the current analysis.
Returns:
a string of comma delimited data type numbers.

---

dataTypeNumberString

```
public java.lang.String dataTypeNumberString(int experimentNumber)
```
Returns a string containing a comma delimited list of data type numbers of the dataType objects to be outputted from the specified experiment number.
Parameters:
`experimentNumber` - the experiment number of the data types whose data type numbers are to be returned in the string.
Returns:
a string of comma delimited data type numbers.

---

dataTypesFromSameExperiment

```
public boolean dataTypesFromSameExperiment()
```
Returns a true if all data types to be outputted are from the same experiment.
Returns:
a boolean variable that is true if all data types to be outputted are from the same experiment.

RfileHeader

```
public java.lang.String RfileHeader(java.lang.String path,
                                    java.lang.String sessionID)
```
Returns a string that contains the R commands to specify the location of data types and annotations in the data files. These R command load the contents of the data file into R objects such that these objects can be analyzed by an R application.
Parameters:
`path` - the full path of the folder in which the data file is stored.
`sessionID` - the session ID of the current web session.
Returns:
a string of R commands that partition data types and annotations in the data file into R objects.

---

makeDataFile

```
public void makeDataFile(java.lang.String path,
                         java.lang.String sessionID)
                  throws java.io.IOException,
                         java.sql.SQLException
```
Makes a flat text file containing the data for the current analysis. The first row contains a tab delimited list of sample names. Subsequent rows contain the data for the variables of each data type. The first column contains the identifiers for each data type. If annotations are to be utilized for this analysis, a flat text file containing annotations will also be generated.
Parameters:
`path` - the full path of the folder in which the data file is stored.
`sessionID` - the session ID of the current web session.
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

makeCmDataFile

```
public void makeCmDataFile(java.lang.String path,
                           java.lang.String sessionID)
                    throws java.io.IOException,
                           java.sql.SQLException
```
Makes a flat text file containing the data for the current analysis. The first row contains a tab delimited list of sample names. Subsequent rows contain the data for the variables of each data type. The first column contains the identifiers for

[illegible]

each data type. If annotations are to be utilized for this analysis, a flat text file containing annotations will also be generated in the format used by the cm application.
Parameters:
`path` - the full path of the folder in which the data file is stored.
`sessionID` - the session ID of the current web session.
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

makeDataFile

```
public void makeDataFile(java.lang.String path,
                         java.lang.String sessionID,
                         java.lang.String dataFileAppendString)
                  throws java.io.IOException,
                         java.sql.SQLException
```

Makes a flat text file containing the data for the current analysis. The first row contains a tab delimited list of sample names. Subsequent rows contain the data for the variables of each data type. The first column contains the identifiers for each data type. If annotations are to be utilized for this analysis, a flat text file containing annotations will also be generated. The string 'dataFileAppendString' is added to the end of the data file, in case custom data is to be added to the file based upon user input during the analysis.
Parameters:
`path` - the full path of the folder in which the data file is stored.
`sessionID` - the session ID of the current web session.
`dataFileAppendString` - a string to be added at the end of the data file
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

makeDataFile

```
private void makeDataFile(java.lang.String path,
                          java.lang.String sessionID,
                          java.lang.String dataFileAppendString,
                          boolean cmFlag)
                   throws java.io.IOException,
                          java.sql.SQLException
```

Makes a flat text file containing the data for the current analysis. The first row contains a tab delimited list of sample names. Subsequent rows contain the data for the variables of each data type. The first column contains the identifiers for each data type. If annotations are to be utilized for this analysis, a flat text file containing annotations will also be generated. The string 'dataFileAppendString' is added to the end of the data file, in case custom data is to be added to the file based upon user input during the analysis.
Parameters:
`path` - the full path of the folder in which the data file is stored.
`sessionID` - the session ID of the current web session.
`dataFileAppendString` - a string to be added at the end of the data file
`cmFlag` - true if the cm format is to be used for the annotation file
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

makeDataFileWithSameExpts

```
private void makeDataFileWithSameExpts(java.io.FileWriter dataFile)
                                throws java.io.IOException,
                                       java.sql.SQLException
```

Makes a flat text file containing the data for the current analysis. The first row contains a tab delimited list of sample names. Subsequent rows contain the data for the variables of each data type. The first column contains the identifiers for each data type.
Parameters:
`dataFile` - the FileWriter object to which the data is written
Throws:
`java.io.IOException`

java.sql.S

makeAnnotationFile

public void **makeAn**

Makes a flat te
delimited list o
column contai
Parameters:
annotFile -
Throws:
java.io.IO
java.sql.S

makeAnnotationFile

public void **makeAn**

Makes a flat te
annotation typ
identifiers of t
Parameters:
dt - the data
annotFile
Throws:
java.io.I
java.sql.

printAnnotInfo

private void pr

Makes a fl
of annota
contains t
Paramet
dt - the
annot -
annotF
Throws
java.
java.

printStatsInfo

private voi

Mak
deli
type
Par
dt

```
java.sql.SQLException
```

---

makeAnnotationFile

```
public void makeAnnotationFile(java.io.FileWriter annotFile)
                        throws java.io.IOException,
                               java.sql.SQLException
```

Makes a flat text file containing the annotations for all the data types of the current analysis. The first row contains a tab delimited list of annotation types. Subsequent rows contain the annotations for the variables of each data type. The first column contains the identifiers for each data type.

Parameters:

annotFile - the FileWriter object to which the annotation information is written

Throws:

java.io.IOException
java.sql.SQLException

---

makeAnnotationFile

```
public void makeAnnotationFile(DataType dt,
                               java.io.FileWriter annotFile)
                        throws java.io.IOException,
                               java.sql.SQLException
```

Makes a flat text file containing the annotations for the passed data type. The first row contains a tab delimited list of annotation types. Subsequent rows contain the annotations for the variables of the data type. The first column contains the identifiers of the data type.

Parameters:

dt - the data type object whose annotations are to be outputted
annotFile - the FileWriter object to which the annotation information is written

Throws:

java.io.IOException
java.sql.SQLException

---

printAnnotInfo

```
private void printAnnotInfo(DataType dt,
                            Annotation annot,
                            java.io.FileWriter annotFile)
                     throws java.io.IOException,
                            java.sql.SQLException
```

Makes a flat text file containing the passed annotations for the passed data type. The first row contains a tab delimited list of annotation types. Subsequent rows contain the specified annotations for the variables of the data type. The first column contains the identifiers of the data type.

Parameters:

dt - the data type object whose annotations are to be outputted
annot - the annotation object specifying the annotations to be outputted to the flat file
annotFile - the FileWriter object to which the annotation information is written

Throws:

java.io.IOException
java.sql.SQLException

---

printStatsInfo

```
private void printStatsInfo(DataType dt,
                            Annotation annot,
                            java.io.FileWriter annotFile)
                     throws java.io.IOException,
                            java.sql.SQLException
```

Makes a flat text file containing the passed derived annotations for the passed data type. The first row contains a tab delimited list of annotation types. Subsequent rows contain the specified derived annotations for the variables of the data type. The first column contains the identifiers of the data type.

Parameters:

dt - the data type object whose annotations are to be outputted

[illegible]

annot - the annotation object specifying the annotations to be outputted to the flat file
annotFile - the FileWriter object to which the annotation information is written
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

makeCmAnnotationFile

```
public void makeCmAnnotationFile(java.io.FileWriter annotFile)
                         throws java.io.IOException,
                                java.sql.SQLException
```

Makes a flat text file containing annotations in cm file formate for the current analysis. Annotations are written to the flat file for each data type that is to be analyzed.Each row contains a tab delimited list containing the data type name, ordinal position, annotation type number, and annotation value.
Parameters:
annotFile - the FileWriter object to which the annotation information is written
Throws:
`java.io.IOException`
`java.sql.SQLException`

---

edu.ucsf.Magellan
Class AnalysisThread

```
java.lang.Object
  └─ edu.ucsf.Magellan.AnalysisThread
```

All Implemented Interfaces:
java.lang.Runnable

```
public class AnalysisThread
extends java.lang.Object
implements java.lang.Runnable
```

The AnalysisThread object allows users to create and run threads for the analysis of data and annotations. Users can create subclasses of AnalysisThread in their web pages such that all the different parts of an analysis (creating data and annotation files, running analytical methods, file cleanup, etc.) can be put in a single thread and run.

Field Summary	
protected AnalysisInfo	**Analysis**
protected javax.servlet.ServletContext	**application**
protected boolean	**running**
protected java.lang.String	**sessionID**

Constructor Summary
AnalysisThread()

Method Summary	
boolean	**isRunning**()

void	run()
void	setRunning(boolean running)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

Analysis

protected AnalysisInfo **Analysis**

application

protected javax.servlet.ServletContext **application**

sessionID

protected java.lang.String **sessionID**

running

protected boolean **running**

Constructor Detail

AnalysisThread

public **AnalysisThread**()

Method Detail

setRunning

public void **setRunning**(boolean running)

isRunning

public boolean **isRunning**()

run

public void **run**()
Specified by:
run in interface java.lang.Runnable

edu.ucsf.Magellan
Class Annotation

```
java.lang.Object
  └─ edu.ucsf.Magellan.Annotation
```

All Implemented Interfaces:
java.util.EventListener, javax.servlet.http.HttpSessionBindingListener

```
public class Annotation
extends java.lang.Object
implements javax.servlet.http.HttpSessionBindingListener
```

The Annotation object provides properties and methods for the analysis of curated or derived biological annotation information. Annotations can be generally defined as quantitative or qualitative that describe the variables of a data type. For example, the variables of a data set of mRNA expression data are the genes themselves and annotations would describe those genes (chromosomal position, pathway designation, enzymatic activity, correlation with a clinical parameter, etc). Annotation objects are linked to the Data Type objects they describe. Annotations are linked to individual variables of a data type via 'identifiers'. Identifiers are names that are used to label data type variables. Identifiers include names such as genbank Id's, Unigene Id's, etc.

Field Summary	
private int	**annotClass**
private java.util.Vector	**annotTypes**
static int	**CURATED**
static int	**DATABASE**
static int	**DERIVED**
private java.lang.String	**description**
private java.lang.String	**filePath**
private java.lang.String	**name**
private boolean	**outputToAnnotationFile**
static int	**PASTED**
private java.lang.String[][]	**pastedValues**
private int	**uploadID**
private int	**valueSource**

Constructor Summary
Annotation()
Annotation(int uploadID)

Method Summary

void	**addAnnotationType**(java.lang.String annotType) Adds a string to the vector of annotation types for the current annotation set.
java.lang.Object	**clone**() Performs a deep copy of an Annotation object.
boolean	**equals**(java.lang.Object obj)
int	**getAnnotationClass**() Returns the annotation class.
java.util.Vector	**getAnnotationTypes**() Returns a vector containing strings of the annotation types for the annotation set.
java.lang.String	**getDescription**() Returns the annotation description
java.lang.String	**getGenomicDistanceSelectLists**() Returns a string containing an HTML select list which allows user to specify which annotation type of the current annotation set contains chromosome information and distance from p telomere information, as well as the scale of the distance (bp or kb).
java.lang.String	**getGenomicDistanceSelectLists**(java.lang.String chromName, java.lang.String distName, java.lang.String scaleName) Returns a string containing an HTML select list which allows user to specify which annotation type of the current annotation set contains chromosome information and distance from p telomere information, as well as the scale of the distance (bp or kb).
static java.util.Vector	**getIdentifierTypes**(int uploadID) Returns a vector containing strings of the annotation types for the annotation set of the passed upload ID number.
java.lang.String	**getName**() Returns the name of the annotation
java.lang.String[][]	**getPastedValues**() Returns annotation values that the user has pasted into a web page.
int	**getUploadID**() Returns the upload ID number for this annotation set.
int	**getValueSource**() Returns a coded integer corresponding to the source of the annotation values.
java.lang.String	**infoString**() Returns a string containing HTML code containing a list of properties of the current annotation information.
boolean	**outputToAnnotationFile**() Returns true if annotations are to be outputted to a flat text file for analysis.
void	**setAnnotationClass**(int annotClass) Sets the class of the annotation.
void	**setDescription**(java.lang.String description) Sets the description of the annotation
void	**setName**(java.lang.String annotationName) Sets the name of the annotation
void	**setOutputToAnnotationFile**(boolean output) Sets whether or not annotations are to be outputted to a flat text file for analysis.
void	**setPastedValues**(java.lang.String[][] annotValues) Sets the description of the annotation
void	**setUploadID**(int uploadID) Sets the upload ID number for this annotation set.
void	**setValueSource**(int source) Sets a coded integer corresponding to the source of the annotation values.
void	**valueBound**(javax.servlet.http.HttpSessionBindingEvent event) Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface

Methods inherited from

`finalize, getClas`

Field Detail

DATABASE

`public static fin`

See Also:

Constant Fiel

PASTED

`public static fin`

See Also:

Constant Fiel

CURATED

`public static fin`

See Also:

Constant Fie

DERIVED

`public static fin`

See Also:

Constant Fie

name

`private java.lan`

description

`private java.lan`

filePath

`private java.lan`

pastedValues

`private java.lan`

outputToAnnotationFi

void	**valueUnbound**(javax.servlet.http.HttpSessionBindingEvent event) Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface

Methods inherited from class java.lang.Object
finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

DATABASE

`public static final int DATABASE`
See Also:
Constant Field Values

PASTED

`public static final int PASTED`
See Also:
Constant Field Values

CURATED

`public static final int CURATED`
See Also:
Constant Field Values

DERIVED

`public static final int DERIVED`
See Also:
Constant Field Values

---

name

`private java.lang.String name`

---

description

`private java.lang.String description`

---

filePath

`private java.lang.String filePath`

---

pastedValues

`private java.lang.String[][] pastedValues`

---

outputToAnnotationFile

```
private boolean outputToAnnotationFile
```

---

uploadID

```
private int uploadID
```

---

valueSource

```
private int valueSource
```

---

annotClass

```
private int annotClass
```

---

annotTypes

```
private java.util.Vector annotTypes
```

Constructor Detail

Annotation

```
public Annotation()
```

Annotation

```
public Annotation(int uploadID)
           throws java.sql.SQLException
```

Throws:
`java.sql.SQLException`

Method Detail

getName

```
public java.lang.String getName()
```

Returns the name of the annotation
Returns:
the annotation name

---

setName

```
public void setName(java.lang.String annotationName)
```

Sets the name of the annotation
Parameters:
`annotationName` - name the annotation name

---

getAnnotationClass

```
public int getAnnotationClass()
```

Returns the annotation class. Allowable values are 'curated' and 'derived' for the two classes of annotations.
Returns:
the annotation class

---

setAnnotationClass

`public void` **`setAnnotationClass`**`(int annotClass)`

Sets the class of the annotation. Allowable values are the constants CURATED and DERIVED

Parameters:

`annotClass` - class the annotation class

---

getDescription

`public java.lang.String` **`getDescription`**`()`

Returns the annotation description

Returns:

the annotation description

---

setDescription

`public void` **`setDescription`**`(java.lang.String description)`

Sets the description of the annotation

Parameters:

`description` - the annotation description

---

getPastedValues

`public java.lang.String[][]` **`getPastedValues`**`()`

Returns annotation values that the user has pasted into a web page. These values are stored in a 2D String table.

Returns:

a String table of annotation values.

---

setPastedValues

`public void` **`setPastedValues`**`(java.lang.String[][] annotValues)`

Sets the description of the annotation

Parameters:

`annotValues` - description the annotation description

---

outputToAnnotationFile

`public boolean` **`outputToAnnotationFile`**`()`

Returns true if annotations are to be outputted to a flat text file for analysis.

Returns:

true if annotations are to be outputted

---

setOutputToAnnotationFile

`public void` **`setOutputToAnnotationFile`**`(boolean output)`

Sets whether or not annotations are to be outputted to a flat text file for analysis.

Parameters:

`output` - true if annotations are to be outputted

---

getUploadID

`public int` **`getUploadID`**`()`

Returns the upload ID number for this annotation set.

Returns:

the annotation upload ID

setUploadID

`public void setU`

Sets the upl

Parameters:

`uploadID -`

---

getValueSource

`public int getVa`

Returns a co

PASTED De

Returns:

an integer co

---

setValueSource

`public void setV`

Sets a codec

Default valu

Parameter

`source -`

---

getAnnotationTypes

`public java.ut`

Returns a

for chrom

would be

Returns:

a vector

---

addAnnotationTy

`public void a`

Adds a

Param

`annot`

---

getIdentifierTyp

`public stat`

Retur

Para

`upl`

Retu

a ve

Thr

`jav`

---

clone

`public ja`

setUploadID

```
public void setUploadID(int uploadID)
```

Sets the upload ID number for this annotation set.
Parameters:
`uploadID` - output true if annotations are to be outputted

getValueSource

```
public int getValueSource()
```

Returns a coded integer corresponding to the source of the annotation values. Allowed values are DATABASE or PASTED. Default value for a newly instantiated Annotation object is DATABASE.
Returns:
an integer corresponding to the source of the annotation values.

setValueSource

```
public void setValueSource(int source)
```

Sets a coded integer corresponding to the source of the annotation values. Allowed values are DATABASE or PASTED. Default value for a newly instantiated Annotation object is DATABASE.
Parameters:
`source` - an integer corresponding to the source of the annotation values.

getAnnotationTypes

```
public java.util.Vector getAnnotationTypes()
```

Returns a vector containing strings of the annotation types for the annotation set. For example, if annotations are uploaded for chromosomal position with the annotation types 'chromosome' and 'bp', then a vector containing 'chromosome' and 'bp' would be returned.
Returns:
a vector containing the annotation types that exist in the annotation set.

addAnnotationType

```
public void addAnnotationType(java.lang.String annotType)
```

Adds a string to the vector of annotation types for the current annotation set.
Parameters:
`annotType` - the annotation type to be added to the vector of annotation sets.

getIdentifierTypes

```
public static java.util.Vector getIdentifierTypes(int uploadID)
                                           throws java.sql.SQLException
```

Returns a vector containing strings of the annotation types for the annotation set of the passed upload ID number.
Parameters:
`uploadID` - the upload ID number of the annotation set whose annotation types are to be returned.
Returns:
a vector containing the annotation types that exist in the annotation set of the passed upload ID.
Throws:
`java.sql.SQLException`

clone

```
public java.lang.Object clone()
```

[illegible]

Performs a deep copy of an Annotation object.
Overrides:
`clone` in class `java.lang.Object`
Returns:
a cloned Annotation object.

---

equals

```
public boolean equals(java.lang.Object obj)
```
Overrides:
`equals` in class `java.lang.Object`

---

getGenomicDistanceSelectLists

```
public java.lang.String getGenomicDistanceSelectLists(java.lang.String chromName,
                                                      java.lang.String distName,
                                                      java.lang.String scaleName)
```
Returns a string containing an HTML select list which allows user to specify which annotation type of the current annotation set contains chromosome information and distance from p telomere information, as well as the scale of the distance (bp or kb). These select lists are useful for web pages of analytical methods where genomic position information is used.
Parameters:
`chromName` - the HTML select name for the chromosome information
`distName` - the HTML select name for the distance from p telomere information
`scaleName` - the HTML select name for the distance scale information information
Returns:
HTML code for a SELECT list

---

getGenomicDistanceSelectLists

```
public java.lang.String getGenomicDistanceSelectLists()
```
Returns a string containing an HTML select list which allows user to specify which annotation type of the current annotation set contains chromosome information and distance from p telomere information, as well as the scale of the distance (bp or kb). These select lists are useful for web pages of analytical methods where genomic position information is used. The HTML select names 'chrom', 'distance', 'scale' are used in the HTML code.
Returns:
HTML code for a SELECT list

---

infoString

```
public java.lang.String infoString()
```
Returns a string containing HTML code containing a list of properties of the current annotation information. The upload ID number, description, and a comma delimited list of annotation types are listed.
Returns:
HTML code listing information for the current annotation set.

---

valueBound

```
public void valueBound(javax.servlet.http.HttpSessionBindingEvent event)
```
Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface
Specified by:
`valueBound` in interface `javax.servlet.http.HttpSessionBindingListener`
Parameters:
`event` - the HttpSessionBindingEvent passed by the expired session to the method

---

valueUnbound

[illegible]

public void **valueUnbound**(javax.servlet.http.HttpSessionBindingEvent event)

Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface

Specified by:

valueUnbound in interface javax.servlet.http.HttpSessionBindingListener

Parameters:

event - the HttpSessionBindingEvent passed by the expired session to the method

edu.ucsf.Magellan
Class AnnotationComparison

```
java.lang.Object
  └─ edu.ucsf.Magellan.AnnotationComparison
```

```
public class AnnotationComparison
extends java.lang.Object
```

The AnnotationComparison object provides properties and methods for the comparison of annotations or identifiers as part of projecting one data type onto another. Annotations/Identifiers can be compared in a number of quantitative and qualitative ways (string comparisons, numerical comparisons, etc.). AnnotationComparison objects store what kind of information is to be compared (identifiers vs annotations as well as the annotation type), and the type of comparison to be made. The comparison itself is performed by the compare() method.

Field Summary	
private Annotation[]	annotations
static int	ANNOTATIONS
private int[]	annotTypeNumbers
static int	CASE_INS_STRING_EQUALITY
private int	comparisonType
static int	EQUALS
static int	GREATER
static int	GREATER_OR_EQUAL
static int	IDENTIFIERS
private int[]	infoSource
static int	LESS
static int	LESS_OR_EQUAL
static int	STRING_EQUALITY
private double	threshold
static int	WITHIN_THRESHOLD

[illegible]

Constructor Summary
`AnnotationComparison()`

Method Summary	
`java.lang.Object`	`clone()` Returns a clone of the AnnotationComparison object.
`boolean`	`compare(java.lang.String value1, java.lang.String value2)` Returns true if the passed string values meet the comparison criteria established by the AnnotationComparison object.
`boolean`	`equals(java.lang.Object obj)` Returns true if the two AnnotationComparison objects are equal.
`Annotation[]`	`getAnnotations()` Returns the array of Annotation objects that are used for the annotation comparison.
`int[]`	`getAnnotationTypeNumbers()` Returns the array of annotation type numbers that are used for the annotation comparison.
`java.lang.String`	`getComparisonString()` Returns a brief string explanation of the comparison type
`int`	`getComparisonType()` Returns the integer value corresponding to the comparison type
`int`	`getInfoSource(int index)` Returns the information type for the comparison at the specified index - 'annotations' or 'identifiers'
`java.lang.String`	`getInfoSourceString(int index)` Returns a string for the type of information to be compared.
`double`	`getThreshold()` Returns the threshold for 'within' comparisons.
`void`	`setComparisonType(int type)` Sets the integer value corresponding to the comparison type: STRING_EQUALITY: string equality CASE_INS_STRING_EQUALITY: case insensitive string equality EQUALS: = GREATER: > GREATER_OR_EQUAL: >= LESS: < LESS_OR_EQUAL: <= WITHIN_THRESHOLD: within a threshold of each other - abs(a-b)<=threshold
`void`	`setInfoSource(int index, int sourceType)` Sets the information source (ANNOTATIONS or IDENTIFIERS) for the comparison at the specified index.
`void`	`setInfoType(DataType dt, java.lang.String infoTypeNumber, int index)` Sets the type of information that is to be used to compare annotations or identifiers at the specified index.
`void`	`setThreshold(double threshold)` Sets the double value corresponding the threshold used for 'within' comparisons.

Methods inherited from class java.lang.Object
`finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

STRING_EQUALITY

`public static final int STRING_EQUALITY`

See Also:

Constant Field Values

CASE_INS_STRING_EQUALITY

[illegible]

```
public static final int CASE_INS_STRING_EQUALITY
```
See Also:
Constant Field Values

EQUALS

```
public static final int EQUALS
```
See Also:
Constant Field Values

GREATER

```
public static final int GREATER
```
See Also:
Constant Field Values

GREATER_OR_EQUAL

```
public static final int GREATER_OR_EQUAL
```
See Also:
Constant Field Values

LESS

```
public static final int LESS
```
See Also:
Constant Field Values

LESS_OR_EQUAL

```
public static final int LESS_OR_EQUAL
```
See Also:
Constant Field Values

WITHIN_THRESHOLD

```
public static final int WITHIN_THRESHOLD
```
See Also:
Constant Field Values

IDENTIFIERS

```
public static final int IDENTIFIERS
```
See Also:
Constant Field Values

ANNOTATIONS

```
public static final int ANNOTATIONS
```
See Also:
Constant Field Values

[illegible]

comparisonType

`private int` **`comparisonType`**

---

threshold

`private double` **`threshold`**

---

infoSource

`private int[]` **`infoSource`**

---

annotations

`private Annotation[]` **`annotations`**

---

annotTypeNumbers

`private int[]` **`annotTypeNumbers`**

Constructor Detail

AnnotationComparison

`public` **`AnnotationComparison`**`()`

Method Detail

setComparisonType

`public void` **`setComparisonType`**`(int type)`

Sets the integer value corresponding to the comparison type: STRING_EQUALITY: string equality CASE_INS_STRING_EQUALITY: case insensitive string equality EQUALS: = GREATER: > GREATER_OR_EQUAL: >= LESS: < LESS_OR_EQUAL: <= WITHIN_THRESHOLD: within a threshold of each other - abs(a-b)<=threshold

Parameters:

`type` - the type number of the comparison

---

getComparisonType

`public int` **`getComparisonType`**`()`

Returns the integer value corresponding to the comparison type

Returns:

the type number of the comparison operator

---

getComparisonString

`public java.lang.String` **`getComparisonString`**`()`

Returns a brief string explanation of the comparison type

Returns:

a brief string explanation of the comparison type

---

setThreshold

```
public void setThreshold(double threshold)
```

Sets the double value corresponding the threshold used for 'within' comparisons. In this type of comparison, numbers are tested to be within a threshold level of each other

Parameters:

`threshold` - the threshold for 'within' comparisons.

---

getThreshold

```
public double getThreshold()
```

Returns the threshold for 'within' comparisons.

Returns:

the threshold for 'within' comparisons.

---

setInfoType

```
public void setInfoType(DataType dt,
                        java.lang.String infoTypeNumber,
                        int index)
```

Sets the type of information that is to be used to compare annotations or identifiers at the specified index.

Parameters:

`dt` - the dataType object that is to be projected from or onto

`infoTypeNumber` - a string containing the value '-1' if identifiers are to be compared, or two tab delimited numbers - the first is the index of the annotation object in the list of annotations associated with the data type and the second is the index of the annotation type in the list of types associated with the annotation object.

`index` - the array index at which to set the type (0 or 1)

---

getInfoSourceString

```
public java.lang.String getInfoSourceString(int index)
```

Returns a string for the type of information to be compared. Possible results would be 'Identifiers' if identifiers are to be compared, or 'Annotations' followed by the identifier type at the specified index in the list of identifier types.

Parameters:

`index` - the array index in the annotation types array (0 or 1)

---

getAnnotations

```
public Annotation[] getAnnotations()
```

Returns the array of Annotation objects that are used for the annotation comparison. This array is of length 2 - the first is the annotations projected from, the second is the annotations projected onto.

---

getAnnotationTypeNumbers

```
public int[] getAnnotationTypeNumbers()
```

Returns the array of annotation type numbers that are used for the annotation comparison. This array is of length 2 - the first is the type number of the annotations projected from, the second is the type number of the annotations projected onto.

---

getInfoSource

```
public int getInfoSource(int index)
```

Returns the information type for the comparison at the specified index - 'annotations' or 'identifiers'

Parameters:

`index` - the array index in the annotation types array (0 or 1)

---

setInfoSource

```
public void setInfoSource(int index,
                          int sourceType)
```

Sets the information source (ANNOTATIONS or IDENTIFIERS) for the comparison at the specified index.

Parameters:

`index` - the array index in the annotation types array (0 or 1)

`sourceType` - the type of infomation to be compared (ANNOTATIONS or IDENTIFIERS)

---

clone

```
public java.lang.Object clone()
```

Returns a clone of the AnnotationComparison object.

Overrides:

`clone` in class `java.lang.Object`

---

equals

```
public boolean equals(java.lang.Object obj)
```

Returns true if the two AnnotationComparison objects are equal. Their comparisonTypes must be the same, and their thresholds the same if a 'within' comparison is to be performed.

Overrides:

`equals` in class `java.lang.Object`

Parameters:

`obj` - the AnnotationComparison object to be compared with the receiver.

Returns:

true if the two AnnotationComparison objects are equal.

---

compare

```
public boolean compare(java.lang.String value1,
                       java.lang.String value2)
```

Returns true if the passed string values meet the comparison criteria established by the AnnotationComparison object.

Parameters:

`value1` - the first string value to be compared

`value2` - the second string value to be compared

---

edu.ucsf.Magellan
Class DataProjection

```
java.lang.Object
  └─ edu.ucsf.Magellan.DataProjection
```

```
public class DataProjection
extends java.lang.Object
```

The DataProjection object allows one data type to be projected onto another, subject to user defined parameters. Projection refers to the process of determining 'equivalent' variables between two data types, where 'equivalence' is determined by the user. The DataProjection stores as properties pointers to the data type to project from and onto, as well as a vector of AnnotationComparison objects that determine how variables of a data type are to be compared to determine equivalence. To project, the relevant identifiers or annotations are queried from the database and stored in 2D arrays (making the procedure somewhat memory intensive). The rows of these arrays are compared in a pairwise fashion, and those rows of the data type projected onto which satisfy the criteria are returned as a HashSet of stringified integer ordinal positions.

Field Summary	
private Annotation	annot

[illegible]

private java.util.Vector	**annotComparisons**
static int	**DATA ASSOCIATED**
private DataType	**dtProjectee**
private DataType	**dtProjector**
static int	**PASTED ANNOTATION**
private int	**projectionSource**

Constructor Summary
DataProjection()

Method Summary	
void	**addAnnotationComparison**(AnnotationComparison annotComp) Adds an AnnotationComparison object to the vector of objects to be used to compare variables of data types.
java.lang.Object	**clone**() Returns a clone of an AnnotationComparison objects
private boolean	**compare**(java.lang.String[] tuple1, java.lang.String[] tuple2) Compares two string arrays containing identifiers/annotations, and returns true if they satisfy the comparison criteria in the AnnotationComparison objects associated with this DataProjection object.
boolean	**equals**(java.lang.Object obj) Overrides the equals method to perform a deep comparison between the receiver and the passed object.
Annotation	**getAnnotation**() Gets the Annotation object to be projected from
AnnotationComparison	**getAnnotationComparison**(int position) Returns an AnnotationComparison object from the vector of objects at the specified position
java.util.Vector	**getAnnotationComparisons**() Returns the vector of AnnotationComparison objects associated with the data projection
private java.lang.String[][]	**getAnnotationTable**(int tableNum) Queries the database for the necessary identifiers/annotations for the data projection, and returns them in a 2D array.
DataType	**getDataProjectee**() Gets the data type to be projected onto
DataType	**getDataProjector**() Gets the data type to be projected from
java.util.HashSet	**getProjectedOrdinalPositions**() Performs a variable projection from one data type to another, returning the ordinal positions in the data projectee that satisfied the projection criteria in a HashSet.
int	**getProjectionSource**() Returns the source of the information that is to projected onto a data type.
java.lang.String	**infoString**() Returns a string containing HTML code printing out the projection information associated with the DataProjection object.
void	**removeAnnotationComparison**(int position) Removes an AnnotationComparison object from the vector of objects at the specified position
void	**setAnnotation**(Annotation annot)

[illegible]

	Sets the Annotation object to be projected from
void	**setDataProjectee**(DataType dt) Sets the data type to be projected onto
void	**setDataProjector**(DataType dt) Sets the data type to be projected from
void	**setProjectionSource**(int source) Sets the source of the information that is to projected onto a data type.

Methods inherited from class java.lang.Object
finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

DATA_ASSOCIATED

public static final int **DATA_ASSOCIATED**
See Also:
Constant Field Values

PASTED_ANNOTATION

public static final int **PASTED_ANNOTATION**
See Also:
Constant Field Values

---

dtProjector

private DataType **dtProjector**

---

dtProjectee

private DataType **dtProjectee**

---

annot

private Annotation **annot**

---

annotComparisons

private java.util.Vector **annotComparisons**

---

projectionSource

private int **projectionSource**

Constructor Detail

DataProjection

public **DataProjection**()

[illegible]

Method Detail

setDataProjector

```
public void setDataProjector(DataType dt)
```

Sets the data type to be projected from
Parameters:
`dt` - the DataType object to project from

---

getDataProjector

```
public DataType getDataProjector()
```

Gets the data type to be projected from
Returns:
the DataType object to project from

---

setDataProjectee

```
public void setDataProjectee(DataType dt)
```

Sets the data type to be projected onto
Parameters:
`dt` - the DataType object to project onto

---

getDataProjectee

```
public DataType getDataProjectee()
```

Gets the data type to be projected onto
Returns:
the DataType object to project onto

---

setAnnotation

```
public void setAnnotation(Annotation annot)
```

Sets the Annotation object to be projected from
Parameters:
`annot` - the Annotation object to be projected from

---

getAnnotation

```
public Annotation getAnnotation()
```

Gets the Annotation object to be projected from
Returns:
the Annotation object to be projected from

---

addAnnotationComparison

```
public void addAnnotationComparison(AnnotationComparison annotComp)
```

Adds an AnnotationComparison object to the vector of objects to be used to compare variables of data types.
Parameters:
`annotComp` - the AnnotationComparison object to add.

---

removeAnnotationComparison

```
public void removeAnnotationComparison(int position)
```

Removes an AnnotationComparison object from the vector of objects at the specified position
Parameters:
`position` - the position of the AnnotationComparison object to remove.

---

getAnnotationComparison

`public AnnotationComparison getAnnotationComparison(int position)`

Returns an AnnotationComparison object from the vector of objects at the specified position
Parameters:
`position` - the position of the AnnotationComparison object to return.

---

getAnnotationComparisons

`public java.util.Vector getAnnotationComparisons()`

Returns the vector of AnnotationComparison objects associated with the data projection
Returns:
the vector of AnnotationComparison objects.

---

getProjectionSource

`public int getProjectionSource()`

Returns the source of the information that is to projected onto a data type. Allowable values are the constants DATA_ASSOCIATED or PASTED_ANNOTATION
Returns:
an integer corresponding to the source of the information projected onto data.

---

setProjectionSource

`public void setProjectionSource(int source)`

Sets the source of the information that is to projected onto a data type. Allowable values are the constants DATA_ASSOCIATED or PASTED_ANNOTATION
Parameters:
`source` - an integer corresponding to the source of the information projected onto data.

---

clone

`public java.lang.Object clone()`

Returns a clone of an AnnotationComparison objects
Overrides:
`clone` in class `java.lang.Object`
Returns:
the cloned AnnotationComparison object.

---

equals

`public boolean equals(java.lang.Object obj)`

Overrides the equals method to perform a deep comparison between the receiver and the passed object. The two projector data types and projectee data types must be the same, and the AnnotationComparison objects must be equal.
Overrides:
`equals` in class `java.lang.Object`
Parameters:
`obj` - the DataProjection object to be compared with the receiver.
Returns:
true if the two DataProjection objects are equal.

---

getProjectedOrdinalPositions

[illegible]

```
public java.util.HashSet getProjectedOrdinalPositions()
                                                throws java.sql.SQLException
```
Performs a variable projection from one data type to another, returning the ordinal positions in the data projectee that satisfied the projection criteria in a HashSet.

Returns:
a HashSet of ordinal positions in the data type projected onto that satisfied the projection criteria.

Throws:
`java.sql.SQLException`

---

compare

```
private boolean compare(java.lang.String[] tuple1,
                        java.lang.String[] tuple2)
```
Compares two string arrays containing identifiers/annotations, and returns true if they satisfy the comparison criteria in the AnnotationComparison objects associated with this DataProjection object.

Returns:
true if the contents of the passed arrays meet the comparison criteria.

---

getAnnotationTable

```
private java.lang.String[][] getAnnotationTable(int tableNum)
                                         throws java.sql.SQLException
```
Queries the database for the necessary identifiers/annotations for the data projection, and returns them in a 2D array. Each row of the table is an ordinal position's worth of information. Each column is a different class of information (such as a different annotation).

Returns:
a 2D string array of identifiers/annotations for the data projection.

Throws:
`java.sql.SQLException`

---

infoString

```
public java.lang.String infoString()
```
Returns a string containing HTML code printing out the projection information associated with the DataProjection object. The two data type names are listed, as well as the characteristics of the annotation comparisons to be performed.

Returns:
an HTML string containing the characteristics of the data projection.

---

edu.ucsf.Magellan
Class DataType

```
java.lang.Object
  └─ edu.ucsf.Magellan.DataType
```
Direct Known Subclasses:
DisplayDataType

```
public class DataType
extends java.lang.Object
```

DataType objects represent the data types that are analyzed in Magellan. Data can be thought of as any quantitative or qualitative information that is gathered from samples. Data types represent the different types of information that can be gathered from each sample, such as mRNA expression, clinical characteristics and patient data that could be gathered from a tumor. For each sample analyzed, each data type contains a vector of information of arbitrary length. Each component of a this vector is a separate variable, such as a gene's worth of data in the case of mRNA expression. These variables are often named with a textual label called an 'identifier', such as a genbank ID. Identifiers are used to label variables of a data type, and to link data to curated annotations. DataType objects can be linked to both curated and derived annotations. Curated annotations are linked to the variables of a data type through identifiers. Derived annotations are only applicable to a particular data set, and are linked to the variables of a data type through the ordinal position (row number). Data types can be subselected based on curated or derived annotations, and the DataType

[illegible]

object provides methods to do this. DataType objects contain a vector of Subselection objects that define the criteria by which a the variables of a data type are to be restricted to a subset of the total.

Field Summary	
protected java.util.Vector	**annotationList**
protected int	**dataPointsFromQuery**
protected int	**dataPointsPerSample**
protected int	**dataTypeNumber**
protected int	**experiment**
protected boolean	**isStoredInDatabase**
protected java.lang.String	**name**
protected java.util.Hashtable	**parameters**
protected DataProjection	**projection**
protected int[]	**selectedOrdinalPositions**
protected java.util.Vector	**subSelectionList**
protected java.util.Vector	**uploadIDs**

Constructor Summary
DataType()

Method Summary	
void	**addAnnotation**(Annotation annot) Adds an annotation object to the list of annotations of a data type.
void	**addAnnotation**(Annotation annot, int index) Adds an annotation object to the list of annotations of a data type at the specified position.
void	**addAnnotationFromUpload**(int uploadID) Adds an annotation object from the specified upload to the list of annotations of a data type.
void	**addSubSelection**(SubSelection select) Adds a SubSelection object to the list of SubSelections associated with a data type.
void	**addSubSelection**(SubSelection select, int index) Adds a SubSelection object to the list of SubSelections associated with a data type, at the specified position.
void	**addUploadID**(int uploadID) Adds an upload ID to the vector of upload ID's associated with the data type.
void	**addUploadIDs**(java.util.Vector _uploadIDs) Adds a vector of upload ID's to the existing vector of upload ID's associated with the data type.
java.lang.String	**annotationInfoString**() Returns an HTMLized string listing the properties of the annotations associated with a data

[illegible]

	type.
java.lang.String	**annotationInfoString**(boolean printAll) Returns an HTMLized string listing the properties of the annotations associated with a data type.
java.lang.String	**annotationSelectList**(java.lang.String name) Returns a string containing an HTML SELECT list with the specified name containing the annotations associated with a data type.
java.lang.String	**annotationSelectList**(java.lang.String name, int numSelectionsDisplayed) Returns a string containing an HTML SELECT list with the specified name and number of rows, containing the annotations associated with a data type.
boolean	**annotationsSpecified**() Returns true if there are annotation objects associated with a data type.
boolean	**annotationsToOutput**() Returns true if there are annotations to output to a flat file for analysis.
void	**clearAllAnnotations**() Clears the list of annotations associated with a data type.
void	**clearAllSubSelections**() Clears the list of SubSelection objects associated with a data type.
void	**clearOutputToAnnotationFile**() No annotations associated with a data type will be outputted to a file.
void	**clearOutputToAnnotationFile**(int index) The annotation object at the specified position in the list of annotations will not be outputted to a file.
java.lang.Object	**clone**() Performs a deep copy of a DataType object.
boolean	**equals**(java.lang.Object obj) Overrides the equals method to perform a deep comparison between the receiver and the passed object.
java.util.Vector	**getAnnotationList**() Returns a vector of Annotation objects associated with a data type.
int	**getDataPointsPerSample**() Gets the number of data points per sample that is stored in the database for the data type.
int	**getDataPointsReturnedFromQuery**() Gets the number of data points per sample that is returned from the database after any subselections on the data type are performed
DataProjection	**getDataProjection**() Returns the DataProjection object associated with a data type.
java.lang.String	**getDataSqlStatement**(java.lang.String sampleNumberString) Returns a SQL statement that returns ordinal positions, sample numbers, and data values for a data type.
static DataType	**getDataTypeFromExperiment**(int dtNumber, int experiment) Returns a DataType object from a given experiment with a given data type number.
static DataType	**getDataTypeFromExperiment**(java.lang.String name, int experiment) Returns a DataType object from a given experiment with a given name.
int	**getDataTypeNumber**() Gets the data type number of the data type in its experiment.
static java.util.Vector	**getDataTypesFromExperiment**(int experiment) Returns a vector of DataType objects from a given experiment.
static java.util.Vector	**getDataTypesFromUpload**(int uploadID) Returns a vector of DataType objects from a given upload.
int	**getExperimentNumber**() Returns the experiment number of a data type.

[illegible]

`java.lang.String`	**getName**() Returns the name of a data type.
`java.lang.String`	**getNameWithSubselections**() Returns the name of a data type and a list of its subselections, if any.
`int[]`	**getOrdinalPositions**() Returns an integer array containing the ordinal positions of a data type after subselection and projection.
`java.util.Hashtable`	**getParameters**() Returns a hash containing key:value pairs associated with a data type.
`java.lang.String`	**getParameterValue**(`java.lang.String parameter`) Returns a value for a key that is associated with a data type.
`java.util.HashSet`	**getSubselectedOrdinalPositions**() Takes all the subselection information for a data type, and returns a HashSet of stringified integers containing the ordinal positions of the data that satisfy the selection criteria.
`java.util.Vector`	**getSubselectionList**() Returns a vector of SubSelection objects associated with a data type.
`java.util.Vector`	**getUploadIDs**() Returns a vector of uploadIDs of a data type.
`java.lang.String[]`	**identifiers**() Returns a string array containing the identifiers of a data type.
`boolean`	**isStoredInDatabase**() Returns true if this data type is physically stored in the database.
`void`	**loadDerivedStatistics**() Queries the database for all derived annotations that belongs to this dataType, and adds each one as an annotation Object.
`void`	**loadStoredAnnotations**() Queries the database to see if users have associated annotations with a data type.
`void`	**outputAnnotations**(`java.lang.String[] selectionNumbers`) Allows the user to set the annotations to be outputted.
`void`	**outputToAnnotationFile**(`int index`) The annotation object at the specified position in the list of annotations will be outputted to a file.
`void`	**setDataPointsPerSample**(`int dataPointsPerSample`) Sets the number of data points per sample that was uploaded and stored in the database for a data type.
`void`	**setDataPointsReturnedFromQuery**(`int dataPointsPerSample`) Sets the number of data points per sample that is returned from the database after any subselections on the data type are performed
`void`	**setDataProjection**(DataProjection `project`) Associates a DataProjection object with a data type.
`void`	**setExperimentNumber**(`int experiment`) Sets the experiment number of a data type.
`void`	**setName**(`java.lang.String name`) Sets the name of a data type.
`void`	**setParameterValue**(`java.lang.String parameter, java.lang.String value`) Sets a key:value pair that is associated with a data type.
`boolean`	**statisticUploaded**(`java.lang.String statName`) Returns true if a statistic named 'statName' belonging to the data type has already been uploaded to the database.
`java.lang.String`	**subSelectionInfoString**() Returns an HTML string listing the subselection information for a data type.

Methods inherited from class java.lang.Object

[illegible]

```
finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

Field Detail

name

`protected java.lang.String name`

---

isStoredInDatabase

`protected boolean isStoredInDatabase`

---

experiment

`protected int experiment`

---

dataTypeNumber

`protected int dataTypeNumber`

---

dataPointsPerSample

`protected int dataPointsPerSample`

---

dataPointsFromQuery

`protected int dataPointsFromQuery`

---

selectedOrdinalPositions

`protected int[] selectedOrdinalPositions`

---

uploadIDs

`protected java.util.Vector uploadIDs`

---

annotationList

`protected java.util.Vector annotationList`

---

subSelectionList

`protected java.util.Vector subSelectionList`

---

projection

`protected DataProjection projection`

[illegible]

parameters

```
protected java.util.Hashtable parameters
```

Constructor Detail

DataType

```
public DataType()
```

Method Detail

getName

```
public java.lang.String getName()
```

Returns the name of a data type.
Returns:
the data type name

---

setName

```
public void setName(java.lang.String name)
```

Sets the name of a data type.
Parameters:
name - the name of the data type

---

getNameWithSubselections

```
public java.lang.String getNameWithSubselections()
```

Returns the name of a data type and a list of its subselections, if any. The results are in HTML format.
Returns:
data type name and subselections

---

isStoredInDatabase

```
public boolean isStoredInDatabase()
```

Returns true if this data type is physically stored in the database. Returns false if the data type is derived by subselection or projection.
Returns:
true if data type is stored in the database.

---

getExperimentNumber

```
public int getExperimentNumber()
```

Returns the experiment number of a data type.
Returns:
experiment number of the data type.

---

setExperimentNumber

```
public void setExperimentNumber(int experiment)
```

Sets the experiment number of a data type.
Parameters:
`experiment` - the experiment number of the data type

[illegible]

getUploadIDs

public java.util.Vector **getUploadIDs**()

Returns a vector of uploadIDs of a data type. A data type may have more than one upload ID number if different samples are loaded to the database at different times.

Returns:

a vector of uploadIDs of a data type.

---

addUploadID

public void **addUploadID**(int uploadID)

Adds an upload ID to the vector of upload ID's associated with the data type.

Parameters:

`uploadID` - the upload ID to add to the vector of upload ID's

---

addUploadIDs

public void **addUploadIDs**(java.util.Vector _uploadIDs)

Adds a vector of upload ID's to the existing vector of upload ID's associated with the data type.

Parameters:

`_uploadIDs` - the vector of upload ID's to add to the existing vector of upload ID's

---

getDataPointsPerSample

public int **getDataPointsPerSample**()

Gets the number of data points per sample that is stored in the database for the data type.

Returns:

the number of data points per sample for a data type.

---

setDataPointsPerSample

public void **setDataPointsPerSample**(int dataPointsPerSample)

Sets the number of data points per sample that was uploaded and stored in the database for a data type. This method is generally passed the results of a SQL query.

Parameters:

`dataPointsPerSample` - the number of data points per sample for the data type in question.

---

getDataPointsReturnedFromQuery

public int **getDataPointsReturnedFromQuery**()

Gets the number of data points per sample that is returned from the database after any subselections on the data type are performed

Returns:

the number of data points per sample after subselections/projections

---

setDataPointsReturnedFromQuery

public void **setDataPointsReturnedFromQuery**(int dataPointsPerSample)

Sets the number of data points per sample that is returned from the database after any subselections on the data type are performed

Parameters:

`dataPointsPerSample` - the number of data points per sample returned from the database after subselections.

---

getDataTypeNumber

[illegible]

```
public int getDataTypeNumber()
```
Gets the data type number of the data type in its experiment.
Returns:
the data type number

---

getAnnotationList

```
public java.util.Vector getAnnotationList()
```
Returns a vector of Annotation objects associated with a data type.
Returns:
a vector of Annotation objects associated with a data type.

---

getSubselectionList

```
public java.util.Vector getSubselectionList()
```
Returns a vector of SubSelection objects associated with a data type.
Returns:
a vector of SubSelection objects associated with a data type.

---

getParameters

```
public java.util.Hashtable getParameters()
```
Returns a hash containing key:value pairs associated with a data type. This hash is generally used to store analysis parameters for a data type.
Returns:
a HashTable of key:value pairs used in an analysis.

---

setParameterValue

```
public void setParameterValue(java.lang.String parameter,
                              java.lang.String value)
```
Sets a key:value pair that is associated with a data type. This is generally used to store an analysis parameter for a data type.
Parameters:
`parameter` - the key
`value` - the value

---

getParameterValue

```
public java.lang.String getParameterValue(java.lang.String parameter)
```
Returns a value for a key that is associated with a data type. This is generally used to store analysis parameters for a data type.
Returns:
the key whose value is to be looked up

---

getDataProjection

```
public DataProjection getDataProjection()
```
Returns the DataProjection object associated with a data type.
Returns:
a DataProjection object associated with a data type.

---

setDataProjection

```
public void setDataProjection(DataProjection project)
```
Associates a DataProjection object with a data type.

[illegible]

Parameters:
`project` - the DataProjection object to associate with a data type.

---

equals

```
public boolean equals(java.lang.Object obj)
```

Overrides the equals method to perform a deep comparison between the receiver and the passed object. Tests by comparing such properties as name, experiment, subselections, and projections.
Overrides:
`equals` in class `java.lang.Object`
Parameters:
`obj` - the DataType object to compare.
Returns:
true if the two DataType objects are equal.

---

clone

```
public java.lang.Object clone()
```

Performs a deep copy of a DataType object.
Overrides:
`clone` in class `java.lang.Object`
Returns:
a DataType object with deep copies of annotation and subselection lists

---

getDataTypesFromExperiment

```
public static java.util.Vector getDataTypesFromExperiment(int experiment)
                                                   throws java.sql.SQLException
```

Returns a vector of DataType objects from a given experiment. The database is queried to fill the properties of each object.
Parameters:
`experiment` - the number of the experiment in the database
Returns:
a vector of DataType objects
Throws:
`java.sql.SQLException`

---

getDataTypeFromExperiment

```
public static DataType getDataTypeFromExperiment(java.lang.String name,
                                                 int experiment)
                                          throws java.sql.SQLException
```

Returns a DataType object from a given experiment with a given name. The database is queried to fill the properties the object.
Parameters:
`experiment` - the number of the experiment in the database
`name` - the name of the data type to be returned
Returns:
a DataType object with the specified name from the specified experiment
Throws:
`java.sql.SQLException`

---

getDataTypeFromExperiment

```
public static DataType getDataTypeFromExperiment(int dtNumber,
                                                 int experiment)
                                          throws java.sql.SQLException
```

Returns a DataType object from a given experiment with a given data type number. The database is queried to fill the properties the object.
Parameters:

[illegible]

`dtNumber` - the data type number of the data type to be returned
`experiment` - the number of the experiment in the database
Returns:
a DataType object with the specified name from the specified experiment
Throws:
`java.sql.SQLException`

---

getDataTypesFromUpload

```
public static java.util.Vector getDataTypesFromUpload(int uploadID)
                                               throws java.sql.SQLException
```

Returns a vector of DataType objects from a given upload. The database is queried to fill the properties of each object.
Parameters:
`uploadID` - the number of the upload in the database
Returns:
a vector of DataType objects from the specified upload.
Throws:
`java.sql.SQLException`

---

loadStoredAnnotations

```
public void loadStoredAnnotations()
                           throws java.sql.SQLException
```

Queries the database to see if users have associated annotations with a data type. If so, Annotation objects are instantiated and added to the list of annotations of the data type.
Throws:
`java.sql.SQLException`

---

annotationsToOutput

```
public boolean annotationsToOutput()
```

Returns true if there are annotations to output to a flat file for analysis.
Returns:
true if there are annotations to be outputted.

---

outputAnnotations

```
public void outputAnnotations(java.lang.String[] selectionNumbers)
```

Allows the user to set the annotations to be outputted. The passed parameter selectionNumbers is usually an array returned by the request.getParameterValues() method.
Parameters:
`selectionNumbers` - an array of Strings corresponding to the annotations to be outputted.

---

addAnnotation

```
public void addAnnotation(Annotation annot)
```

Adds an annotation object to the list of annotations of a data type.
Parameters:
`annot` - the annotation object to be added.

---

addAnnotation

```
public void addAnnotation(Annotation annot,
                          int index)
```

Adds an annotation object to the list of annotations of a data type at the specified position.
Parameters:
`annot` - the annotation object to be added.

[illegible]

`index` - the position in the list of annotations where the annotation object should be added.

---

addAnnotationFromUpload

```
public void addAnnotationFromUpload(int uploadID)
                              throws java.sql.SQLException
```

Adds an annotation object from the specified upload to the list of annotations of a data type.
Parameters:
`uploadID` - the upload ID of the annotation object to be added.
Throws:
`java.sql.SQLException`

---

loadDerivedStatistics

```
public void loadDerivedStatistics()
                           throws java.sql.SQLException
```

Queries the database for all derived annotations that belongs to this dataType, and adds each one as an annotation Object.
Throws:
`java.sql.SQLException`

---

statisticUploaded

```
public boolean statisticUploaded(java.lang.String statName)
                          throws java.sql.SQLException
```

Returns true if a statistic named 'statName' belonging to the data type has already been uploaded to the database.
Parameters:
`statName` - the name of the statistic whose existence will be checked in the database
Returns:
true if a statistic named 'statName' belonging to the data type is in the database.
Throws:
`java.sql.SQLException`

---

clearAllAnnotations

```
public void clearAllAnnotations()
```

Clears the list of annotations associated with a data type.

---

clearOutputToAnnotationFile

```
public void clearOutputToAnnotationFile()
```

No annotations associated with a data type will be outputted to a file.

---

clearOutputToAnnotationFile

```
public void clearOutputToAnnotationFile(int index)
```

The annotation object at the specified position in the list of annotations will not be outputted to a file.
Parameters:
`index` - the index in the vector of annotations which will not be outputted.

---

outputToAnnotationFile

```
public void outputToAnnotationFile(int index)
```

The annotation object at the specified position in the list of annotations will be outputted to a file.
Parameters:
`index` - the index in the vector of annotations which will be outputted.

[illegible]

annotationsSpecified

`public boolean annotationsSpecified()`

Returns true if there are annotation objects associated with a data type.

Returns:

true if there are annotation objects associated with a data type.

---

annotationInfoString

`public java.lang.String annotationInfoString()`

Returns an HTMLized string listing the properties of the annotations associated with a data type.

Returns:

HTML code listing annotation properties

---

annotationInfoString

`public java.lang.String annotationInfoString(boolean printAll)`

Returns an HTMLized string listing the properties of the annotations associated with a data type.

Parameters:

`printAll` - if false, print out information only for annotations that are to be outputted for the current analysis

Returns:

HTML code listing annotation properties

---

annotationSelectList

`public java.lang.String annotationSelectList(java.lang.String name)`

Returns a string containing an HTML SELECT list with the specified name containing the annotations associated with a data type. The SELECT list displays the description of each annotation, and the value for each annotation passed to the next web page is its position in the list of annotations associated with a data type.

Parameters:

`name` - the NAME property of the SELECT list

Returns:

a string containing an HTML SELECT list with the specified name containing the annotations associated with a data type.

---

annotationSelectList

```
public java.lang.String annotationSelectList(java.lang.String name,
                                              int numSelectionsDisplayed)
```

Returns a string containing an HTML SELECT list with the specified name and number of rows, containing the annotations associated with a data type. The SELECT list displays the description of each annotation, and the value for each annotation passed to the next web page is its position in the list of annotations associated with a data type.

Parameters:

`name` - the NAME property of the SELECT list

`numSelectionsDisplayed` - the SIZE property of the MULTIPLE SELECT drop down list.

Returns:

a string containing an HTML SELECT list with the specified name and number of rows, containing the annotations associated with a data type.

---

addSubSelection

`public void addSubSelection(SubSelection select)`

Adds a SubSelection object to the list of SubSelections associated with a data type.

Parameters:

`select` - the SubSelection object to associate with the data type.

[illegible]

addSubSelection

```
public void addSubSelection(SubSelection select,
                            int index)
```

Adds a SubSelection object to the list of SubSelections associated with a data type, at the specified position.
Parameters:
`select` - the SubSelection object to associate with the data type.
`index` - the position in the list of subselections where the SubSelection object should be added.

---

clearAllSubSelections

```
public void clearAllSubSelections()
```

Clears the list of SubSelection objects associated with a data type.

---

subSelectionInfoString

```
public java.lang.String subSelectionInfoString()
```

Returns an HTML string listing the subselection information for a data type. Prints out information for all subselections associated with a data type, whether they are to be applied or not.
Returns:
an HTML string listing the subselection information for a data type.

---

identifiers

```
public java.lang.String[] identifiers()
                               throws java.sql.SQLException
```

Returns a string array containing the identifiers of a data type.
Returns:
a string array containing the identifiers of a data type.
Throws:
`java.sql.SQLException`

---

getOrdinalPositions

```
public int[] getOrdinalPositions()
                          throws java.sql.SQLException
```

Returns an integer array containing the ordinal positions of a data type after subselection and projection. Subselection is performed prior to projection.
Returns:
an integer array containing the ordinal positions of a data type after subselection and projection.
Throws:
`java.sql.SQLException`

---

getSubselectedOrdinalPositions

```
public java.util.HashSet getSubselectedOrdinalPositions()
                                                 throws java.sql.SQLException
```

Takes all the subselection information for a data type, and returns a HashSet of stringified integers containing the ordinal positions of the data that satisfy the selection criteria.
Returns:
A HashSet of ordinal positions that satify the subselection criteria for a data type.
Throws:
`java.sql.SQLException`

---

getDataSqlStatement

```
public java.lang.String getDataSqlStatement(java.lang.String sampleNumberString)
```

[illegible]

Returns a SQL statement that returns ordinal positions, sample numbers, and data values for a data type. If a developer is overriding methods to produce flat data files in novel file formats, this method will return a SQL statement to retrieve the data first.
Returns:
a SQL statement that returns ordinal positions, sample numbers, and data values for a data type.

edu.ucsf.Magellan
Class DisplayDataType

```
java.lang.Object
  └─ edu.ucsf.Magellan.DataType
       └─ edu.ucsf.Magellan.DisplayDataType
```

```
public class DisplayDataType
extends DataType
```

DisplayDataType is a subclass of the DataType class that contains properties and methods specifically for the display of data in the web browser.

Field Summary	
`protected static java.util.Vector`	**dataTypeList**
`protected boolean`	**displayAllRows**
`protected java.lang.String`	**identifierType**
`protected int`	**numRowsToDisplay**
`protected java.util.Vector`	**sampleNames**

Fields inherited from class edu.ucsf.Magellan.DataType
annotationList, dataPointsFromQuery, dataPointsPerSample, dataTypeNumber, experiment, isStoredInDatabase, name, parameters, projection, selectedOrdinalPositions, subSelectionList, uploadIDs

Constructor Summary
DisplayDataType()
DisplayDataType(boolean addToList)

Method Summary	
`static void`	**addDataTypesFromExperimentToList**(int experiment) Returns a DisplayDataType object with the specified name and experiment number.
`static void`	**clearDataTypeList**() Removes all elements from the vector of DisplayDataType objects that are to be displayed in the output.
`boolean`	**displayAllRows**() If true, then all the rows of the data type will be displayed in the output.
`java.util.Vector`	**getAllAnnotationTypes**() Returns a vector containing strings of the annotation types associated with the

[illegible]

	DisplayDataType object.
java.lang.String[][]	**getDataTable**() Returns a 2D string array containing the data from the experiment and data type stored in the DisplayDataType object's properties.
static DisplayDataType	**getDataTypeFromExperimentInList**(java.lang.String name, int experiment) Returns a DisplayDataType object with the specified name and experiment number.
static DisplayDataType	**getDataTypeFromUploadInList**(java.lang.String name, int uploadID) Returns a DisplayDataType object with the specified name and upload ID number.
static java.util.Vector	**getDataTypeList**() Returns the vector of DisplayDataType objects that are to be displayed in the output.
static DisplayDataType	**getDisplayDataTypeFromExperiment**(int dtNumber, int experiment) Returns a DataType object from a given experiment with a given data type number.
java.lang.String[]	**getIdentifiers**() Returns a string array containing the identifiers from the experiment and data type stored in the DisplayDataType object's properties.
java.lang.String	**getIdentifierType**() Returns the user defined type of the identifiers associated with the data type (genbank ID for example).
int	**getNumRowsToDisplay**() Returns the number of rows of the data type to display in the output.
java.util.Vector	**getSampleNames**() Returns the vector of sample names associated with this data type.
void	**setDisplayAllRows**(boolean displayAll) Sets whether all the rows of the data type will be displayed in the output.
void	**setNumRowsToDisplay**(int numRows) Sets the number of rows of the data type to display in the output.

Methods inherited from class edu.ucsf.Magellan.DataType
addAnnotation, addAnnotation, addAnnotationFromUpload, addSubSelection, addSubSelection, addUploadID, addUploadIDs, annotationInfoString, annotationInfoString, annotationSelectList, annotationSelectList, annotationsSpecified, annotationsToOutput, clearAllAnnotations, clearAllSubSelections, clearOutputToAnnotationFile, clearOutputToAnnotationFile, clone, equals, getAnnotationList, getDataPointsPerSample, getDataPointsReturnedFromQuery, getDataProjection, getDataSqlStatement, getDataTypeFromExperiment, getDataTypeFromExperiment, getDataTypeNumber, getDataTypesFromExperiment, getDataTypesFromUpload, getExperimentNumber, getName, getNameWithSubselections, getOrdinalPositions, getParameters, getParameterValue, getSubselectedOrdinalPositions, getSubselectionList, getUploadIDs, identifiers, isStoredInDatabase, loadDerivedStatistics, loadStoredAnnotations, outputAnnotations, outputToAnnotationFile, setDataPointsPerSample, setDataPointsReturnedFromQuery, setDataProjection, setExperimentNumber, setName, setParameterValue, statisticUploaded, subSelectionInfoString

Methods inherited from class java.lang.Object
finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

displayAllRows

```
protected boolean displayAllRows
```

numRowsToDisplay

```
protected int numRowsToDisplay
```

[illegible]

identifierType

```
protected java.lang.String identifierType
```

---

sampleNames

```
protected java.util.Vector sampleNames
```

---

dataTypeList

```
protected static java.util.Vector dataTypeList
```

Constructor Detail

DisplayDataType

```
public DisplayDataType()
```

DisplayDataType

```
public DisplayDataType(boolean addToList)
```

Method Detail

getIdentifierType

```
public java.lang.String getIdentifierType()
```

Returns the user defined type of the identifiers associated with the data type (genbank ID for example).
Returns:
the type of identifier

---

getNumRowsToDisplay

```
public int getNumRowsToDisplay()
```

Returns the number of rows of the data type to display in the output.
Returns:
the number of rows of the data type to display.

---

setNumRowsToDisplay

```
public void setNumRowsToDisplay(int numRows)
```

Sets the number of rows of the data type to display in the output. If the number 10 is passed to this method, then the first and last 10 rows of data will be displayed
Parameters:
`numRows` - the number of rows of the data type to display.

---

displayAllRows

```
public boolean displayAllRows()
```

If true, then all the rows of the data type will be displayed in the output.
Returns:
true if all rows of the data type are to be displayed.

---

setDisplayAllRows

[illegible]

`public void **setDisplayAllRows**(boolean displayAll)`

Sets whether all the rows of the data type will be displayed in the output.
Parameters:
`displayAll` - true if all rows of the data type are to be displayed.

---

getDataTypeList

`public static java.util.Vector **getDataTypeList**()`

Returns the vector of DisplayDataType objects that are to be displayed in the output. This is a static variable that contains a list of all instantiated objects to be displayed.
Returns:
a vector of DisplayDataType objects that are to be displayed in the output.

---

clearDataTypeList

`public static void **clearDataTypeList**()`

Removes all elements from the vector of DisplayDataType objects that are to be displayed in the output. This is a static variable that contains a list of all instantiated objects to be displayed.

---

getSampleNames

`public java.util.Vector **getSampleNames**()`

Returns the vector of sample names associated with this data type.
Returns:
the vector of sample names associated with this data type.

---

getDataTypeFromUploadInList

```
public static DisplayDataType getDataTypeFromUploadInList(java.lang.String name,
                                                           int uploadID)
```

Returns a DisplayDataType object with the specified name and upload ID number. If no data type in the database meets the passed criteria, null is returned.
Parameters:
`name` - the name of the data type to be searched for in the database
`uploadID` - the uploadID of the data type to be searched for in the database
Returns:
the DisplayDataType object with the specified name and upload ID number.

---

getDataTypeFromExperimentInList

```
public static DisplayDataType getDataTypeFromExperimentInList(java.lang.String name,
                                                               int experiment)
```

Returns a DisplayDataType object with the specified name and experiment number. If no data type in the database meets the passed criteria, null is returned.
Parameters:
`name` - the name of the data type to be searched for in the database
`experiment` - the experiment of the data type to be searched for in the database
Returns:
the DisplayDataType object with the specified name and experiment number.

---

addDataTypesFromExperimentToList

```
public static void addDataTypesFromExperimentToList(int experiment)
                                      throws java.sql.SQLException
```

Returns a DisplayDataType object with the specified name and experiment number. If no data type in the database meets the passed criteria, null is returned.
Parameters:

[illegible]

`experiment` - the experiment number of the data types to be added
Throws:
`java.sql.SQLException`

---

getDisplayDataTypeFromExperiment

```
public static DisplayDataType getDisplayDataTypeFromExperiment(int dtNumber,
                                                               int experiment)
                                                        throws java.sql.SQLException
```

Returns a DataType object from a given experiment with a given data type number. The database is queried to fill the properties the object.
Parameters:
`experiment` - the number of the experiment in the database
`dtNumber` - the data type number of the data type to be returned
Returns:
a DataType object with the specified name from the specified experiment
Throws:
`java.sql.SQLException`

---

getDataTable

```
public java.lang.String[][] getDataTable()
                                  throws java.sql.SQLException
```

Returns a 2D string array containing the data from the experiment and data type stored in the DisplayDataType object's properties. If the data types has more than twice the number of rows specified by the numRowsToDisplay property, then the first and last numRowsToDisplay rows of data will be returned.
Returns:
a 2D string array of data.
Throws:
`java.sql.SQLException`

---

getIdentifiers

```
public java.lang.String[] getIdentifiers()
                                  throws java.sql.SQLException
```

Returns a string array containing the identifiers from the experiment and data type stored in the DisplayDataType object's properties. If the data types has more than twice the number of rows specified by the numRowsToDisplay property, then the first and last numRowsToDisplay rows of identifiers will be returned.
Returns:
a string array of identifiers.
Throws:
`java.sql.SQLException`

---

getAllAnnotationTypes

```
public java.util.Vector getAllAnnotationTypes()
```

Returns a vector containing strings of the annotation types associated with the DisplayDataType object.
Returns:
a vector containing strings of annotation types

---

edu.ucsf.Magellan
Class execProcess

```
java.lang.Object
  └─ edu.ucsf.Magellan.execProcess
```

```
public class execProcess
```

[illegible]

```
extends java.lang.Object
```

The execProcess class allows the execution of one or more command line statements as separate processes. execProcess extends the Thread class, and uses the runtime.exec() method to fork off the command line processes. The StreamGobbler class is used to capture the stdout and stderr streams associated with the process (if the streams are not captured, the processes will usually hang). The execProcess object is generally used to execute command line arguments such as those necessary to execute various analytical applications.

Constructor Summary
`execProcess()`

Method Summary	
`static int`	`exec(java.lang.String cmd)` Executes a single command line argument.
`static int`	`exec(java.lang.String cmd, boolean outputFlag)` Executes a single command line argument.
`static int`	`exec(java.lang.String cmd, boolean outputFlag, java.io.FileWriter outFile)` Executes a single command line argument.
`static int`	`exec(java.lang.String cmd, java.io.FileWriter outFile)` Executes a single command line argument.
`static void`	`exec(java.util.Vector execStatements)` Executes a series of command line arguments contained in the passed vector.
`static void`	`exec(java.util.Vector execStatements, boolean out)` Executes a series of command line arguments contained in the passed vector.
`static void`	`exec(java.util.Vector execStatements, java.io.FileWriter outFile)` Executes a series of command line arguments contained in the passed vector.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

execProcess

```
public execProcess()
```

Method Detail

exec

```
public static void exec(java.util.Vector execStatements)
```

Executes a series of command line arguments contained in the passed vector.
Parameters:
`execStatements` - a vector containing strings of command line arguments to be executed.

---

exec

```
public static void exec(java.util.Vector execStatements,
                        java.io.FileWriter outFile)
```

Executes a series of command line arguments contained in the passed vector. The passed FileWriter object is used to print any output generated by the forked process to a text file (in development).
Parameters:
`execStatements` - a vector containing strings of command line arguments to be executed.

[illegible]

`outFile` - the FileWriter object to which any process output will be printed.

---

exec

```
public static void exec(java.util.Vector execStatements,
                        boolean out)
```

Executes a series of command line arguments contained in the passed vector. If the passed boolean flag is true, then any output generated by the forked process with be printed to stdout.
Parameters:
`execStatements` - a vector containing strings of command line arguments to be executed.
`out` - if true, any process output will be printed to stdout.

---

exec

```
public static int exec(java.lang.String cmd)
```

Executes a single command line argument.
Parameters:
cmd - the command line argument to be executed.
Returns:
the return value of the process (typically 0 if no errors occurred, otherwise 1).

---

exec

```
public static int exec(java.lang.String cmd,
                       java.io.FileWriter outFile)
```

Executes a single command line argument. If the passed boolean flag is true, then any output generated by the forked process with be printed to stdout.
Parameters:
cmd - the command line argument to be executed.
`outFile` - the FileWriter object to which any process output will be printed.
Returns:
the return value of the process (typically 0 if no errors occurred, otherwise 1).

---

exec

```
public static int exec(java.lang.String cmd,
                       boolean outputFlag)
```

Executes a single command line argument. If the passed boolean flag is true, then any output generated by the forked process with be printed to stdout.
Parameters:
cmd - the command line argument to be executed.
`outputFlag` - if true, any process output will be printed to stdout.
Returns:
the return value of the process (typically 0 if no errors occurred, otherwise 1).

---

exec

```
public static int exec(java.lang.String cmd,
                       boolean outputFlag,
                       java.io.FileWriter outFile)
```

Executes a single command line argument. If the passed boolean flag is true, then any output generated by the forked process with be printed to stdout. If the passed FileWriter object is not null, then any output generated by the forked process will be printed to a file (in development).
Parameters:
cmd - the command line argument to be executed.
`outputFlag` - if true, any process output will be printed to stdout.
`outFile` - the FileWriter object to which any process output will be printed.
Returns:
the return value of the process (typically 0 if no errors occurred, otherwise 1).

[illegible]

edu.ucsf.Magellan
Class execThread

```
java.lang.Object
  └─java.lang.Thread
      └─edu.ucsf.Magellan.execThread
```

All Implemented Interfaces:
java.lang.Runnable

```
 class execThread
extends java.lang.Thread
```

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread
java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler

Field Summary	
(package private) java.util.Vector	**execStatements**
(package private) java.io.FileWriter	**outFile**
(package private) boolean	**outputFlag**

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary
execThread(java.util.Vector execStatements, boolean outputFlag)
execThread(java.util.Vector execStatements, java.io.FileWriter outFile)

Method Summary	
void	**run**()

Methods inherited from class java.lang.Thread
activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getAllStackTraces, getContextClassLoader, getDefaultUncaughtExceptionHandler, getId, getName, getPriority, getStackTrace, getState, getThreadGroup, getUncaughtExceptionHandler, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setDefaultUncaughtExceptionHandler, setName, setPriority, setUncaughtExceptionHandler, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

[illegible]

execStatements

```
java.util.Vector execStatements
```

---

outFile

```
java.io.FileWriter outFile
```

---

outputFlag

```
boolean outputFlag
```

Constructor Detail

execThread

```
public execThread(java.util.Vector execStatements,
                  java.io.FileWriter outFile)
```

---

execThread

```
public execThread(java.util.Vector execStatements,
                  boolean outputFlag)
```

Method Detail

run

```
public void run()
```

Specified by:
run in interface `java.lang.Runnable`
Overrides:
run in class `java.lang.Thread`

---

edu.ucsf.Magellan
Class Sample

```
java.lang.Object
  └─ edu.ucsf.Magellan.Sample
```

```
public class Sample
extends java.lang.Object
```

The Sample class captures the functionality of samples (biological or otherwise) from which data is gathered.

Field Summary	
protected int	experiment
protected boolean	outputToDataFile
protected java.lang.String	sampleName

[illegible]

protected int	**sampleNumber**
protected java.util.Vector	**uploadIDs**

Constructor Summary
Sample()

Method Summary	
void	**addUploadID**(int uploadID) Adds the passed uploadID number to the vector of upload ID's associated with a sample.
boolean	**equals**(java.lang.Object obj) Overrides the equals method to perform a deep comparison between the receiver and the passed object.
int	**getExperimentNumber**() Returns the experiment number for a given sample
java.lang.String	**getName**() Returns the name of a given sample
static java.util.Vector	**getSampleNames**(int experiment) Returns a vector containing the sample names for a given experiment.
int	**getSampleNumber**() Returns the sample number for a given sample.
static java.util.Vector	**getSamplesFromExperiment**(int experiment) Returns a vector populated with Sample objects from the passed experiment number
java.util.Vector	**getSamplesFromUpload**(int upload) Returns a vector populated with Sample objects from the passed upload number

Methods inherited from class java.lang.Object
clone, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

experiment

protected int **experiment**

---

sampleNumber

protected int **sampleNumber**

---

uploadIDs

protected java.util.Vector **uploadIDs**

---

sampleName

protected java.lang.String **sampleName**

---

outputToDataFile

[illegible]

`protected boolean` **`outputToDataFile`**

Constructor Detail

Sample

`public` **`Sample`**`()`

Method Detail

getName

`public java.lang.String` **`getName`**`()`

Returns the name of a given sample
Returns:
the sample name

---

getExperimentNumber

`public int` **`getExperimentNumber`**`()`

Returns the experiment number for a given sample
Returns:
the sample experiment number

---

getSampleNumber

`public int` **`getSampleNumber`**`()`

Returns the sample number for a given sample. The sample number is an integer ranging from 1 to the total number of samples in a given experiment.
Returns:
the sample number

---

equals

`public boolean` **`equals`**`(java.lang.Object obj)`

Overrides the equals method to perform a deep comparison between the receiver and the passed object. The equals method for samples returns true if the sample number and experiment number of two samples are the same.
Overrides:
`equals` in class `java.lang.Object`
Parameters:
`obj` - the Sample object to compare.
Returns:
true if the two Sample objects are equal.

---

getSamplesFromExperiment

```
public static java.util.Vector getSamplesFromExperiment(int experiment)
                                                  throws java.sql.SQLException
```

Returns a vector populated with Sample objects from the passed experiment number
Parameters:
`experiment` - the experiment number
Returns:
a vector of Sample objects
Throws:
`java.sql.SQLException`

---

getSamplesFromUpload

[illegible]

```
public java.util.Vector getSamplesFromUpload(int upload)
                                      throws java.sql.SQLException
```

Returns a vector populated with Sample objects from the passed upload number
Parameters:
`upload` - the upload ID number
Returns:
a vector of Sample objects
Throws:
`java.sql.SQLException`

---

getSampleNames

```
public static java.util.Vector getSampleNames(int experiment)
                                       throws java.sql.SQLException
```

Returns a vector containing the sample names for a given experiment.
Parameters:
`experiment` - the experiment number
Throws:
`java.sql.SQLException`

---

addUploadID

```
public void addUploadID(int uploadID)
```

Adds the passed uploadID number to the vector of upload ID's associated with a sample. This vector may contain multiple upload ID's if different data types are loaded for a common sample in different data uploads.
Parameters:
`uploadID` - the upload ID number

---

edu.ucsf.Magellan
Class StreamGobbler

```
java.lang.Object
  └─java.lang.Thread
      └─edu.ucsf.Magellan.StreamGobbler
```

All Implemented Interfaces:
java.lang.Runnable

```
 class StreamGobbler
extends java.lang.Thread
```

The StreamGobbler class is used to capture streams associated with command line processes that have been forked off by execProcess objects.

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread
`java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler`

Field Summary	
`(package private) java.io.InputStream`	`is`
`(package private) java.io.FileWriter`	`outFile`
`(package private)`	`outputFlag`

[illegible]

boolean	
(package private) java.lang.String	type

Fields inherited from class java.lang.Thread
MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary
StreamGobbler(java.io.InputStream is, java.lang.String type, boolean outputFlag, java.io.FileWriter outFile)

Method Summary	
void	run()

Methods inherited from class java.lang.Thread
activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getAllStackTraces, getContextClassLoader, getDefaultUncaughtExceptionHandler, getId, getName, getPriority, getStackTrace, getState, getThreadGroup, getUncaughtExceptionHandler, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setDefaultUncaughtExceptionHandler, setName, setPriority, setUncaughtExceptionHandler, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

is

```
java.io.InputStream is
```

type

```
java.lang.String type
```

outFile

```
java.io.FileWriter outFile
```

outputFlag

```
boolean outputFlag
```

Constructor Detail

StreamGobbler

```
StreamGobbler(java.io.InputStream is,
              java.lang.String type,
              boolean outputFlag,
```

[illegible]

```
java.io.FileWriter outFile)
```

Method Detail

run

```
public void run()
```

Specified by:
run in interface `java.lang.Runnable`
Overrides:
run in class `java.lang.Thread`

---

edu.ucsf.Magellan Class SubSelection

```
java.lang.Object
  └─ edu.ucsf.Magellan.SubSelection
```

```
public class SubSelection
extends java.lang.Object
```

SubSelection objects provide the functionality to select variables from a data type based on user defined criteria. Variable sub-selection allows a user to focus on a subset of the variables of a data type. Subselection can be performed using identifers or annotations. If annotations are used, selection can be performed based on qualitative or quantitative criteria.

Field Summary	
protected Annotation	annot
protected java.lang.String	annotationType
protected int	annotTypeNumber
protected java.lang.String	comparisonOperator
protected java.lang.String	comparisonType
protected boolean	selectByAnnotation
protected boolean	selectByIdentifier
protected java.lang.String[]	stringSelections
protected double	threshold

Constructor Summary
SubSelection()

Method Summary	
java.lang.Object	clone() Implements the clone method to allow a deep copy of a subselection item.

[illegible]

boolean	**equals**(java.lang.Object obj) Overrides the equals method to perform a deep comparison between the receiver and the passed object.
Annotation	**getAnnotation**() Returns the Annotation object that is to be used in an annotation based subselection.
java.lang.String	**getAnnotationType**() Returns the type of the annotation that is to be used for subselection.
int	**getAnnotTypeNumber**() Returns the number of the annotation type that is to be used for subselection.
java.lang.String	**getComparisonOperator**() Returns the comparison operator (>, <, =, etc) that is to be used for a quantitative subselection.
java.lang.String	**getComparisonType**() Returns the comparison type ('qualitative' or 'quantitative') that is to be used in the subselection.
java.lang.String	**getDescription**() Returns a string description of the subselection criteria.
java.lang.String[]	**getStringSelections**() Returns an array of the strings that are to be selected for.
double	**getThreshold**() Returns the threshold that is to be used for a quantitative subselection.
java.lang.String	**infoString**() Returns a string containing the subselection criteria specified by the object properties.
boolean	**selectByAnnotation**() Returns true if a subselection is to be made based on annotations.
boolean	**selectByIdentifier**() Returns true if a subselection is to be made based on identifier names.
private void	**setAllSelectToFalse**() Sets all selection flags to false.
void	**setAnnotation**(Annotation annotation) Sets the Annotation object that is to be used in an annotation based subselection.
void	**setAnnotationType**(java.lang.String annotationType) Sets the type of the annotation that is to be used for subselection.
void	**setAnnotTypeNumber**(int annotTypeNumber) Sets the number of the annotation type that is to be used for subselection.
void	**setComparisonOperator**(java.lang.String comparisonOperator) Sets the comparison operator (>, <, =, etc) that is to be used for a quantitative subselection.
void	**setComparisonType**(java.lang.String comparisonType) Sets the comparison type ('qualitative' or 'quantitative') that is to be used in the subselection.
void	**setSelectByAnnotation**() Specifies that a subselection shall be made based on annotations.
void	**setSelectByIdentifier**() Specifies that a subselection shall be made based on identifier names.
void	**setStringSelections**(java.lang.String[] stringSelections) Sets the array of the strings that are to be selected for.
void	**setThreshold**(double threshold) Sets the threshold that is to be used for a quantitative subselection.

Methods inherited from class java.lang.Object
finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

selectByIdentifier

[illegible]

```
protected boolean selectByIdentifier
```

---

selectByAnnotation

```
protected boolean selectByAnnotation
```

---

stringSelections

```
protected java.lang.String[] stringSelections
```

---

annotTypeNumber

```
protected int annotTypeNumber
```

---

annotationType

```
protected java.lang.String annotationType
```

---

comparisonType

```
protected java.lang.String comparisonType
```

---

comparisonOperator

```
protected java.lang.String comparisonOperator
```

---

threshold

```
protected double threshold
```

---

annot

```
protected Annotation annot
```

Constructor Detail

SubSelection

```
public SubSelection()
```

Method Detail

setSelectByIdentifier

```
public void setSelectByIdentifier()
```

Specifies that a subselection shall be made based on identifier names.

---

selectByIdentifier

```
public boolean selectByIdentifier()
```

[illegible]

Returns true if a subselection is to be made based on identifier names.
Returns:
true if subselection is to be based on identifiers

---

setSelectByAnnotation

```
public void setSelectByAnnotation()
```
Specifies that a subselection shall be made based on annotations.

---

selectByAnnotation

```
public boolean selectByAnnotation()
```
Returns true if a subselection is to be made based on annotations.
Returns:
true if subselection is to be based on annotations

---

setAllSelectToFalse

```
private void setAllSelectToFalse()
```
Sets all selection flags to false.

---

getStringSelections

```
public java.lang.String[] getStringSelections()
```
Returns an array of the strings that are to be selected for. Used for qualitative selections by annotation or identifier.
Returns:
an array of the strings that are to be used for subselection.

---

setStringSelections

```
public void setStringSelections(java.lang.String[] stringSelections)
```
Sets the array of the strings that are to be selected for. Used for qualitative selections by annotation or identifier.
Parameters:
`stringSelections` - the array of the strings that are to be selected for

---

getAnnotationType

```
public java.lang.String getAnnotationType()
```
Returns the type of the annotation that is to be used for subselection.
Returns:
the type of the annotation that is to be used for subselection.

---

setAnnotationType

```
public void setAnnotationType(java.lang.String annotationType)
```
Sets the type of the annotation that is to be used for subselection.
Parameters:
`annotationType` - the type of the annotation that is to be used for subselection.

---

getAnnotTypeNumber

```
public int getAnnotTypeNumber()
```
Returns the number of the annotation type that is to be used for subselection.
Returns:

[illegible]

the number of the annotation type that is to be used for subselection.

---

setAnnotTypeNumber

`public void` **`setAnnotTypeNumber`**`(int annotTypeNumber)`

Sets the number of the annotation type that is to be used for subselection.

Parameters:

`annotTypeNumber` - the number of the annotation type that is to be used for subselection.

---

getComparisonType

`public java.lang.String` **`getComparisonType`**`()`

Returns the comparison type ('qualitative' or 'quantitative') that is to be used in the subselection.

Returns:

the comparison type that is to be used in the subselection.

---

setComparisonType

`public void` **`setComparisonType`**`(java.lang.String comparisonType)`

Sets the comparison type ('qualitative' or 'quantitative') that is to be used in the subselection.

Parameters:

`comparisonType` - the comparison type that is to be used in the subselection.

---

getComparisonOperator

`public java.lang.String` **`getComparisonOperator`**`()`

Returns the comparison operator (>, <, =, etc) that is to be used for a quantitative subselection.

Returns:

the comparison operator that is to be used for a quantitative subselection.

---

setComparisonOperator

`public void` **`setComparisonOperator`**`(java.lang.String comparisonOperator)`

Sets the comparison operator (>, <, =, etc) that is to be used for a quantitative subselection.

Parameters:

`comparisonOperator` - the comparison operator that is to be used for a quantitative subselection.

---

getThreshold

`public double` **`getThreshold`**`()`

Returns the threshold that is to be used for a quantitative subselection.

Returns:

the threshold that is to be used for a quantitative subselection.

---

setThreshold

`public void` **`setThreshold`**`(double threshold)`

Sets the threshold that is to be used for a quantitative subselection.

Parameters:

`threshold` - the threshold that is to be used for a quantitative subselection.

---

getAnnotation

[illegible]

public Annotation **getAnnotation**()

Returns the Annotation object that is to be used in an annotation based subselection.

Returns:

the Annotation object that is to be used in an annotation based subselection.

---

setAnnotation

public void **setAnnotation**(Annotation annotation)

Sets the Annotation object that is to be used in an annotation based subselection.

Parameters:

`annotation` - the Annotation object that is to be used in an annotation based subselection.

---

getDescription

public java.lang.String **getDescription**()

Returns a string description of the subselection criteria. This string is in HTML format, ready to be printed in a web browser.

Returns:

a string description of the subselection criteria.

---

clone

public java.lang.Object **clone**()

Implements the clone method to allow a deep copy of a subselection item.

Overrides:

`clone` in class `java.lang.Object`

Returns:

the cloned SubSelection object.

---

equals

public boolean **equals**(java.lang.Object obj)

Overrides the equals method to perform a deep comparison between the receiver and the passed object. Returns true if two SubSelection objects are selecting for the same criteria. The comparison information (identifiers vs annotations), type (quantitive vs qualitative), and comparison type (=, >, etc - if used) must be the same.

Overrides:

`equals` in class `java.lang.Object`

Parameters:

`obj` - the SubSelection object to compare.

Returns:

true if the two SubSelection objects are equal.

---

infoString

public java.lang.String **infoString**()

Returns a string containing the subselection criteria specified by the object properties. The string is in HTML format, and is ready to be printed in a web browser.

Returns:

a string containing the subselection criteria

---

edu.ucsf.Magellan
Class SystemInfo

```
java.lang.Object
  └─ edu.ucsf.Magellan.SystemInfo
```

[illegible]

```
public class SystemInfo
extends java.lang.Object
```

SystemInfo objects store information about the current system implementation of Magellan, such as database connection information, paths to necessary applications, etc.

Field Summary	
private static java.lang.String	adminEmail
private static boolean	connected
private static java.lang.String	dbConnectionString
private static java.lang.String	pageFooter
private static java.lang.String	Rpath

Constructor Summary
SystemInfo()

Method Summary	
static void	closeDbConnections(java.sql.Connection conn, java.sql.Statement stmt, java.sql.ResultSet rs) Closes the Connection, Statement, and ResultSet objects associated with a pooled database connection.
static boolean	DBconnected() Returns true if a successful connection was made to the Oracle database.
static java.lang.String	getAdminEmail() Returns the email address of the Magellan administrator
static java.sql.Connection	getDBconnection() Returns the database Connection object from the DBCP connection pool.
static java.lang.String	getDBconnectionString() Returns the database connection string used for applications like sqlldr
static java.lang.String	getPageFooter() Returns a string containing HTML for a footer to appear at the bottom of web pages.
static java.lang.String	getRpath() Returns the path to the R application
static void	setAdminEmail(java.lang.String email) Sets the email address of the Magellan administrator
static void	setDBconnectionString(java.lang.String _dbConnectionString) Sets the database connection string used for applications like sqlldr
static void	setPageFooter(java.lang.String footer) Sets a string containing HTML for a footer to appear at the bottom of web pages.
static void	setRpath(java.lang.String path) Sets the path to the R application

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

[illegible]

Field Detail

connected

```
private static boolean connected
```

Rpath

```
private static java.lang.String Rpath
```

---

dbConnectionString

```
private static java.lang.String dbConnectionString
```

---

pageFooter

```
private static java.lang.String pageFooter
```

---

adminEmail

```
private static java.lang.String adminEmail
```

Constructor Detail

SystemInfo

```
public SystemInfo()
```

Method Detail

getDBconnection

```
public static java.sql.Connection getDBconnection()
                                           throws java.sql.SQLException
```

Returns the database Connection object from the DBCP connection pool.

Returns:
the database Connection object

Throws:
`java.sql.SQLException`

---

closeDbConnections

```
public static void closeDbConnections(java.sql.Connection conn,
                                      java.sql.Statement stmt,
                                      java.sql.ResultSet rs)
```

Closes the Connection, Statement, and ResultSet objects associated with a pooled database connection. If the objects are not null, then close() method is executed for each and then each one is set to null so that the connection can be returned to the pool.

Parameters:
`conn` - the database Connection object to be closed
`stmt` - the database Statement object to be closed
`rs` - the database ResultSet object to be closed

---

getDBconnectionString

[illegible]

`public static java.lang.String` **`getDBconnectionString`**`()`

Returns the database connection string used for applications like sqlldr

Returns:

the database connection string

---

setDBconnectionString

`public static void` **`setDBconnectionString`**`(java.lang.String _dbConnectionString)`

Sets the database connection string used for applications like sqlldr

Parameters:

`_dbConnectionString` - the database connection string

DBconnected

`public static boolean` **`DBconnected`**`()`

Returns true if a successful connection was made to the Oracle database.

Returns:

true if a successful connection was made to the Oracle database.

---

getPageFooter

`public static java.lang.String` **`getPageFooter`**`()`

Returns a string containing HTML for a footer to appear at the bottom of web pages.

Returns:

an HTML string containing a footer for web pages.

---

setPageFooter

`public static void` **`setPageFooter`**`(java.lang.String footer)`

Sets a string containing HTML for a footer to appear at the bottom of web pages.

Parameters:

`footer` - an HTML string containing a footer for web pages.

---

setRpath

`public static void` **`setRpath`**`(java.lang.String path)`

Sets the path to the R application

Parameters:

`path` - the path to the R application

---

getRpath

`public static java.lang.String` **`getRpath`**`()`

Returns the path to the R application

Returns:

the path to the R application

---

setAdminEmail

`public static void` **`setAdminEmail`**`(java.lang.String email)`

Sets the email address of the Magellan administrator

Parameters:

`email` - the email address of the Magellan administrator

[illegible]

getAdminEmail

```
public static java.lang.String getAdminEmail()
```

Returns the email address of the Magellan administrator

Returns:

the email address of the Magellan administrator

edu.ucsf.Magellan
Class SystemInfoHandler

```
java.lang.Object
  └─ org.xml.sax.helpers.DefaultHandler
       └─ edu.ucsf.Magellan.SystemInfoHandler
```

All Implemented Interfaces:

org.xml.sax.ContentHandler, org.xml.sax.DTDHandler, org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler

```
public class SystemInfoHandler
extends org.xml.sax.helpers.DefaultHandler
```

The SystemInfoHandler class extends DefaultHandler to parse the XML file 'magellan.xml' containing database and other parameters for Magellan.

Field Summary	
private java.lang.String	adminEmail
private java.lang.StringBuffer	charDataBuffer
private java.lang.String	contextPath
private java.lang.String	dbHost
private java.lang.String	dbInstance
private java.lang.String	dbLogin
private java.lang.String	dbPassword
private java.lang.String	dbService
private java.lang.String	pageFooter
private java.lang.String	rPath

Constructor Summary
SystemInfoHandler()

Method Summary	
void	characters(char[] text, int start, int length)
void	endElement(java.lang.String namespaceURI, java.lang.String localName,

[illegible]

	java.lang.String qName)
java.lang.String	**getAdminEmail**() Returns the email address of the magellan administrator specified in magellan.xml
java.lang.String	**getContextPath**() Returns the context path of the magellan instance.
java.lang.String	**getDbHost**() Returns the oracle database host specified in magellan.xml
java.lang.String	**getDbInstance**() Returns the oracle database instance specified in magellan.xml
java.lang.String	**getDbLogin**() Returns the oracle database login specified in magellan.xml
java.lang.String	**getDbPassword**() Returns the oracle database password specified in magellan.xml
java.lang.String	**getDbService**() Returns the oracle database service name specified in magellan.xml
java.lang.String	**getPageFooter**() Returns the web page footer specified in magellan.xml
java.lang.String	**getRpath**() Returns the path of the application 'rterm.exe' specified in magellan.xml
void	**startElement**(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, org.xml.sax.Attributes atts)

Methods inherited from class org.xml.sax.helpers.DefaultHandler
endDocument, endPrefixMapping, error, fatalError, ignorableWhitespace, notationDecl, processingInstruction, resolveEntity, setDocumentLocator, skippedEntity, startDocument, startPrefixMapping, unparsedEntityDecl, warning

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

charDataBuffer

```
private java.lang.StringBuffer charDataBuffer
```

dbHost

```
private java.lang.String dbHost
```

dbInstance

```
private java.lang.String dbInstance
```

dbService

```
private java.lang.String dbService
```

[illegible]

dbLogin

```
private java.lang.String dbLogin
```

---

dbPassword

```
private java.lang.String dbPassword
```

---

rPath

```
private java.lang.String rPath
```

---

pageFooter

```
private java.lang.String pageFooter
```

---

adminEmail

```
private java.lang.String adminEmail
```

---

contextPath

```
private java.lang.String contextPath
```

Constructor Detail

SystemInfoHandler

```
public SystemInfoHandler()
```

Method Detail

getDbHost

```
public java.lang.String getDbHost()
```

Returns the oracle database host specified in magellan.xml
Returns:
the oracle database host specified in magellan.xml

---

getDbInstance

```
public java.lang.String getDbInstance()
```

Returns the oracle database instance specified in magellan.xml
Returns:
the oracle database instance specified in magellan.xml

---

getDbService

```
public java.lang.String getDbService()
```

Returns the oracle database service name specified in magellan.xml
Returns:
the oracle database service name specified in magellan.xml

[illegible]

getDbLogin

```
public java.lang.String getDbLogin()
```

Returns the oracle database login specified in magellan.xml

Returns:

the oracle database login specified in magellan.xml

---

getDbPassword

```
public java.lang.String getDbPassword()
```

Returns the oracle database password specified in magellan.xml

Returns:

the oracle database password specified in magellan.xml

---

getRpath

```
public java.lang.String getRpath()
```

Returns the path of the application 'rterm.exe' specified in magellan.xml

Returns:

the path of the application 'rterm.exe' specified in magellan.xml

---

getContextPath

```
public java.lang.String getContextPath()
```

Returns the context path of the magellan instance. This is the path of the magellan web application instance provided when the web app was installed in the Tomcat manager. All names of installed web applications can be viewed in the Tomcat manager web app.

Returns:

the context path of the magellan instance

---

getPageFooter

```
public java.lang.String getPageFooter()
```

Returns the web page footer specified in magellan.xml

Returns:

the web page footer specified in magellan.xml

---

getAdminEmail

```
public java.lang.String getAdminEmail()
```

Returns the email address of the magellan administrator specified in magellan.xml

Returns:

the email address of the magellan administrator specified in magellan.xml

---

startElement

```
public void startElement(java.lang.String namespaceURI,
                         java.lang.String localName,
                         java.lang.String qName,
                         org.xml.sax.Attributes atts)
                  throws org.xml.sax.SAXException
```

Specified by:

`startElement` in interface `org.xml.sax.ContentHandler`

Overrides:

`startElement` in class `org.xml.sax.helpers.DefaultHandler`

Throws:

`org.xml.sax.SAXException`

[illegible]

endElement

```
public void endElement(java.lang.String namespaceURI,
                       java.lang.String localName,
                       java.lang.String qName)
                throws org.xml.sax.SAXException
```

Specified by:
`endElement` in interface `org.xml.sax.ContentHandler`
Overrides:
`endElement` in class `org.xml.sax.helpers.DefaultHandler`
Throws:
`org.xml.sax.SAXException`

characters

```
public void characters(char[] text,
                       int start,
                       int length)
                throws org.xml.sax.SAXException
```

Specified by:
`characters` in interface `org.xml.sax.ContentHandler`
Overrides:
`characters` in class `org.xml.sax.helpers.DefaultHandler`
Throws:
`org.xml.sax.SAXException`

edu.ucsf.Magellan
Class UploadInfo

```
java.lang.Object
  └─ edu.ucsf.Magellan.UploadInfo
```

```
public class UploadInfo
extends java.lang.Object
```

UploadInfo objects implement the core functionality for uploads of data and annotations to the database. Each session involving an upload has one instantiated UploadInfo object associated with it. UploadInfo objects store user information as well as other information relevant to the upload of information to the database. UploadInfo methods are used to extract information from data files uploaded from client to server, prepare sqlldr upload files, and load/delete information to the database.

Field Summary	
private int	**annotIdentifierEnd**
private java.lang.String	**annotIdentifierFormat**
private int	**annotIdentifierLocation**
private int	**annotIdentifierStart**
private java.lang.String	**annotIdentifierType**
private int	**dataTypeNumber**

[illegible]

private java.lang.String	**delimiter**
private java.lang.String	**description**
private int	**experiment**
private java.lang.String	**lab**
private boolean	**newExperiment**
private int	**numFileCols**
private int	**numFileRows**
private int	**numSamples**
private java.util.Vector	**prevUploadedDataTypes**
private boolean	**prevUploadedFormat**
private int	**sampleNamesEnd**
private java.lang.String	**sampleNamesFormat**
private int	**sampleNamesLocation**
private int	**sampleNamesSkip**
private int	**sampleNamesStart**
private boolean	**saveFileFormat**
private boolean	**singleSample**
private java.lang.String	**singleSampleName**
private int	**upload**
private java.util.Vector	**uploadList**
private boolean	**uploadPublic**
private java.lang.String	**uploadType**
private java.lang.String	**user**

Constructor Summary
UploadInfo()

Method Summary

[illegible]

void	**addToUploadList**(UploadItem upld) Adds an UploadItem object to the vector of those objects that make up the current upload.
private java.lang.String	**annotationInfoString**() Returns a string of HTML code that prints out the current information about the annotations to be uploaded to the database.
void	**calculateNumSamples**() Calculates the number of samples in the upload file based on the location information entered by the user.
void	**clearAllUploadInfo**() Clears all the information about the current upload.
void	**clearUploadList**() Clears the vector of UploadItem objects corresponding to those objects that make up the current upload.
private java.lang.String	**dataTypeAnnotationString**(UploadItem dt, int dtNum) Returns a string of HTML code that prints out the current information about annotations associated with one data type to be uploaded.
private java.lang.String	**dataTypeIdentifierInfoString**(UploadItem dt, int i) Returns a string of HTML code that prints out the current information about one data type to be uploaded.
private java.lang.String	**dataTypeInfoString**() Returns a string of HTML code that prints out the current information about the data types to be uploaded to the database.
boolean	**dataTypePrevUploaded**(java.lang.String dataTypeName) Returns true if the passed data type name has already been used in the current experiment.
static void	**deleteExperiment**(int experimentNum) Deletes the experiment with the specified experiment number from the database.
static void	**deleteUpload**(int uploadID) Deletes information with the specified upload ID number from the database.
boolean	**experimentExists**(int experiment) Returns true if the passed experiment number exists in the database.
boolean	**experimentExists**(java.lang.String experimentString) Returns true if the passed string contains an integer corresponding to an experiment number that exists in the database.
int	**getAnnotIdentifierEnd**() Returns the ending row or column of the identifiers in an annotation upload.
java.lang.String	**getAnnotIdentifierFormat**() Returns the format of identifiers for an upload of annotations ('row' or 'column').
int	**getAnnotIdentifierLocation**() Returns the row or column number where identifiers are located in an annotation upload.
int	**getAnnotIdentifierStart**() Returns the starting row or column of the identifiers in an annotation upload.
java.lang.String	**getAnnotIdentifierType**() Returns the type of identifiers for an upload of annotations.
static java.lang.String[]	**getArrayFromFile**(java.lang.String filePath, java.lang.String delimiter, java.lang.String lineFormat, int location, int start, int end, int skip) Returns a string array from a file given the position or the row or column specified in the parameters.
static java.lang.String[]	**getArrayFromLine**(java.lang.String line, java.lang.String delimiter, int start, int end, int skip) Takes an input string, splits it using the passed delimiter and returns a subset of the resulting elements as a string array.
int	**getDataTypeNumber**() Returns the data type number for an upload of derived annotations.

[illegible]

java.util.HashMap	**getDataTypesFromExperiment**(int experiment) Returns a HashMap object in which the keys are the names of the data types from the passed experiment number and the values are the number of variables per sample of each data type.
java.lang.String	**getDelimiter**() Returns the file delimiter of the current upload ('\t', ',', or ' ')
java.lang.String	**getDescription**() Returns the description of the current upload
int	**getExperimentNumber**() Returns the experiment number for the current upload of data or derived annotations.
void	**getFileDimensions**(java.lang.String filePath) Gets the number of rows and columns of information of the file whose full path is passed as a paramter.
void	**getFileDimensions**(java.lang.String filePath, int firstRows, int lastRows) Gets the number of rows and columns of information of the file whose full path is passed as a paramter, skipping the firstRows number of rows at the beginning of the file and lastRows number of rows at the end of the file.
static java.lang.String	**getItemFromLine**(java.lang.String line, java.lang.String delimiter, int index) Takes an input string, splits it using the passed delimiter and returns a single element of the resulting array at the specified index.
java.lang.String	**getLab**() Returns the lab name for the current user
static int	**getNextAvailableExperimentNumber**() Returns the next available experiment number that can be used to store information in the database.
static int	**getNextAvailableUploadNumber**() Returns the next available upload ID number that can be used to store information in the database.
int	**getNumFileCols**() Returns the number of columns in the uploaded file.
int	**getNumFileRows**() Returns the number of rows in the uploaded file.
int	**getNumSamples**() Returns the number of samples in the upload file.
java.util.Vector	**getPrevUploadedDataTypes**() Returns a vector containing the DataType objects that have been previously uploaded as part of the current experiment.
int	**getSampleNamesEnd**() Returns the ending row or column of the sample names in the upload file.
java.lang.String	**getSampleNamesFormat**() Returns the format of the sample names in the data file uploaded by the user ('row' or 'column').
int	**getSampleNamesLocation**() Returns the row or column number where the sample names are located in the upload file.
int	**getSampleNamesSkip**() Returns the number of rows or columns to skip between sample names in the upload file.
int	**getSampleNamesStart**() Returns the starting row or column of the sample names in the upload file.
java.lang.String	**getSingleSampleName**() Returns the name of the sample for an upload of a single sample's worth of data.
UploadItem	**getUploadInList**(java.lang.String uploadName) Returns an UploadItem object of the passed uploadName from the list of items to upload, or null if an UploadItem object of that name does not exist in the list.

[illegible]

java.util.Vector	**getUploadList**() Returns a vector of UploadItem objects corresponding to those objects that make up the current upload.
int	**getUploadNumber**() Returns the upload number for the current upload.
java.lang.String	**getUploadType**() Returns the type of the current upload ('data', 'annotation', or 'statistics')
java.lang.String	**getUser**() Returns the login (email address) for the current user
private java.lang.String	**identifierInfoString**() Returns a string of HTML code that prints out the current information about the identifiers to be uploaded to the database.
boolean	**inUploadList**(java.lang.String uploadName) Returns true if the passed upload item name has already been used in the list of uploaded items.
boolean	**isPublic**() Returns true if the current upload is to be made publicly available, false otherwise.
static void	**LoadUploadTuple**(int uploadID, int experimentNumber, java.lang.String type, java.lang.String userName, java.lang.String labName, java.lang.String delimiter, java.lang.String accessType, java.lang.String description) Loads information about the current upload into the Upload table of the database.
static void	**LoadUploadTuple**(UploadInfo Upload, int uploadID, java.lang.String type) Loads information about the current upload into the Upload table of the database.
boolean	**newExperiment**() Returns true if the current upload contains data for a new experiment, as opposed to an addition to an old experiment.
boolean	**prevUploadedFormat**() Returns true if the current upload uses a file format of a previous upload, false otherwise.
static void	**printUploadFileHeader**(java.io.FileWriter DBfile) Prints a header for sqlldr that specifies how information in upload files is loaded into the database tables.
void	**removeFromUploadList**(int position) Removes an UploadItem object at the specified position in the vector of those objects that make up the current upload.
private java.lang.String	**sampleInfoString**() Returns a string of HTML code that prints out the current information about the samples to be uploaded to the database.
boolean	**sampleNamesSpecified**() Returns true if the sample names for the current upload have been specified.
boolean	**saveFileFormat**() Returns true if the file format of the current upload will be saved to the database, false otherwise.
void	**setAnnotIdentifierEnd**(int annotIdentifierEnd) Sets the ending row or column of the identifiers in an annotation upload.
void	**setAnnotIdentifierFormat**(java.lang.String annotIdentifierFormat) Sets the format of identifiers for an upload of annotations ('row' or 'column').
void	**setAnnotIdentifierLocation**(int annotIdentifierLocation) Returns the row or column number where identifiers are located in an annotation upload.
void	**setAnnotIdentifierStart**(int annotIdentifierStart) Sets the starting row or column of the identifiers in an annotation upload.
void	**setAnnotIdentifierType**(java.lang.String annotIdentifierType) Sets the type of identifiers for an upload of annotations.
void	**setDataTypeNumber**(int dataTypeNumber) Sets the data type number of the data type that are associated with an upload of derived

[illegible]

	annotations.
void	**setDelimiter**(java.lang.String delimiter) Sets the file delimiter of the current upload.
void	**setDescription**(java.lang.String description) Sets the description of the current upload
void	**setExperimentNumber**(int experiment) Sets the experiment number for the current upload of data or derived annotations.
void	**setLab**(java.lang.String lab) Sets the lab name for the current user
void	**setNewExperiment**(boolean newExperiment) Sets to true if the current upload contains data for a new experiment, as opposed to an addition to an old experiment.
void	**setNumSamples**(int numSamples) Sets the number of samples in the upload file.
void	**setPrevUploadedFormat**(boolean prevUploaded) Sets to true if the current upload uses a file format of a previous upload, false otherwise..
void	**setPublic**(boolean uploadPublic) Sets to true if the current upload is to be made publicly available, false otherwise.
void	**setSampleNamesEnd**(int sampleNamesEnd) Returns the ending row or column of the sample names in the upload file.
void	**setSampleNamesFormat**(java.lang.String sampleNamesFormat) Sets the format of the sample names in the data file uploaded by the user ('row' or 'column').
void	**setSampleNamesLocation**(int sampleNamesLocation) Sets the row or column number where the sample names are located in the upload file.
void	**setSampleNamesSkip**(int sampleNamesSkip) Sets the number of rows or columns to skip between sample names in the upload file.
void	**setSampleNamesStart**(int sampleNamesStart) Sets the starting row or column of the sample names in the upload file.
void	**setSaveFileFormat**(boolean saveFileFormat) Sets to true if the file format of the current upload will be saved to the database, false otherwise.
void	**setSingleSample**(boolean singleSample) Sets a boolen flag to true if the current upload contains data from a single sample.
void	**setSingleSampleName**(java.lang.String singleSampleName) Sets the name of the sample for an upload of a single sample's worth of data.
void	**setUploadNumber**(int upload) Sets the upload number for the current upload.
void	**setUploadType**(java.lang.String type) Sets the type of the current upload.
void	**setUser**(java.lang.String user) Sets the login (email address) for the current user
boolean	**singleSample**() Returns true if the current upload contains data from a single sample.
java.lang.String	**uploadButtonString**() Returns a string of HTML code that prints out the a message and HTML button the user can click to upload information to the database.
boolean	**uploadExists**(int uploadID) Returns true if the passed upload number exists in the database.
static boolean	**uploadExists**(int uploadID, java.lang.String content) Returns true if an upload of the passed content (data, annotations, statistics, etc) and upload number exists in the database.
boolean	**uploadExists**(java.lang.String uploadIDString) Returns true if the passed string contains an integer corresponding to an upload number

[illegible]

	that exists in the database.
`java.lang.String`	`uploadInfoString()` Returns a string of HTML code that prints out the current information about the items to be uploaded to the database.
`static void`	`UploadStatisticsFile(java.lang.String statFilePath, java.lang.String tempFilePath, java.lang.String type, java.lang.String description, java.lang.String userName, java.lang.String labName, int experimentNumber, int dataTypeNumber, int linesToSkip, int ordPosCol, int resultsCol)` Uploads the contents of a statistics file to the database as a derived annotation.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

user

`private java.lang.String` **`user`**

---

lab

`private java.lang.String` **`lab`**

---

description

`private java.lang.String` **`description`**

---

uploadType

`private java.lang.String` **`uploadType`**

---

delimiter

`private java.lang.String` **`delimiter`**

---

sampleNamesFormat

`private java.lang.String` **`sampleNamesFormat`**

---

singleSampleName

`private java.lang.String` **`singleSampleName`**

---

annotIdentifierType

`private java.lang.String` **`annotIdentifierType`**

---

annotIdentifierFormat

`private java.lang.String` **`annotIdentifierFormat`**

---

experiment

`private int` **`experiment`**

---

upload

`private int` **`upload`**

---

sampleNamesStart

`private int` **`sampleNamesStart`**

---

sampleNamesEnd

`private int` **`sampleNamesEnd`**

---

sampleNamesSkip

`private int` **`sampleNamesSkip`**

---

sampleNamesLocation

`private int` **`sampleNamesLocation`**

---

numSamples

`private int` **`numSamples`**

---

numFileRows

`private int` **`numFileRows`**

---

numFileCols

`private int` **`numFileCols`**

---

annotIdentifierStart

`private int` **`annotIdentifierStart`**

---

annotIdentifierEnd

`private int` **`annotIdentifierEnd`**

[illegible]

annotIdentifierLocation

```
private int annotIdentifierLocation
```

dataTypeNumber

```
private int dataTypeNumber
```

singleSample

```
private boolean singleSample
```

newExperiment

```
private boolean newExperiment
```

prevUploadedFormat

```
private boolean prevUploadedFormat
```

saveFileFormat

```
private boolean saveFileFormat
```

uploadPublic

```
private boolean uploadPublic
```

uploadList

```
private java.util.Vector uploadList
```

prevUploadedDataTypes

```
private java.util.Vector prevUploadedDataTypes
```

Constructor Detail

UploadInfo

```
public UploadInfo()
```

Method Detail

getUser

```
public java.lang.String getUser()
```

Returns the login (email address) for the current user

Returns:

the login of the current user

---

setUser

```
public void setUser(java.lang.String user)
```
Sets the login (email address) for the current user
Parameters:
`user` - the login of the current user

---

getLab

```
public java.lang.String getLab()
```
Returns the lab name for the current user
Returns:
the lab name of the current user

---

setLab

```
public void setLab(java.lang.String lab)
```
Sets the lab name for the current user
Parameters:
`lab` - the lab name of the current user

---

getDescription

```
public java.lang.String getDescription()
```
Returns the description of the current upload
Returns:
the description of the current upload

---

setDescription

```
public void setDescription(java.lang.String description)
```
Sets the description of the current upload
Parameters:
`description` - the description of the current upload

---

getUploadType

```
public java.lang.String getUploadType()
```
Returns the type of the current upload ('data', 'annotation', or 'statistics')
Returns:
the type of the current upload

---

setUploadType

```
public void setUploadType(java.lang.String type)
```
Sets the type of the current upload. Should be one of 'data', 'annotation', or 'statistics'.
Parameters:
`type` - the type of the current upload.

---

getDelimiter

[illegible]

`public java.lang.String` **`getDelimiter`**`()`

Returns the file delimiter of the current upload ('\t', ',', or ' ')

Returns:

the file delimiter of the current upload

---

setDelimiter

`public void` **`setDelimiter`**`(java.lang.String delimiter)`

Sets the file delimiter of the current upload. Should be one of '\t', ',', or ' '.

Parameters:

`delimiter` - the file delimiter of the current upload

---

getSampleNamesFormat

`public java.lang.String` **`getSampleNamesFormat`**`()`

Returns the format of the sample names in the data file uploaded by the user ('row' or 'column').

Returns:

the format of the sample names in the file ('row' or 'column').

---

setSampleNamesFormat

`public void` **`setSampleNamesFormat`**`(java.lang.String sampleNamesFormat)`

Sets the format of the sample names in the data file uploaded by the user ('row' or 'column').

Parameters:

`sampleNamesFormat` - the format of the sample names in the file

---

getSingleSampleName

`public java.lang.String` **`getSingleSampleName`**`()`

Returns the name of the sample for an upload of a single sample's worth of data.

Returns:

the name of the sample for an upload of a single sample's worth of data

---

setSingleSampleName

`public void` **`setSingleSampleName`**`(java.lang.String singleSampleName)`

Sets the name of the sample for an upload of a single sample's worth of data.

Parameters:

`singleSampleName` - the name of the sample for an upload of a single sample's worth of data.

---

getAnnotIdentifierType

`public java.lang.String` **`getAnnotIdentifierType`**`()`

Returns the type of identifiers for an upload of annotations.

Returns:

the type of identifiers for an upload of annotations.

---

setAnnotIdentifierType

`public void` **`setAnnotIdentifierType`**`(java.lang.String annotIdentifierType)`

Sets the type of identifiers for an upload of annotations.

Parameters:

`annotIdentifierType` - the type of identifiers for an upload of annotations.

---

getAnnotIdentifierFormat

`public java.lang.String` **`getAnnotIdentifierFormat`**`()`

Returns the format of identifiers for an upload of annotations ('row' or 'column').

Returns:

the format of identifiers for an upload of annotations ('row' or 'column').

---

setAnnotIdentifierFormat

`public void` **`setAnnotIdentifierFormat`**`(java.lang.String annotIdentifierFormat)`

Sets the format of identifiers for an upload of annotations ('row' or 'column').

Parameters:

`annotIdentifierFormat` - the format of identifiers for an upload of annotations ('row' or 'column').

---

getExperimentNumber

`public int` **`getExperimentNumber`**`()`

Returns the experiment number for the current upload of data or derived annotations.

Returns:

the experiment number for the current upload of data or derived annotations.

---

setExperimentNumber

`public void` **`setExperimentNumber`**`(int experiment)`

Sets the experiment number for the current upload of data or derived annotations.

Parameters:

`experiment` - the experiment number for the current upload of data or derived annotations.

---

getUploadNumber

`public int` **`getUploadNumber`**`()`

Returns the upload number for the current upload.

Returns:

the upload number for the current upload.

---

setUploadNumber

`public void` **`setUploadNumber`**`(int upload)`

Sets the upload number for the current upload.

Parameters:

`upload` - the upload number for the current upload.

---

getSampleNamesStart

`public int` **`getSampleNamesStart`**`()`

Returns the starting row or column of the sample names in the upload file.

Returns:

the starting row or column of the sample names in the upload file.

---

setSampleNamesStart

`public void` **`setSampleNamesStart`**`(int sampleNamesStart)`

Sets the starting row or column of the sample names in the upload file.

Parameters:

`sampleNamesStart` - the starting row or column of the sample names in the upload file.

[illegible]

getSampleNamesEnd

`public int getSampleNamesEnd()`

Returns the ending row or column of the sample names in the upload file.
Returns:
the ending row or column of the sample names in the upload file.

setSampleNamesEnd

`public void setSampleNamesEnd(int sampleNamesEnd)`

Returns the ending row or column of the sample names in the upload file.
Parameters:
`sampleNamesEnd` - the ending row or column of the sample names in the upload file.

getSampleNamesSkip

`public int getSampleNamesSkip()`

Returns the number of rows or columns to skip between sample names in the upload file.
Returns:
the number of rows or columns to skip between sample names in the upload file.

setSampleNamesSkip

`public void setSampleNamesSkip(int sampleNamesSkip)`

Sets the number of rows or columns to skip between sample names in the upload file.
Parameters:
`sampleNamesSkip` - the number of rows or columns to skip between sample names in the upload file.

getSampleNamesLocation

`public int getSampleNamesLocation()`

Returns the row or column number where the sample names are located in the upload file.
Returns:
the row or column number where the sample names are located in the upload file.

setSampleNamesLocation

`public void setSampleNamesLocation(int sampleNamesLocation)`

Sets the row or column number where the sample names are located in the upload file.
Parameters:
`sampleNamesLocation` - the row or column number where the sample names are located in the upload file.

getNumSamples

`public int getNumSamples()`

Returns the number of samples in the upload file.
Returns:
the number of samples in the upload file.

setNumSamples

`public void setNumSamples(int numSamples)`

[illegible]

Sets the number of samples in the upload file.
Parameters:
`numSamples` - the number of samples in the upload file.

---

calculateNumSamples

`public void` **`calculateNumSamples`**`()`

Calculates the number of samples in the upload file based on the location information entered by the user. The number of samples is stored as a property of the UploadInfo object.

---

getAnnotIdentifierStart

`public int` **`getAnnotIdentifierStart`**`()`

Returns the starting row or column of the identifiers in an annotation upload.
Returns:
the starting row or column of the identifiers in an annotation upload.

---

setAnnotIdentifierStart

`public void` **`setAnnotIdentifierStart`**`(int annotIdentifierStart)`

Sets the starting row or column of the identifiers in an annotation upload.
Parameters:
`annotIdentifierStart` - the starting row or column of the identifiers in an annotation upload.

---

getAnnotIdentifierEnd

`public int` **`getAnnotIdentifierEnd`**`()`

Returns the ending row or column of the identifiers in an annotation upload.
Returns:
the ending row or column of the identifiers in an annotation upload.

---

setAnnotIdentifierEnd

`public void` **`setAnnotIdentifierEnd`**`(int annotIdentifierEnd)`

Sets the ending row or column of the identifiers in an annotation upload.
Parameters:
`annotIdentifierEnd` - the ending row or column of the identifiers in an annotation upload.

---

getAnnotIdentifierLocation

`public int` **`getAnnotIdentifierLocation`**`()`

Returns the row or column number where identifiers are located in an annotation upload.
Returns:
the row or column number where identifiers are located in an annotation upload.

---

setAnnotIdentifierLocation

`public void` **`setAnnotIdentifierLocation`**`(int annotIdentifierLocation)`

Returns the row or column number where identifiers are located in an annotation upload.
Parameters:
`annotIdentifierLocation` - the row or column number where identifiers are located in an annotation upload.

---

getDataTypeNumber

`public int getDataTypeNumber()`

Returns the data type number for an upload of derived annotations. The annotations will be associated with this data type in the database.

Returns:

the data type number for an upload of derived annotations.

---

setDataTypeNumber

`public void setDataTypeNumber(int dataTypeNumber)`

Sets the data type number of the data type that are associated with an upload of derived annotations.

Parameters:

`dataTypeNumber` - the data type number for an upload of derived annotations.

---

getNumFileRows

`public int getNumFileRows()`

Returns the number of rows in the uploaded file.

Returns:

the number of rows in the uploaded file.

---

getNumFileCols

`public int getNumFileCols()`

Returns the number of columns in the uploaded file.

Returns:

the number of columns in the uploaded file.

---

singleSample

`public boolean singleSample()`

Returns true if the current upload contains data from a single sample.

Returns:

true if the current upload contains data from a single sample.

---

setSingleSample

`public void setSingleSample(boolean singleSample)`

Sets a boolen flag to true if the current upload contains data from a single sample.

Parameters:

`singleSample` - true if the current upload contains data from a single sample.

---

newExperiment

`public boolean newExperiment()`

Returns true if the current upload contains data for a new experiment, as opposed to an addition to an old experiment.

Returns:

true if upload is part of a new experiment, false otherwise

---

setNewExperiment

`public void setNewExperiment(boolean newExperiment)`

Sets to true if the current upload contains data for a new experiment, as opposed to an addition to an old experiment.

Parameters:

`newExperiment` - true if upload is part of a new experiment, false otherwise

---

[illegible]

prevUploadedFormat

`public boolean` **`prevUploadedFormat`**`()`

Returns true if the current upload uses a file format of a previous upload, false otherwise.
Returns:
true if the current upload uses a file format of a previous upload.

---

setPrevUploadedFormat

`public void` **`setPrevUploadedFormat`**`(boolean prevUploaded)`

Sets to true if the current upload uses a file format of a previous upload, false otherwise..
Parameters:
`prevUploaded` - true if the current upload uses a file format of a previous upload

---

saveFileFormat

`public boolean` **`saveFileFormat`**`()`

Returns true if the file format of the current upload will be saved to the database, false otherwise.
Returns:
true if the file format of the current upload will be saved to the database.

---

setSaveFileFormat

`public void` **`setSaveFileFormat`**`(boolean saveFileFormat)`

Sets to true if the file format of the current upload will be saved to the database, false otherwise.
Parameters:
`saveFileFormat` - true if the file format of the current upload will be saved to the database

---

isPublic

`public boolean` **`isPublic`**`()`

Returns true if the current upload is to be made publicly available, false otherwise.
Returns:
true if the current upload is to be made publicly available.

---

setPublic

`public void` **`setPublic`**`(boolean uploadPublic)`

Sets to true if the current upload is to be made publicly available, false otherwise.
Parameters:
`uploadPublic` - true if the current upload is to be made publicly available.

---

getUploadList

`public java.util.Vector` **`getUploadList`**`()`

Returns a vector of UploadItem objects corresponding to those objects that make up the current upload. These objects correspond to data types (with or without associated annotations), annotations, statistics, etc.
Returns:
a vector of UploadItem objects to be uploaded.

---

addToUploadList

`public void` **`addToUploadList`**`(UploadItem upld)`

Adds an UploadItem object to the vector of those objects that make up the current upload.
Parameters:

[illegible]

`upld` - the UploadItem object to be added to the vector of objects to be uploaded.

---

removeFromUploadList

`public void` **`removeFromUploadList`**`(int position)`

Removes an UploadItem object at the specified position in the vector of those objects that make up the current upload.
Parameters:
`position` - the position to be removed in the vector of UploadItem objects

---

clearUploadList

`public void` **`clearUploadList`**`()`

Clears the vector of UploadItem objects corresponding to those objects that make up the current upload.

---

clearAllUploadInfo

`public void` **`clearAllUploadInfo`**`()`

Clears all the information about the current upload. Items to be cleared include the list of UploadItems to be uploaded, and all sample information.

---

dataTypePrevUploaded

`public boolean` **`dataTypePrevUploaded`**`(java.lang.String dataTypeName)`

Returns true if the passed data type name has already been used in the current experiment. The name is checked against those data types that have been previously loaded in the database.
Parameters:
`dataTypeName` - the data type name to check in the list of previously uploaded data types.
Returns:
true if the passed data type name has already been used in the current experiment.

---

inUploadList

`public boolean` **`inUploadList`**`(java.lang.String uploadName)`

Returns true if the passed upload item name has already been used in the list of uploaded items.
Parameters:
`uploadName` - the name to search for in the list of upload items for the current upload.
Returns:
true if the passed upload item name has already been used in the list of uploaded items.

---

getUploadInList

`public UploadItem` **`getUploadInList`**`(java.lang.String uploadName)`

Returns an UploadItem object of the passed uploadName from the list of items to upload, or null if an UploadItem object of that name does not exist in the list.
Parameters:
`uploadName` - the name to search for in the current list of items to upload.
Returns:
an UploadItem object of the passed uploadName from the list of items to upload

---

getPrevUploadedDataTypes

`public java.util.Vector` **`getPrevUploadedDataTypes`**`()`

Returns a vector containing the DataType objects that have been previously uploaded as part of the current experiment.
Returns:
a vector containing the DataType objects that have been previously uploaded for the current experiment.

[illegible]

sampleNamesSpecified

```
public boolean sampleNamesSpecified()
```

Returns true if the sample names for the current upload have been specified.

Returns:

true if the sample names for the current upload have been specified.

---

getFileDimensions

```
public void getFileDimensions(java.lang.String filePath)
```

Gets the number of rows and columns of information of the file whose full path is passed as a paramter. The delimiter is used to determine column breaks, and the information can be retrieved by using the getNumFileRows() and getNumFileCols() methods.

Parameters:

`filePath` - the full path to the file

---

getFileDimensions

```
public void getFileDimensions(java.lang.String filePath,
                              int firstRows,
                              int lastRows)
```

Gets the number of rows and columns of information of the file whose full path is passed as a paramter, skipping the firstRows number of rows at the beginning of the file and lastRows number of rows at the end of the file. The delimiter is used to determine column breaks, and the information can be retrieved by using the getNumFileRows() and getNumFileCols() methods.

Parameters:

`filePath` - the full path to the file

`firstRows` - the number of rows to skip at the beginning of the file.

`lastRows` - the number of rows to skip at the end of the file.

---

uploadExists

```
public boolean uploadExists(java.lang.String uploadIDString)
                     throws java.sql.SQLException
```

Returns true if the passed string contains an integer corresponding to an upload number that exists in the database.

Parameters:

`uploadIDString` - a string containing an integer corresponding to the upload ID number.

Returns:

true if the passed upload ID exists in the database.

Throws:

`java.sql.SQLException`

---

uploadExists

```
public boolean uploadExists(int uploadID)
                     throws java.sql.SQLException
```

Returns true if the passed upload number exists in the database.

Parameters:

`uploadID` - an integer corresponding to the upload ID number.

Returns:

true if the passed upload ID exists in the database.

Throws:

`java.sql.SQLException`

---

uploadExists

[illegible]

```
public static boolean uploadExists(int uploadID,
                                   java.lang.String content)
                            throws java.sql.SQLException
```

Returns true if an upload of the passed content (data, annotations, statistics, etc) and upload number exists in the database.

Parameters:

`uploadID` - the upload ID number to search for in the database

`content` - the content to search for in the database ('data', 'annotations', 'statistics')

Returns:

true if an upload with the specified upload ID number and content exists in the database.

Throws:

`java.sql.SQLException`

---

experimentExists

```
public boolean experimentExists(java.lang.String experimentString)
                         throws java.sql.SQLException
```

Returns true if the passed string contains an integer corresponding to an experiment number that exists in the database.

Parameters:

`experimentString` - a string containing an integer corresponding to an experiment number

Returns:

true if the experiment number exists in the database

Throws:

`java.sql.SQLException`

---

experimentExists

```
public boolean experimentExists(int experiment)
                         throws java.sql.SQLException
```

Returns true if the passed experiment number exists in the database.

Parameters:

`experiment` - an integer corresponding to an experiment number

Returns:

true if the experiment number exists in the database

Throws:

`java.sql.SQLException`

---

getDataTypesFromExperiment

```
public java.util.HashMap getDataTypesFromExperiment(int experiment)
                                             throws java.sql.SQLException
```

Returns a HashMap object in which the keys are the names of the data types from the passed experiment number and the values are the number of variables per sample of each data type.

Parameters:

`experiment` - the experiment number to query from the database

Returns:

a HashMap with keys=data type names and values=variables per sample.

Throws:

`java.sql.SQLException`

---

uploadInfoString

```
public java.lang.String uploadInfoString()
```

Returns a string of HTML code that prints out the current information about the items to be uploaded to the database. This string can be directly printed in the web browser.

Returns:

HTML code specifying current upload information.

---

identifierInfoString

```
private java.lang.String identifierInfoString()
```

[illegible]

Returns a string of HTML code that prints out the current information about the identifiers to be uploaded to the database.
Returns:
HTML code specifying current identifier upload information.

---

annotationInfoString

```
private java.lang.String annotationInfoString()
```

Returns a string of HTML code that prints out the current information about the annotations to be uploaded to the database.
Returns:
HTML code specifying current annotation upload information.

---

sampleInfoString

```
private java.lang.String sampleInfoString()
```

Returns a string of HTML code that prints out the current information about the samples to be uploaded to the database.
Returns:
HTML code specifying current sample upload information.

---

dataTypeInfoString

```
private java.lang.String dataTypeInfoString()
```

Returns a string of HTML code that prints out the current information about the data types to be uploaded to the database.
Returns:
HTML code specifying current data type upload information.

---

dataTypeIdentifierInfoString

```
private java.lang.String dataTypeIdentifierInfoString(UploadItem dt,
                                                      int i)
```

Returns a string of HTML code that prints out the current information about one data type to be uploaded.
Parameters:
`dt` - an UploadItem object corresponding to a data type to upload.
`i` - the position of the data type in the list of data types to be uploaded
Returns:
HTML code specifying upload information for the specified data type.

---

dataTypeAnnotationString

```
private java.lang.String dataTypeAnnotationString(UploadItem dt,
                                                  int dtNum)
```

Returns a string of HTML code that prints out the current information about annotations associated with one data type to be uploaded.
Parameters:
`dt` - an UploadItem object corresponding to a data type to upload.
`dtNum` - the position of the data type in the list of data types to be uploaded
Returns:
HTML code specifying upload information for the annotations of the specified data type.

---

uploadButtonString

```
public java.lang.String uploadButtonString()
```

Returns a string of HTML code that prints out the a message and HTML button the user can click to upload information to the database.

---

deleteExperiment

[illegible]

```
public static void deleteExperiment(int experimentNum)
                                throws java.sql.SQLException
```

Deletes the experiment with the specified experiment number from the database.

Parameters:

`experimentNum` - the experiment number to be removed from the database.

Throws:

`java.sql.SQLException`

---

deleteUpload

```
public static void deleteUpload(int uploadID)
                         throws java.sql.SQLException
```

Deletes information with the specified upload ID number from the database.

Parameters:

`uploadID` - the upload ID number to be removed from the database.

Throws:

`java.sql.SQLException`

---

getNextAvailableExperimentNumber

```
public static int getNextAvailableExperimentNumber()
                                        throws java.sql.SQLException
```

Returns the next available experiment number that can be used to store information in the database.

Returns:

the next available experiment number for database storage.

Throws:

`java.sql.SQLException`

---

getNextAvailableUploadNumber

```
public static int getNextAvailableUploadNumber()
                                    throws java.sql.SQLException
```

Returns the next available upload ID number that can be used to store information in the database.

Returns:

the next available upload ID number for database storage.

Throws:

`java.sql.SQLException`

---

getArrayFromFile

```
public static java.lang.String[] getArrayFromFile(java.lang.String filePath,
                                                  java.lang.String delimiter,
                                                  java.lang.String lineFormat,
                                                  int location,
                                                  int start,
                                                  int end,
                                                  int skip)
```

Returns a string array from a file given the position or the row or column specified in the parameters. For example, getArrayFromFile('results.txt', '\t', 'column', 3, 1, 100, 0) would retrieve a 100 member array from the tab delimited file 'results.txt' consisting of the contents of column 3, from row 1 to 100 with no rows skipped.

Parameters:

`filePath` - the path of the file from which the information will be retrieved
`delimiter` - the file delimiter that separates columns in the file
`lineFormat` - the format of the line to be retrieved ('row' or 'column')
`location` - the row or column number containing the information
`start` - the start location of the information
end - the end location of the information
`skip` - the number of rows or columns to skip between items retrieved.

Returns:

the number of elements the list would contain

---

getArrayFromLine

```
public static java.lang.String[] getArrayFromLine(java.lang.String line,
                                                  java.lang.String delimiter,
                                                  int start,
                                                  int end,
                                                  int skip)
```

Takes an input string, splits it using the passed delimiter and returns a subset of the resulting elements as a string array. The split array's boundaries are determined by the specified start, end, and skip parameters.

Parameters:

`line` - the input string to be split and sampled to generate the returned array

`delimiter` - the file delimiter that separates elements in the passed string 'line'

`start` - the starting index of the split array to be incorporated in the output array

`end` - the ending index of the split array to be incorporated in the output array

`skip` - the number of element to skip between items incorporated in the output array.

Returns:

a string array that is a subset of the elements of the split string

---

getItemFromLine

```
public static java.lang.String getItemFromLine(java.lang.String line,
                                               java.lang.String delimiter,
                                               int index)
```

Takes an input string, splits it using the passed delimiter and returns a single element of the resulting array at the specified index. If the index exceeds the number of elements of the array after splitting, an empty string ("") is returned. This method does not use the String.split() method, as that is somewhat wasteful of memory when parsing large files.

Parameters:

`line` - the input string to be split and sampled to generate the returned array

`delimiter` - the file delimiter that separates elements in the passed string 'line'

`index` - the index of the element to be returned after splitting the input string

Returns:

the string at the specified index after splitting the input string using the passed delimiter

---

LoadUploadTuple

```
public static void LoadUploadTuple(UploadInfo Upload,
                                   int uploadID,
                                   java.lang.String type)
                            throws java.sql.SQLException
```

Loads information about the current upload into the Upload table of the database. The loading of this information into the Upload table is performed immediately, rather than through sqlldr with the rest of the upload, such that the upload ID number of this upload will not overlap an upload that comes soon after.

Parameters:

`Upload` - the UploadInfo object that contains the information about the current upload.

`uploadID` - the upload ID number for the current upload.

`type` - the content of the upload ('data', 'annotations', 'statistics')

Throws:

`java.sql.SQLException`

LoadUploadTuple

```
public static void LoadUploadTuple(int uploadID,
                                   int experimentNumber,
                                   java.lang.String type,
                                   java.lang.String userName,
                                   java.lang.String labName,
                                   java.lang.String delimiter,
                                   java.lang.String accessType,
                                   java.lang.String description)
                            throws java.sql.SQLException
```

Loads information about the current upload into the Upload table of the database. The loading of this information into the Upload table is performed immediately, rather than through sqlldr with the rest of the upload, such that the upload ID number of this upload will not overlap an upload that comes soon after.

Parameters:
`uploadID` - the upload ID number for the current upload.
`experimentNumber` - the experiment number for the current upload.
`type` - the content of the upload ('data', 'annotations', 'statistics')
`userName` - the web user's name (first and last)
`labName` - the name of the web user's lab
`delimiter` - the file delimiter that was used in the upload file
`accessType` - the degree of access the user has specified for the upload ('pub', or 'prv')
`description` - a brief description of the upload provided by the user
Throws:
`java.sql.SQLException`

UploadStatisticsFile

```
public static void UploadStatisticsFile(java.lang.String statFilePath,
                                        java.lang.String tempFilePath,
                                        java.lang.String type,
                                        java.lang.String description,
                                        java.lang.String userName,
                                        java.lang.String labName,
                                        int experimentNumber,
                                        int dataTypeNumber,
                                        int linesToSkip,
                                        int ordPosCol,
                                        int resultsCol)
```

Uploads the contents of a statistics file to the database as a derived annotation. Statistics files are the result of analyses performed on data in the database, and this method allows for the direct upload of analytical results into the database without having to go through the upload pages. The statistics file must have a column that specifies ordinal position (row number of data type) and one column containing the statistics themselves.
Parameters:
`statFilePath` - the full path of the statistics results file
`tempFilePath` - the full path of the temporary sqlldr file
`type` - the content of the upload ('data', 'annotations', 'statistics')
`description` - a brief description of the upload provided by the user
`userName` - the web user's name (first and last)
`labName` - the name of the web user's lab
`experimentNumber` - the experiment number for the current upload.
`dataTypeNumber` - the data type number of the data type from which the statistics were derived
`linesToSkip` - the number of lines in the statistics file to discard prior to extracting information to be stored
`ordPosCol` - the column number in the file that contains the ordinal position (zero based)
`resultsCol` - the column number in the file that contains the statistical results (zero based)

---

printUploadFileHeader

```
public static void printUploadFileHeader(java.io.FileWriter DBfile)
```

Prints a header for sqlldr that specifies how information in upload files is loaded into the database tables. After the header has been printed out, the information itself is included in the file and the sqlldr application is run on that file to upload the information.
Parameters:
`DBfile` - the FileWriter object to which the header information will be printed.

---

edu.ucsf.Magellan
Class UploadItem

```
java.lang.Object
  └─ edu.ucsf.Magellan.UploadItem
```

```
public class UploadItem
extends java.lang.Object
```

[illegible]

The UploadItem class stores information about a single item to be uploaded, such as a single data type, a single set of identifiers or annotations, etc.

Field Summary	
private boolean	**annotLocatedWithIdentifiers**
private java.util.Vector	**associatedUploadItems**
private boolean	**dataUnderSampleNames**
private int	**identifierEnd**
private java.lang.String	**identifierFormat**
private int	**identifierLocation**
private boolean	**identifiersLocatedWithData**
private int	**identifierStart**
private java.lang.String	**identifierType**
private int	**infoEnd**
private java.lang.String	**infoFormat**
private int	**infoLocation**
private int	**infoStart**
private java.lang.String	**name**
private int	**sampleEnd**
private java.lang.String	**sampleFormat**
private int	**sampleSkip**
private int	**sampleStart**

Constructor Summary
UploadItem()

Method Summary	
void	**addAssociatedUploadItem**(UploadItem upld) Adds an UploadItem object to the list of objects associated with the current UploadItem.
boolean	**dataUnderSampleNames**() Returns true if the data for each sample in the data file is directly underneath the sample name.
UploadItem	**getAssociatedUploadItem**(java.lang.String name) Returns the UploadItem object with the specified name from the list of objects associated with the

[illegible]

	current UploadItem.
java.util.Vector	**getAssociatedUploadItems**() Returns the list of UploadItem objects associated with the current UploadItem.
int	**getDataPointsPerSample**() Returns the number of variables per sample that have been specified for the item to upload.
int	**getIdentifierEnd**() Returns the row or column number where the identifiers of an item to upload end
java.lang.String	**getIdentifierFormat**() Returns the format of the identifiers associated with the current item to be uploaded ('row' or 'column').
int	**getIdentifierLocation**() Returns the row or column number where the identifiers of an item to upload are located
int	**getIdentifierStart**() Returns the row or column number where the identifiers of an item to upload start
java.lang.String	**getIdentifierType**() Returns the type of the identifiers associated with the current item to be uploaded.
int	**getInfoEnd**() Returns the ending row or column number of the variables of an item to be uploaded
java.lang.String	**getInfoFormat**() Returns the format of the information of the current item to be uploaded ('row' or 'column').
int	**getInfoLocation**() Returns the row or column number where information is located
int	**getInfoStart**() Returns the starting row or column number of the variables of an item to be uploaded
java.lang.String	**getName**() Returns the name of the item to be uploaded.
int	**getSampleEnd**() Returns the row or column number where the samples of an item to be uploaded end
java.lang.String	**getSampleFormat**() Returns the sample format of the item to be uploaded ('row' or 'column').
int	**getSampleSkip**() Returns the number of rows or columns to skip between samples
int	**getSampleStart**() Returns the row or column number where the samples of an item to be uploaded start
boolean	**identifiersLocatedWithData**() Returns true if the identifiers of a data type start and stop in the same row/column as the data type.
boolean	**identifiersSpecified**() Returns true if identifiers have been specified for the current item to upload
boolean	**inAssociatedUploadList**(java.lang.String name) Returns true if there is an UploadItem object with the specified name associated with the current UploadItem.
boolean	**infoLocatedWithIdentifiers**() Returns true if the identifiers of an annotation set start and stop in the same row/column as the annotation set.
void	**removeAssociatedUploadItem**(int position) Removes an UploadItem object from the list of objects associated with the current UploadItem, at the specified position.
void	**setAnnotationsLocatedWithIdentifiers**(boolean annotLocatedWithIdentifiers) Sets a flag indicating whether the identifiers of an annotation set start and stop in the same row/column as the annotation set
void	**setDataUnderSampleNames**(boolean dataUnderSampleNames) Sets a flag indicating whether the data for each sample in the data file is directly underneath the sample name.
void	**setIdentifierEnd**(int identifierEnd) Sets the row or column number where the identifiers of an item to upload end

[illegible]

void	**setIdentifierFormat**(java.lang.String identifierFormat) Sets the format of the identifiers associated with the current item to be uploaded ('row' or 'column').
void	**setIdentifierLocation**(int identifierLocation) Sets the row or column number where the identifiers of an item to upload are located
void	**setIdentifiersLocatedWithData**(boolean identifiersLocatedWithData) Sets a flag indicating whether the identifiers of a data type start and stop in the same row/column as the data type
void	**setIdentifierStart**(int identifierStart) Sets the row or column number where the identifiers of an item to upload start
void	**setIdentifierType**(java.lang.String identifierType) Sets the type of the identifiers associated with the current item to be uploaded.
void	**setInfoEnd**(int infoEnd) Sets the ending row or column number of the variables of an item to be uploaded
void	**setInfoFormat**(java.lang.String infoFormat) Sets the format of the information of the current item to be uploaded ('row' or 'column').
void	**setInfoLocation**(int infoLocation) Sets the row or column number where information is located.
void	**setInfoStart**(int infoStart) Sets the starting row or column number of the variables of an item to be uploaded
void	**setName**(java.lang.String name) Sets the name of the item to be uploaded.
void	**setSampleEnd**(int sampleEnd) Sets the row or column number where the samples of an item to be uploaded end
void	**setSampleFormat**(java.lang.String sampleFormat) Sets the sample format of the item to be uploaded ('row' or 'column').
void	**setSampleSkip**(int sampleSkip) Sets the number of rows or columns to skip between samples
void	**setSampleStart**(int sampleStart) Sets the row or column number where the samples of an item to be uploaded start

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

name

```
private java.lang.String name
```

sampleFormat

```
private java.lang.String sampleFormat
```

infoFormat

```
private java.lang.String infoFormat
```

identifierType

```
private java.lang.String identifierType
```

identifierFormat

```
private java.lang.String identifierFormat
```

sampleStart

```
private int sampleStart
```

sampleEnd

```
private int sampleEnd
```

sampleSkip

```
private int sampleSkip
```

infoStart

```
private int infoStart
```

infoEnd

```
private int infoEnd
```

infoLocation

```
private int infoLocation
```

identifierStart

```
private int identifierStart
```

identifierEnd

```
private int identifierEnd
```

identifierLocation

```
private int identifierLocation
```

dataUnderSampleNames

```
private boolean dataUnderSampleNames
```

identifiersLocatedWithData

private boolean **identifiersLocatedWithData**

---

annotLocatedWithIdentifiers

private boolean **annotLocatedWithIdentifiers**

---

associatedUploadItems

private java.util.Vector **associatedUploadItems**

Constructor Detail

UploadItem

public **UploadItem**()

Method Detail

getName

public java.lang.String **getName**()
Returns the name of the item to be uploaded.
Returns:
the name of the item to be uploaded.

---

setName

public void **setName**(java.lang.String name)
Sets the name of the item to be uploaded.
Parameters:
name - the name of the item to be uploaded.

---

getSampleFormat

public java.lang.String **getSampleFormat**()
Returns the sample format of the item to be uploaded ('row' or 'column'). This string indicates whether the samples of an item (usually a data type) are located in rows or in columns.
Returns:
the sample format of the item to be uploaded

---

setSampleFormat

public void **setSampleFormat**(java.lang.String sampleFormat)
Sets the sample format of the item to be uploaded ('row' or 'column').
Parameters:
sampleFormat - the sample format of the item to be uploaded

---

getInfoFormat

public java.lang.String **getInfoFormat**()
Returns the format of the information of the current item to be uploaded ('row' or 'column'). This string indicates whether the information of an item to be uploaded (such as annotations or identifiers) is located in rows or in columns.
Returns:
the format of the information to be uploaded.

---

setInfoFormat

public void **setInfoFormat**(java.lang.String infoFormat)

Sets the format of the information of the current item to be uploaded ('row' or 'column').

Parameters:

`infoFormat` - the format of the information to be uploaded.

---

getIdentifierType

public java.lang.String **getIdentifierType**()

Returns the type of the identifiers associated with the current item to be uploaded.

Returns:

the type of the identifiers

---

setIdentifierType

public void **setIdentifierType**(java.lang.String identifierType)

Sets the type of the identifiers associated with the current item to be uploaded.

Parameters:

`identifierType` - the type of the identifiers

---

getIdentifierFormat

public java.lang.String **getIdentifierFormat**()

Returns the format of the identifiers associated with the current item to be uploaded ('row' or 'column').

Returns:

the format of the identifiers

---

setIdentifierFormat

public void **setIdentifierFormat**(java.lang.String identifierFormat)

Sets the format of the identifiers associated with the current item to be uploaded ('row' or 'column').

Parameters:

`identifierFormat` - the format of the identifiers

---

getSampleStart

public int **getSampleStart**()

Returns the row or column number where the samples of an item to be uploaded start

Returns:

the row or column number where the samples start

---

setSampleStart

public void **setSampleStart**(int sampleStart)

Sets the row or column number where the samples of an item to be uploaded start

Parameters:

`sampleStart` - the row or column number where the samples start

---

getSampleEnd

public int **getSampleEnd**()

Returns the row or column number where the samples of an item to be uploaded end

Returns:

the row or column number where the samples end

setSampleEnd

`public void setSampleEnd(int sampleEnd)`

Sets the row or column number where the samples of an item to be uploaded end
Parameters:
`sampleEnd` - the row or column number where the samples end

---

getSampleSkip

`public int getSampleSkip()`

Returns the number of rows or columns to skip between samples
Returns:
the number of rows or columns to skip between samples

---

setSampleSkip

`public void setSampleSkip(int sampleSkip)`

Sets the number of rows or columns to skip between samples
Parameters:
`sampleSkip` - the number of rows or columns to skip between samples

---

getInfoStart

`public int getInfoStart()`

Returns the starting row or column number of the variables of an item to be uploaded
Returns:
the row or column number where the information starts

---

setInfoStart

`public void setInfoStart(int infoStart)`

Sets the starting row or column number of the variables of an item to be uploaded
Parameters:
`infoStart` - the row or column number where the information starts

---

getInfoEnd

`public int getInfoEnd()`

Returns the ending row or column number of the variables of an item to be uploaded
Returns:
the row or column number where the information ends

---

setInfoEnd

`public void setInfoEnd(int infoEnd)`

Sets the ending row or column number of the variables of an item to be uploaded
Parameters:
`infoEnd` - the row or column number where the information ends

---

getInfoLocation

`public int getInfoLocation()`

Returns the row or column number where information is located
Returns:
the row or column number where the information ends

---

setInfoLocation

`public void` **`setInfoLocation`**`(int infoLocation)`
Sets the row or column number where information is located.
Parameters:
`infoLocation` - the row or column number where information is located

---

getIdentifierStart

`public int` **`getIdentifierStart`**`()`
Returns the row or column number where the identifiers of an item to upload start
Returns:
the row or column number where identifiers start

---

setIdentifierStart

`public void` **`setIdentifierStart`**`(int identifierStart)`
Sets the row or column number where the identifiers of an item to upload start
Parameters:
`identifierStart` - the row or column number where identifiers start

---

getIdentifierEnd

`public int` **`getIdentifierEnd`**`()`
Returns the row or column number where the identifiers of an item to upload end
Returns:
the row or column number where identifiers end

---

setIdentifierEnd

`public void` **`setIdentifierEnd`**`(int identifierEnd)`
Sets the row or column number where the identifiers of an item to upload end
Parameters:
`identifierEnd` - the row or column number where identifiers end

---

getIdentifierLocation

`public int` **`getIdentifierLocation`**`()`
Returns the row or column number where the identifiers of an item to upload are located
Returns:
the row or column number where the identifiers are located

---

setIdentifierLocation

`public void` **`setIdentifierLocation`**`(int identifierLocation)`
Sets the row or column number where the identifiers of an item to upload are located
Parameters:
`identifierLocation` - the row or column number where identifiers are located

---

dataUnderSampleNames

`public boolean` **`dataUnderSampleNames`**`()`

Returns true if the data for each sample in the data file is directly underneath the sample name.
Returns:
true if the data is located under sample names.

---

setDataUnderSampleNames

`public void` **`setDataUnderSampleNames`**`(boolean dataUnderSampleNames)`

Sets a flag indicating whether the data for each sample in the data file is directly underneath the sample name.
Parameters:
`dataUnderSampleNames` - true if the data is located under sample names.

---

identifiersLocatedWithData

`public boolean` **`identifiersLocatedWithData`**`()`

Returns true if the identifiers of a data type start and stop in the same row/column as the data type.
Returns:
true if the identifiers start/stop in the same row/column as the data type

---

setIdentifiersLocatedWithData

`public void` **`setIdentifiersLocatedWithData`**`(boolean identifiersLocatedWithData)`

Sets a flag indicating whether the identifiers of a data type start and stop in the same row/column as the data type
Parameters:
`identifiersLocatedWithData` - true if the identifiers start/stop in the same row/column as their associated data type

---

infoLocatedWithIdentifiers

`public boolean` **`infoLocatedWithIdentifiers`**`()`

Returns true if the identifiers of an annotation set start and stop in the same row/column as the annotation set.
Returns:
true if the identifiers start/stop in the same row/column as the annotation set

---

setAnnotationsLocatedWithIdentifiers

`public void` **`setAnnotationsLocatedWithIdentifiers`**`(boolean annotLocatedWithIdentifiers)`

Sets a flag indicating whether the identifiers of an annotation set start and stop in the same row/column as the annotation set
Parameters:
`annotLocatedWithIdentifiers` - true if the identifiers start/stop in the same row/column as their associated annotation set

---

identifiersSpecified

`public boolean` **`identifiersSpecified`**`()`

Returns true if identifiers have been specified for the current item to upload
Returns:
true if identifiers have been specified for the current item to upload

---

getDataPointsPerSample

`public int` **`getDataPointsPerSample`**`()`

Returns the number of variables per sample that have been specified for the item to upload. This calculation is based upon the starting and ending position of the variables of the upload item.
Returns:

[illegible]

true if identifiers have been specified for the current item to upload

---

addAssociatedUploadItem

`public void` **`addAssociatedUploadItem`**`(UploadItem upld)`

Adds an UploadItem object to the list of objects associated with the current UploadItem. For example, a data type may have UploadItems associated with it corresponding to annotation sets.
Parameters:
`upld` - the UploadItem to associate with the current item to upload.

---

removeAssociatedUploadItem

`public void` **`removeAssociatedUploadItem`**`(int position)`

Removes an UploadItem object from the list of objects associated with the current UploadItem, at the specified position.
Parameters:
`position` - the position in the list that will be removed.

---

getAssociatedUploadItems

`public java.util.Vector` **`getAssociatedUploadItems`**`()`

Returns the list of UploadItem objects associated with the current UploadItem. This vector may contain annotations associated with a data type, for example.
Returns:
a vector of UploadItem objects associated with the current UploadItem.

---

inAssociatedUploadList

`public boolean` **`inAssociatedUploadList`**`(java.lang.String name)`

Returns true if there is an UploadItem object with the specified name associated with the current UploadItem.
Parameters:
`name` - the name to be checked against the list of associated UploadItem objects.

---

getAssociatedUploadItem

`public UploadItem` **`getAssociatedUploadItem`**`(java.lang.String name)`

Returns the UploadItem object with the specified name from the list of objects associated with the current UploadItem. If no such UploadItem is found, null is returned.
Parameters:
`name` - the name to be checked against the list of associated UploadItem objects.

---

edu.ucsf.Magellan
Class UserBean

```
java.lang.Object
  └─ edu.ucsf.Magellan.UserBean
```

All Implemented Interfaces:
java.io.Serializable, java.util.EventListener, javax.servlet.http.HttpSessionBindingListener

---

```
public class UserBean
extends java.lang.Object
implements java.io.Serializable, javax.servlet.http.HttpSessionBindingListener
```

The UserBean class conforms to the JavaBean standard, and stores as properties the information about the current user (login, password, name, lab, etc). In addition, the UserBean class contains methods that return information as to which data and annotation sets are available to the current user.

See Also:
Serialized Form

Field Summary	
private boolean	**administrator**
private java.util.Vector	**filesToDelete**
private java.lang.String	**firstName**
private java.lang.String	**labName**
private java.lang.String	**lastName**
private java.lang.String	**login**
private java.lang.String	**password**

Constructor Summary
UserBean() Constructor

Method Summary	
void	**addFileToBeDeleted**(java.lang.String filePath) Adds a file to the list of files to be deleted once a user's session has expired.
java.lang.String	**getAvailableAnnotationSelectList**(java.lang.String listName) Returns an HTML select list containing those annotation sets that are available for the current user to view or analyze.
java.lang.String	**getAvailableExperimentSelectList**(java.lang.String listName) Returns an HTML select list containing those experimental data sets that are available for the current user to view or analyze.
java.lang.String	**getFirstName**() Returns the first name of the current user
java.lang.String	**getLabName**() Returns the lab name of the current user
java.lang.String	**getLastName**() Returns the last name of the current user
java.lang.String	**getLogin**() Returns the login for the current user
java.lang.String	**getPassword**() Returns the password for the current user
boolean	**isAdministrator**() Returns the administrator status of the current user.
boolean	**sessionExpired**() Returns true if the current user's web session has expired.
void	**setAdministratorStatus**(boolean status) Sets the administrator status of the current user.
void	**setFirstName**(java.lang.String firstName) Sets the first name of the current user
void	**setLabName**(java.lang.String labName) Sets the lab name of the current user
void	**setLastName**(java.lang.String lastName)

	Sets the last name of the current user
void	**setLogin**(java.lang.String login) Sets the login for the current user
void	**setPassword**(java.lang.String password) Sets the password for the current user
void	**valueBound**(javax.servlet.http.HttpSessionBindingEvent event) Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface
void	**valueUnbound**(javax.servlet.http.HttpSessionBindingEvent event) Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

login

```
private java.lang.String login
```

password

```
private java.lang.String password
```

firstName

```
private java.lang.String firstName
```

lastName

```
private java.lang.String lastName
```

labName

```
private java.lang.String labName
```

filesToDelete

```
private java.util.Vector filesToDelete
```

administrator

```
private boolean administrator
```

Constructor Detail

UserBean

```
public UserBean()
```

Constructor

Method Detail

setLogin

`public void` **`setLogin`**`(java.lang.String login)`

Sets the login for the current user

Parameters:

`login` - the login for the current user

---

getLogin

`public java.lang.String` **`getLogin`**`()`

Returns the login for the current user

Returns:

the login for the current user

---

setPassword

`public void` **`setPassword`**`(java.lang.String password)`

Sets the password for the current user

Parameters:

`password` - the password for the current user

---

getPassword

`public java.lang.String` **`getPassword`**`()`

Returns the password for the current user

Returns:

the password for the current user

---

setFirstName

`public void` **`setFirstName`**`(java.lang.String firstName)`

Sets the first name of the current user

Parameters:

`firstName` - the first name of the current user

---

getFirstName

`public java.lang.String` **`getFirstName`**`()`

Returns the first name of the current user

Returns:

the first name of the current user

---

setLastName

`public void` **`setLastName`**`(java.lang.String lastName)`

Sets the last name of the current user

Parameters:

`lastName` - the last name of the current user

---

getLastName

`public java.lang.String` **`getLastName`**`()`

Returns the last name of the current user
Returns:
the last name of the current user

---

setLabName

```
public void setLabName(java.lang.String labName)
```

Sets the lab name of the current user
Parameters:
`labName` - the last name of the current user

---

getLabName

```
public java.lang.String getLabName()
```

Returns the lab name of the current user
Returns:
the lab name of the current user

---

setAdministratorStatus

```
public void setAdministratorStatus(boolean status)
```

Sets the administrator status of the current user. If this flag is set to true, then the current user will have access to all data and annotation sets in the database.
Parameters:
`status` - the administrator status to be set for the current user

---

isAdministrator

```
public boolean isAdministrator()
```

Returns the administrator status of the current user. If this flag is set to true, then the current user will have access to all data and annotation sets in the database.
Returns:
the administrator status of the current user

---

addFileToBeDeleted

```
public void addFileToBeDeleted(java.lang.String filePath)
```

Adds a file to the list of files to be deleted once a user's session has expired.
Parameters:
`filePath` - the full path of the file to be deleted.

---

sessionExpired

```
public boolean sessionExpired()
```

Returns true if the current user's web session has expired.
Returns:
true if the current user's web session has expired.

---

getAvailableExperimentSelectList

```
public java.lang.String getAvailableExperimentSelectList(java.lang.String listName)
                                                  throws java.sql.SQLException
```

Returns an HTML select list containing those experimental data sets that are available for the current user to view or analyze.
Parameters:

`listName` - the name to be used for the HTML select list.
Returns:
an HTML select list of available data sets
Throws:
`java.sql.SQLException`

---

getAvailableAnnotationSelectList

```
public java.lang.String getAvailableAnnotationSelectList(java.lang.String listName)
                                                  throws java.sql.SQLException
```

Returns an HTML select list containing those annotation sets that are available for the current user to view or analyze.
Parameters:
`listName` - the name to be used for the HTML select list.
Returns:
an HTML select list of available annotation sets
Throws:
`java.sql.SQLException`

---

valueBound

```
public void valueBound(javax.servlet.http.HttpSessionBindingEvent event)
```

Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface
Specified by:
valueBound in interface `javax.servlet.http.HttpSessionBindingListener`
Parameters:
`event` - the HttpSessionBindingEvent passed by the expired session to the method

---

valueUnbound

```
public void valueUnbound(javax.servlet.http.HttpSessionBindingEvent event)
```

Implements the valueBound method of javax.servlet.http.HttpSessionBindingListener interface
Specified by:
valueUnbound in interface `javax.servlet.http.HttpSessionBindingListener`
Parameters:
`event` - the HttpSessionBindingEvent passed by the expired session to the method

---

edu.ucsf.Magellan
Class UtilityFunctions

```
java.lang.Object
  └─ edu.ucsf.Magellan.UtilityFunctions
```

```
public class UtilityFunctions
extends java.lang.Object
```

The UtilityFunctions class provides a series of static methods for use in Magellan. These methods include operations on vectors and integer arrays, as well as tests for proper user input (valid integers, real numbers, etc).

Constructor Summary
`UtilityFunctions()`

Method Summary	
`static java.lang.String`	`arrayJoin(int[] intArray, java.lang.String sep)`

	Takes an array of integers and joins them together, separated by the string sep.
static java.lang.String	**arrayJoin**(java.lang.String[] strArray, java.lang.String sep) Takes an array of string objects and joins them together, separated by the string sep.
static java.lang.String	**arrayJoinSqlEscape**(java.lang.String[] strArray, java.lang.String sep) Takes an array of string objects and joins them together, separated by the string sep and escaping single quotes with double quotes for sql statements.
static void	**copyFile**(java.io.File src, java.io.File dst) Copies the src file to the dst file, for any fileType.
static java.lang.String	**getNewFileName**(java.lang.String filePath)
static int[]	**HashSetToIntArray**(java.util.HashSet set) Converts a HashSet containing integers to a sorted integer array.
static int[]	**IntegerArray**(int start, int end, int skip) Returns an array of integers that starts at start, end at end, and has skip numbers skipped between elements of the list.
static boolean	**isPercentile**(java.lang.String num) Returns true if the passed number contains a valid percentile value (integer between 0 and 100 inclusive).
static boolean	**isPositiveInteger**(int num) Returns true if the passed number contains a positive integer
static boolean	**isPositiveInteger**(java.lang.String num) Returns true if the passed string contains a positive integer
static boolean	**isPositiveRealNumber**(java.lang.String num) Returns true if the passed string contains a valid positive real number.
static boolean	**isRealNumber**(java.lang.String num) Returns true if the passed string contains a valid real number.
static int	**numberElementsInList**(int start, int end, int skip) Determines how many elements a list of integers will have if it starts at start, end at end, and has skip numbers skipped between elements of the list Example: (1,10,2) = list of numbers from 1 to 10 with 2 numbers skipped = (1,4,7,10) There are four elements in the list, so 4 is returned
static java.lang.String	**oppositeLineFormat**(java.lang.String lineFormat) Returns "column" if "row" is passed, otherwise returns "row".
static java.lang.String	**printElementInHtmlTable**(java.lang.String item, boolean wrapFlag) Prints an item in an HTML table as the printElementInHtmlTable function below, but uses a default font size of 1.
static java.lang.String	**printElementInHtmlTable**(java.lang.String item, int fontSize, boolean wrapFlag) Returns an element in HTML table format with flanking tags, and with the font size and NOWRAP properties determined by the passed paramters.
static java.lang.String	**spaces**(int numSpaces) Returns the specified number of concatenated " " strings (for HTML pages).
static int[]	**stringArrayToInt**(java.lang.String[] stringArray) Takes an array of strings and converts them to an array of int's, which is returned.
static java.lang.String	**vectorJoin**(java.util.Vector v, java.lang.String sep) Takes a vector of string objects and joins them together, separated by the string sep.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

UtilityFunctions

```
public UtilityFunctions()
```

Method Detail

vectorJoin

```
public static java.lang.String vectorJoin(java.util.Vector v,
                                          java.lang.String sep)
```

Takes a vector of string objects and joins them together, separated by the string sep.

Parameters:

v - the vector of string objects to be joined

sep - the delimiter that is put between each string in the vector

Returns:

a string containing the joined vector components

---

arrayJoin

```
public static java.lang.String arrayJoin(java.lang.String[] strArray,
                                         java.lang.String sep)
```

Takes an array of string objects and joins them together, separated by the string sep.

Parameters:

strArray - the array of string objects to be joined

sep - the delimiter that is put between each string in the array

Returns:

a string containing the joined array components

---

arrayJoinSqlEscape

```
public static java.lang.String arrayJoinSqlEscape(java.lang.String[] strArray,
                                                  java.lang.String sep)
```

Takes an array of string objects and joins them together, separated by the string sep and escaping single quotes with double quotes for sql statements.

Parameters:

strArray - the array of string objects to be joined

sep - the delimiter that is put between each string in the array

Returns:

a string containing the joined array components

---

arrayJoin

```
public static java.lang.String arrayJoin(int[] intArray,
                                         java.lang.String sep)
```

Takes an array of integers and joins them together, separated by the string sep.

Parameters:

intArray - the array of integers to be joined

sep - the delimiter that is put between each integer in the array

Returns:

a string containing the joined array components

---

isPositiveInteger

```
public static boolean isPositiveInteger(java.lang.String num)
```

Returns true if the passed string contains a positive integer

Returns:

true if the passed string contains a positive integer

---

isPositiveInteger

```
public static boolean isPositiveInteger(int num)
```
Returns true if the passed number contains a positive integer
Returns:
true if the passed number contains a positive integer

---

isPercentile

```
public static boolean isPercentile(java.lang.String num)
```
Returns true if the passed number contains a valid percentile value (integer between 0 and 100 inclusive).
Returns:
true if the passed number contains a valid percentile

---

isRealNumber

```
public static boolean isRealNumber(java.lang.String num)
```
Returns true if the passed string contains a valid real number.
Returns:
true if the passed string contains a valid real number.

---

isPositiveRealNumber

```
public static boolean isPositiveRealNumber(java.lang.String num)
```
Returns true if the passed string contains a valid positive real number.
Returns:
true if the passed string contains a valid positive real number.

---

stringArrayToInt

```
public static int[] stringArrayToInt(java.lang.String[] stringArray)
```
Takes an array of strings and converts them to an array of int's, which is returned.
Returns:
an array of integers

---

spaces

```
public static java.lang.String spaces(int numSpaces)
```
Returns the specified number of concatenated " " strings (for HTML pages).
Returns:
the specified number of concatenated " " strings

---

printElementInHtmlTable

```
public static java.lang.String printElementInHtmlTable(java.lang.String item,
                                                        boolean wrapFlag)
```
Prints an item in an HTML table as the printElementInHtmlTable function below, but uses a default font size of 1.
Returns:
an item in an HTML table with a font size of 1.

---

printElementInHtmlTable

```
public static java.lang.String printElementInHtmlTable(java.lang.String item,
                                                        int fontSize,
                                                        boolean wrapFlag)
```
Returns an element in HTML table format with flanking tags, and with the font size and NOWRAP properties determined by the passed paramters. Examples: printElementInHtmlTable("foo", 2, true) returns "foo", printElementInHtmlTable("bar", 3, false) returns "bar"

Parameters:
`item` - the item to be printed in the cell of the HTML table
`fontSize` - the font size to be used to print the item
`wrapFlag` - if false, include the NOWRAP property in the cell
Returns:
an element in HTML table format

---

oppositeLineFormat

```
public static java.lang.String oppositeLineFormat(java.lang.String lineFormat)
```

Returns "column" if "row" is passed, otherwise returns "row".
Returns:
"column" if "row" is passed, otherwise returns "row".

IntegerArray

```
public static int[] IntegerArray(int start,
                                 int end,
                                 int skip)
```

Returns an array of integers that starts at start, end at end, and has skip numbers skipped between elements of the list.
Example: IntegerArray(1,10,2) = list of numbers from 1 to 10 with 2 numbers skipped = (1,4,7,10)
Parameters:
`start` - the starting point of the list
`end` - the ending point of the list
`skip` - the number of integers to skip between elements of the list
Returns:
an integer array with the passed properties

---

numberElementsInList

```
public static int numberElementsInList(int start,
                                       int end,
                                       int skip)
```

Determines how many elements a list of integers will have if it starts at start, end at end, and has skip numbers skipped between elements of the list Example: (1,10,2) = list of numbers from 1 to 10 with 2 numbers skipped = (1,4,7,10) There are four elements in the list, so 4 is returned
Parameters:
`start` - the starting point of the list
`end` - the ending point of the list
`skip` - the number of integers to skip between elements of the list
Returns:
the number of elements the list would contain

HashSetToIntArray

```
public static int[] HashSetToIntArray(java.util.HashSet set)
```

Converts a HashSet containing integers to a sorted integer array.
Parameters:
`set` - a HashSet containing integers
Returns:
a sorted int array containing the HashSet contents

---

copyFile

```
public static void copyFile(java.io.File src,
                            java.io.File dst)
                     throws java.io.IOException
```

Copies the src file to the dst file, for any fileType. If the dst file does not exist, it is created.
Parameters:

`src` - the source file
`dst` - the destination file
Throws:
`java.io.IOException`

---

getNewFileName

```
public static java.lang.String getNewFileName(java.lang.String filePath)
```

1644R