Technical University of Denmark

Long Wang (s080894)

A Graphics Library for System Analysis

Master's Thesis, February 2012

LONG WANG (S080894)

A Graphics Library for System Analysis

Master's Thesis, February 2012

Supervisor:
Christian W. Probst (probst@imm.dtu.dk)

DTU - Technical University of Denmark, Kgs. Lyngby - 2012

Table of Contents

Abstract

This project aims to establish a software library(API) that enables users to analyse, visualize and manipulate graph or network that is common within the domain of system analysis. The library will extend a graph library, and make the relevant functionality available, plus add new, domain-specific functionality.

To this end an evaluation of different libraries will be performed, and several problem domains explored, for example, communication protocols and process calculi. Based on these explorations, a library will be designed and implemented, and evaluated on different case studies.

Chapter 1
Introduction

There are a plethora of graph libraries available in a great number of programming languages, yet there are not any specific one that comes handy for application in the field of system analysis. Some of them are too complicated to use that have unrelated complexity, some are specialized in certain areas (mathematics or graph algorithm, etc) that do not fit well. Visualising the results of system analysis also requires specialized support in a graph library.

The abundance in library gives us great resource to reference when building our own library, however it also made the choice of choosing one as a foundation for our project hard. To select the right one, we carefully examined our requirement and tries to align it with each of the libraries, besides that, we also performed a evaluation of some of the existing libraries, and analyzed the data to get a full view of libraries available, before made the final decision of choosing the JUNG library.

This project aims to establish such a software library(API) that enables users to analyze, visualize and manipulate graph or network that is common within the domain of system analysis. The library is based on the JUNG(Java Universal Network/Graph) system. It is written in the Java programming language, allowing this library to take advantage of the extensive functionalities provided by the JUNG system, as well as the many Java libraries.

The library has a clear structure with various interfaces, abstract classes and implementing classes. It is designed to be used directly as well as to be extended and evolved in the future. The library also includes documentation of the set of class definitions and the behaviors associated with those classes. Concretely, for example, the library provides different interfaces that represents the ability of transferring data between vertexes, and also has concert classes that implement these functionalities so that they are ready for use. On the graphical side, we implemented a basic layout algorithm for simple graphs in the domain of system analysis, which other libraries does not provide.

This report is structured in four parts:

Background. Background information related to this report is listed and discussed here in the hope that it can improve the understanding of the overall issue.

Benchmark. Comparison of existing graph libraries is carried out on different criteria, including availability, reliability and performance, etc. Thus a benchmark for the sake of comparing performance is carried out. The result and analysis is discussed in this chapter.

Design and Implementation. With the results we get from the previous chapter, we continue with the design and implementation of the library. Starting with general principles as well as detail design, also provides the thought behind some decisions.

Case Study. We provide two non-trivial cases demo for the application of the library, we examine the requirements and build the case step by step, in hopes that it can help grasping the uses of this library.

Chapter 2
Background

In this section, we will discuss the background information that may not be obvious, but will be mentioned and applied later in this thesis. It contains a general discussion about graph/network representation. Besides that, we also gives a brief instruction to the JUNG Library which will be the main graph library on which we build our application (though the detailed evaluation and benchmark will be discussed in next chapter).

2.1 Graph and its Representation

The intuitive notion of a graph is a drawing of (possibly labeled) nodes with edges connecting some of them. A more formal data structure definition is like : A graph (denoted *G*) is a pair of sets *(V,E)* , where *V* is the set of vertices (or nodes) and *E* is the set of edges. Vertices are usually the elements of the problem we are interested in. Each edge connects two vertices, so it can be represented as a pair *E(v1,v2)*.[1]

2.1.1 Graph Representation

There are two common ways to represent graphs in memory: as an adjacency list or as an adjacency matrix. Each of them has its own advantages and disadvantages. First, let us investigate their definitions.

According to Cormen Thomas H. (2001):

- **Adjacency list** - Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows to store additional data on the vertices.
- **Adjacency matrix** - A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.

And now the question are which one is better? There basically are two factors that we need to consider: the time complexity and the space trade-offs. The following table 2.1 gives the time complexity cost of performing various operations on graphs.[2]

If we are to implement the adjacency lists on a 32-bit computer using basic array, an adjacency list for an undirected graph requires around 8*e* bytes of storage, where *e* is the number of edges:

[1] `http://www.cs.toronto.edu/~heap/270F02/node32.html`
[2] `http://en.wikipedia.org/wiki/Graph_(data_structure)`

	Adjacency list	**Adjacency matrix**
Storage	$O(\|V\| + \|E\|)$	O(ǀ V ǀ 2)
Add vertex	O(1)	O(ǀ V ǀ 2)
Add edge	O(1)	O(1)
Remove vertex	O(ǀ E ǀ)	O(ǀ V ǀ 2)
Remove edge	O(ǀ E ǀ)	O(1)
Query: are vertices u, v adjacent?	O(ǀ E ǀ)	O(1)
Remarks	When removing edges or vertices, need to find all vertices or edges	Slow to add or remove vertices, because matrix must be resized or copied

Table 2.1: Different representation of graph or networks in memory

each edge will appear in the entries in two adjacency lists and uses four bytes in each, more concretely each edge appears on the adjacency lists of its two endpoints.

On the other hand, because each entry in an adjacency matrix requires only one bit, they can be represented in a very compact way, occupying only $\frac{n^2}{8}$ bytes of contiguous space, where n is the number of vertices.

Noting that a graph can have at most n^2 edges allowing loops, or in our situation $n \times (n-1)$ in directed graph which is called Directed Complete Graph, to simplify things out, we can let $d = \frac{e}{n^2}$ denote the density of the graph. Then, if $8e > \frac{n^2}{8}$, the adjacency list representation occupies more space, which is true when $d > \frac{1}{64}$. Thus a graph must be sparse for an adjacency list representation to be more memory efficient than an adjacency matrix. However, this analysis is valid only when the representation is intended to store the connectivity structure of the graph without any numerical information about its edges.

In our case, considering the requirement, we would assume (quite fairly) that there will be quite a few operations that checks if an edge exists. Also, since its use for visualization, the numerical information on its edges seems less likely. Most importantly is that when sued for small scale application, the space storage does not really matter. However, in the event of the vertexes in a graph goes up, n^2 builds up quickly and the application frequently results in a sparse graph, which cries for the application of adjacent list.[3]

We are building our library on top of JUNG library, and in JUNG most of the current JUNG vertex implementations employ a variant of the adjacency list representation (read Joshua O Madadhain (2005)), which we term an adjacency map representation: each vertex maintains a map from each adjacent vertex to the connecting edge (or connecting edge set, in the case of graphs that permit parallel edges). (Separate maps are maintained, if appropriate, for incoming directed edges, outgoing directed edges, and undirected edges.) This uses slightly more memory than the adjacency list representation, but makes findEdge approximately as fast as the corresponding operation on the 2D array representation. This representation makes JUNG's data structures and

[3] `http://en.wikipedia.org/wiki/Adjacency_list`

algorithms, in general, well-suited for use on large sparse networks.

2.2 JUNG Graph Library

The JUNG library being the foundation of the library we developed undoubtedly needs some elaboration. Here we will present it in two parts:

- The Jung System Overview
- Features of Jung System

2.2.1 The JUNG System

As Joshua O Madadhain (2005) noted, the JUNG (Java Universal Network/Graph) Framework is a free, open-source software library that provides a common and expendable language for the manipulation, analysis,and visualization of data that can be represented as a graph or network. It is written in the Java programming language, allowing JUNG-based applications to make use of the extensive built-in capabilities of the Java Application Programming Interface (API), as well as those of other existing third-party Java libraries. We describe the design, and some details of the implementation, of the JUNG architecture, and provide illustrative examples of its use.

2.2.2 Major features of JUNG

The major features of JUNG includes the following (see Joshua O Madadhain (2005)):

- Support for a variety of representations of entities and their relations, including directed and undirected graphs, multi-modal graphs (graphs which contain more than one type of vertex or edge), graphs with parallel edges (also known as multigraphs), and hypergraphs (which contain hyperedges, each of which may connect any number of vertices). Of course, for our case we used the normal sparse graph with only directed and undirected edges allowed.
- Mechanisms for annotating graphs, entities, and relations with metadata. These capabilities facilitate the creation of analytic tools for complex data sets that can examine the relations between entities, as well as the metadata attached to each entity and relation.
- Implementations of a number of algorithms from graph theory, exploratory data analysis, social network analysis, and machine learning. These include routines for clustering, decomposition, optimization, random graph generation, statistical analysis, and calculation of network distances, ?ows, and ranking measures (centrality, PageRank, HITS, etc.)
- A visualization framework that makes it easy to construct tools for the interactive exploration of network data. Users can choose among the provided layout and rendering algorithms, or use the framework to create their own custom algorithms.
- Filtering mechanisms which extract subsets of a network; this allows users to focus their attention, or their algorithms, on specific portions of a network.

These capabilities make JUNG a good foundation for data representation and manipulation in graphs. Especially considering the visualization framework it provides suit our requirement(detailed in later chapter). And as Joshua O Madadhain (2005) put it: "JUNG is a framework on which applications and tools for manipulating graph and network data can be built. It can be used in simple snippets of code to test ideas, or to aid in the development of a sophisticated tool with a graphic user interface. JUNG is not itself a standalone tool, but rather a library that can be used to support the construction of specialized tools."

2.3 Java Language Related Issues

Since the project is built on Java libraries, and developed in Java Programming Language, there are quite some topics that we need to address. It is though that the prevalence of Java made discussions about the Java Language background seems less indispensable, yet we would like to keep this part and focus more on three less discussed area of Java. Namely JVM, Java generics and Annotation SuppressWarnings.

2.3.1 JVM client and server mode

The knowledge of JVM and its settings are important in this project, not only because the project and the libraries that it reference is in Java programming language, but that a benchmark of these existing libraries is needed, we cannot test without fine tuning the JVM. Thus we shall say few words about this topic here.

The JDK includes two flavors of the VM - a client-side offering, and a VM tuned for server applications. These two solutions share the Java HotSpot runtime environment code base, but use different compilers that are suited to the distinctly unique performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults.[4]

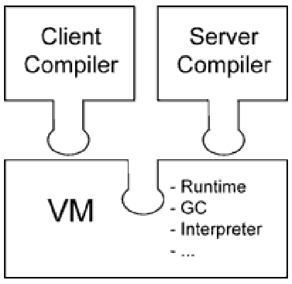


Figure 2.1: The Java HotSpot Client VM, on the left, and the Java HotSpot Server VM, on the right, use a different compiler but otherwise interface to the same virtual machine, using the same garbage collection (GC) routine, interpreter, thread and lock subsystems, and so on.

Although the Server and the Client VMs are similar, the Server VM has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications,

[4]`http://java.sun.com/products/hotspot/whitepaper.html`

which need the fastest possible operating speed more than a fast start-up time or smaller runtime memory footprint.

The Client VM compiler does not try to execute many of the more complex optimizations performed by the compiler in the Server VM, but in exchange, it requires less time to analyze and compile a piece of code. This means the Client VM can start up faster and requires a smaller memory footprint.[5]

The Server VM contains an advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers, as well as some optimizations that cannot be done by traditional compilers, such as aggressive inlining across virtual method invocations. This is a competitive and performance advantage over static compilers. Adaptive optimization technology is very flexible in its approach, and typically outperforms even advanced static analysis and compilation techniques.See footnote 4.

2.3.2 Java Generics

Java generics, like C++ templates, allows the abstraction of types. And most commonly used in containers. As pointed out in Java's own document *Java Programming Language* "This long-awaited enhancement to the type system allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety to the Collections Framework and eliminates the drudgery of casting"[6].

This construct is very useful as it helps debugging the code and are widely accepted since great number of other languages also has the feature available. with generics, type related mistakes would be caught by the compiler, instead of crashing the application at runtime. One always favor compile time error than runtime error, since compile time error informs one immediately and one can use the compiler error messages to figure out what the problem is and fix it. Runtime error, however, can be much more problematic; they do not always surface immediately, and when they do, it may be at a point in time that is far removed from the actual cause of the problem[7].

But there are drawbacks also. The authors of *Design Patterns* note that this technique, especially when combined with delegation, is very powerful but that "[dynamic], highly parameterized software is harder to understand than more static software" (see Erich Gamma (1994)).

2.3.3 Annotations

An Java annotation is a special form of syntactic metadata that can be added to Java source code.They have no direct effect on the operation of the code they annotate. Yet they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. [8]

Annotations have a number of uses, among them[9]:

[5] `http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf`
[6] `http://docs.oracle.com/javase/1.5.0/docs/guide/language/index.html`
[7] `http://docs.oracle.com/javase/tutorial/java/generics/generics.html`
[8] `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`
[9] `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`

- Information for the compiler: Annotations can be used by the compiler to detect errors or suppress warnings.
- Compiler-time and deployment-time processing: Software tools can process annotation information to generate code, XML files, and so forth.
- Runtime processing: Some annotations are available to be examined at runtime.

```
@SuppressWarnings
```

The annotation mentioned above is one of the three annotation types that are predefined by the language specification itself, as stated in the documentation of *Java 2 Platform Standard Ed. 5.0*:"(the above) Indicates that the named compiler warnings should be suppressed in the annotated element (and in all program elements contained in the annotated element). Note that the set of warnings suppressed in a given element is a superset of the warnings suppressed in all containing elements. For example, if you annotate a class to suppress one warning and annotate a method to suppress another, both warnings will be suppressed in the method."

2.3.4 Exceptions

The design of Exception, or more precisely, the choice of which type of exception to use in our library is a topic worth noting. So here we shall briefly list related background information.

Definitions

According to the definition given by Oracle[10]

```
An exception is an event, which occurs during the execution of a program, that
    disrupts the normal flow of the program's␣instructions.
```

In the detail of Java code, Throwable is at the top off all exceptions. Underneath Throwable there is Error and Exception. Underneath Exception there is the subclass called RuntimeExcdeption. When designing we need to differentiate these two classes. Java has two types of exceptions - checked and unchecked. The difference between checked and unchecked exceptions is rather simple: checked exceptions are subclasses of the Exception class, and an unchecked exception is a subclass of RuntimeException, which is a subclass of Exception.

The role and design considerations are also different. Checked exceptions are enforced by the Java compiler and virtual machine. Client programmers will not be able to compile their code unless they have caught checked exceptions somewhere in the execution stack. Virtual machines will not run unless the appropriate exception classes are available. A checked exception indicates that a called function has failed to fulfill its end of the contract, and since the client programmer is dependent on it for their program's operation, it throws a checked exception so that client programmers are given the opportunity to deal with such an anomaly.[11]

[10] http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html
[11] http://osix.net/modules/article/?id=766

An unchecked exception is a subclass of RuntimeException because it is discovered at runtime, not at compile time. An unchecked exception is thrown because the calling function did not fulfill its end of the contract by passing bad or invalid data.

When applying the above listed rules, our general idea is that checked exceptions are something we may be able to foresee but may be based on input that is out of our control, and that we have to deal with. Similarly,the advice by Joshua Bloch in Effective Java best summaries: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors (read Bloch (2008)) [12]. The case study for applying these rules shall be discussed in detail in the next chapter.

Why do we need Exceptions

We shall not go deep into the topic to discuss the mechanism or how the system handles a exception. But we shall provide some evidence for our application of Exception in the system. In other words, why we are using it.

Notes from Oracle states the following advantages of Exception[13]:

- Separating Error-Handling Code from "Regular" Code: Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
- Propagating Errors Up the Call Stack: A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods.
- Grouping and Differentiating Error Types: Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

Principles of exception handling

Exception handling is simple enough in a hello-world scenario. So simple as if one is encountered, catch it and print the stack trace. Yet in real world this approach is not nearly sufficient. So it would be necessary to have some guidelines and the following are some of the generally accepted principles of exception handling (see Shenoy (2002)):

- If you can’t handle an exception, don’t catch it.
- If you catch an exception, don’t swallow it.
- Catch an exception as close as possible to its source.
- Log an exception where you catch it, unless you plan to rethrow it.
- Structure your methods according to how fine-grained your exception handling must be.
- Use as many typed exceptions as you need, particularly for application exceptions.

[12] Item 58

[13] `http://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html`

2.4 Java Graphics related

The Java 2D API maintains two coordinate spaces (see Oracle (2012):

- User space: The space in which graphics primitives are specified
- Device space: The coordinate system of an output device such as a screen, window, or a printer

User space is a device-independent logical coordinate system, the coordinate space that your program uses. All geometries passed into Java 2D rendering routines are specified in user-space coordinates.

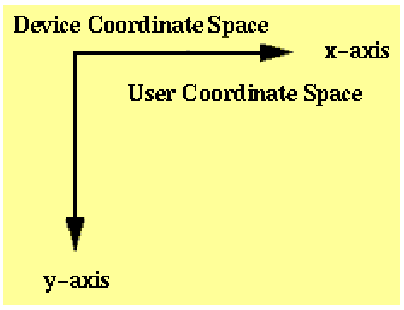


Figure 2.2: User Space and coordinates

When the default transformation from user space to device space is used, the origin of user space is the upper-left corner of the component¡¯s drawing area. The x coordinate increases to the right, and the y coordinate increases downward, as shown in the figure 2.2. The top-left corner of a window is (0,0). All coordinates are specified using integers, which is usually sufficient. However, some cases require floating point or even double precision which are also supported.

2.5 Benchmark

Here listed some of the most basic ideas regarding the benchmark topic.

2.5.1 Definitions

A benchmark in general is a snapshot of some aspect of certain objects under evaluation at a fixed point in time that allows us to measure performance and compare results. In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it[14].

In our project here, it means specifically the measurement of performance. Without benchmarks, it would be hard to make a choice out of the similar libraries, and there is no way to confidently measure and report on the results.

[14]`http://en.wikipedia.org/wiki/Benchmark_(computing)`

2.5.2 Purpose

The purpose of this benchmark is solely to evaluate performance, as it is hard to compare the performance simply by looking at the specifications. And more over, the libraries available usually lack information about their performance. Thus we need to take the action, the process is designed to mimic the type of basic operations we perform when applying the system.

As this benchmark is only for evaluation, it has no consideration over otherwise important features like facility burden, security, reliability, scalability, etc.

2.5.3 How to benchmark

There are some guidance found from the source[15], though it is targeted at different area than this report, I altered it to make it functional also for our purpose. And in the following chapter, the actual benchmark process use this as instruction.

1. Determine what you are going to measure
Take time to identify the quantifiable elements of the library that needs to be and can be evaluate. The more specific your measurements, the more clearly you will be able to assess progress. Benchmarks need to be operational rather than strategic.

2. Work out how to measure it
Think about how you are going to measure the objectives you want to address. Be specific but not too complicated.

3. Take measurements before you start
Measure all the things you want to quantify and record the measurements clearly. These measurements become the benchmarks you will use to compare.

4. Repeat the same measurements
Measure everything in exactly the same way under same system/environment settings as you did before with another library. This will show you the gains you have made from the training.

[15]`http://www.skillshighway.govt.nz/setting-benchmarks_page16.html`

Chapter 3
Analysis and Comparison of Existing Libraries

In this chapter, we will solve the problem of choosing the right graph libraries to use, and present the advantages and disadvantages with that particular choice. Concretely, this chapter contains the following parts:

- Graph Libraries: a general discussion and comparison of existing graph libraries, more focused on what each library is designed for
- System Settings: environment settings such as Java HotSpot virtual machine (JVM) in Sun's J2SE 5.0 release so that the environment is tuned to a suitable state.
- Benchmark: the process and results

3.1 Graph Libraries

There are a large number of graph libraries that are implemented in Java language. Yet some of them belongs to the commercial area whereas others are developed under some free software license. The ones that have been considered are listed in table 3.1, with a line of brief introduction from its own site:

With the listed information, it becomes easy to rule out the commercial ones like EasyCharts and ElegantJ. For the rest, there are some libraries that belongs to the group of charting libraries, among them the main functionalities become to create beautifully presented histograms, fan charts, line charts, etc. And that, is not what we are looking for. And then we can eliminate the JFreeChart and left with the following 5:

- G
- JGraphT
- JGraphX
- JUNG
- Prefuse

Name	Brief Intro.	Site	Free?
EasyCharts	EasyCharts is a 100% java based chart library that enables you to add great-looking charts in your java applications, web pages, and server based web applications with very little coding effort.	`http://www.objectplanet.com/easycharts/`	No
ElegantJ Charts	ElegantJ Charts is a complete Java component model. It supports the standard component architecture features of properties, events, methods, and persistence.	`http://www.elegantjcharts.com/`	No
G	G is a generic graphics library built on top of Java 2D in order to make scene graph oriented 2D graphics available to client applications in a high level, easy to use way.	`http://geosoft.no/graphics/`	Yes
JFreeChart	JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications.	`http://www.jfree.org/jfreechart/`	Yes
JGraphT	JGraphT is a free Java graph library that provides mathematical graph-theory objects and algorithms.	`http://jgrapht.sourceforge.net/`	Yes
JGraphX	JGraphX enables you to produce Java Swing applications that feature interactive diagramming functionality	`http://www.jgraph.com/doc/mxgraph/index_javavis.html`	Yes
JUNG	JUNG, the Java Universal Network/Graph Framework, is a software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network.	`http://jung.sourceforge.net/`	Yes
Prefuse	Prefuse is a set of software tools for creating rich interactive data visualizations.	`http://prefuse.org/`	Yes

Table 3.1: Graph Libraries

As mentioned, our intention for the library is more focused on display and manipulation, rather than usage of the graph theory/algorithm. So according to the introduction given by their corresponding websites, JGraphT focused more on "the mathematical graph-theory objects and algorithms".

Now let us look at *Prefuse*, it is more focused on data modeling and visualization :"supports a rich set of features for data modeling, visualization, and interaction." And for *G*, though the profile fits our expectation, it is quite a lightweight library(80kB, self contained, simple to use) and lacks extensive documentation, thus made it relatively hard to extend.

The two left libraries Jung and JGraphX have both met our basic requirements, they also possess extensive documentations and demos, thus we decided to benchmarking them by choosing the one that has a better performance.

3.2 System Settings

In this section, the detailed settings are listed and discussed. Configuring the system settings are essential, the purpose is not only to tune the system for a better performance (and certainly not for an unrealistic result), but also to keep the environment consistent thorough out the whole evaluation process and also for later replicate.

3.2.1 JVM Options

In order to evaluate the performance of the two different graph libraries, a series of benchmark of basic actions on a graph is carried out. But before we dive into the practice, we need to configure the testing environment. Since both libraries to be tested runs on JVM, the setting and contributing factors are discussed in this section.

In the J2SE 5.0 release, default values for the garbage collector, heap size, and HotSpot virtual machine (client or server) are automatically chosen based on the platform and operating system on which the application is running.

On machines that are not server-class machines, the default values for JVM, garbage collector, and heap sizes are listed:

- the client JVM
- the serial garbage collector
- Initial heap size of 4MB
- Maximum heap size of 64MB

The default options for client(default) mode here are in Table 3.2:

JVM option	Default	Description
-Xms	4MB initial	java heap size
-Xmx	64MB maximum	java heap size
-Xmn	Platform¨Cdependen	Default initial size of the new (young) generation, in bytes

Table 3.2: heap memory options

Another point worth considering is Java's garbage collection mechanism. The heap size of virtual machine determines the cost of collecting garbage on two measurements: time and frequency. These measurements desired value is application dependent, and should be determined by analyzing the actual garbage collection time and frequency. In the event that the heap size is large, full garbage collection will be very slow, but the frequency will be reduced. If the heap size and memory requirements are consistent, complete collection will run quickly, but will be more frequent. The purpose of tuning the heap size is to minimize the garbage collection time, to maximize the handling of customer requests. In the benchmark test in our case, in order to ensure the best performance, we should set a large heap size, in the hope that garbage collection will not be performed throughout the benchmark process, or at least does not have a big influence on this evaluation process.

Processor	Intel Core 2 U9400 @ 1.40GHz
Memory	2 GB (DDR3)
Hard disk	128G SSD
Chipset	Intel ICH9 Family SMBus Controller - 2930
Operating System	Windows 7 32bit Professional SP1

Table 3.3: System specifications

3.2.2 Test System

It does not gives much information if we lack the specifications about the system. System specifications are listed in Table 3.3.

Note: The Java object heap has to be allocated in contiguous virtual addresses, for implementation reasons. In 32 bit operating systems only 2gb memory can be addressed as contiguous [1], which happens to be the memory size of my laptop.

JVM Version

JVM Version for this benchmark is:

```
C:\Users\Logan>java -version
java version "1.6.0_29"
Java(TM) SE Runtime Environment (build 1.6.0_29-b11)
Java HotSpot(TM) Client VM (build 20.4-b02, mixed mode)
```

JVM Set Up

Since Java Virtual Machine is set to default3.2.1, and adding one million vertex would result in **MemoryOutofUse** Exception. Thus, when running the test, VM argument **-Xms128m -Xmx1024m** is passed to VM.

3.3 Benchmark

In order to find the more suitable library as mentioned in section 3.1, we need to perform a basic benchmark on these two candidates. We evaluated two aspects of the library as a representation for their corresponding performance. Specifically, each of them will perform add vertex and add edge action and we compare their efficiency doing so. The initial scale is set to add one million vertex consecutively, and add one million edges in the each two vertex that were added after one another. To make the estimation more accurate, both benchmark process were performed for 10 times and compared against the average value.

More formally, our starting point is a random experiment with a sample space and a probability measure P. In the basic statistical model, we have an observable random variable X taking values in a set S. Here we sample *n = 10* objects from a population and record measurements of running time taken

$$X = (X_1, X_2, ..., X_10) \tag{3.1}$$

[1] http://weblogs.java.net/blog/aiqa/archive/2005/04/jvm_memory_usag.html

where X_i is the vector of measurements for the i'th object. Thus we have sample space $(X_1, X_2, ..., X_{10})$ size of 10 and each of them is independent and identically distributed. The arithmetic mean $\overline{X}$ is defined via the equation

$$\overline{X} := \frac{1}{n} \sum_{i=1}^{10} X_i \tag{3.2}$$

And we are using the arithmetic mean in view of the fact that it holds several properties, the most crucial to us is:

- The mean value $\overline{X}$ we get is a real-valued function of the random sample and thus is a statistic. Like any statistic, the sample mean is itself a random variable with a distribution, mean, and variance of its own. Here the distribution mean is unknown and the sample mean is used as an estimator of the distribution mean. And it is a good estimator, it is unbiased.
- The strong law of large numbers[2] states that the sample mean $\overline{X}$ converges to the distribution mean μ with probability 1:

$$\overline{X} \to \mu \text{ for } n \to \infty \tag{3.3}$$

The above stated properties ensure us that the arithmetic mean is a good estimator of the running time and thus made our benchmark effective in addressing the purpose.

And since JUNG library provides many more ways of representing a graph, there are more tests for JUNG library on different graph types.

The overall result in graph is shown below in figure 3.1 and figure 3.2. We We can easily tell that JUNG library has a better performance under our test criteria.

3.3.1 JUNG Library Results

We performed two evaluation on JUNG library. We chose three different types of graphs of the JUNG library: SparseGraph, DirectedSparseGraph, and UndirectedSparseGraph to evaluate.

Add vertex benchmark

One million vertexes add into different types of graph, namely three: SparseGraph, DirectedSparseGraph, UndirectedSparseGraph.

```
SparseGraph
** Benchmarking add vertex. **
Running time: 4239 ms.
Running time: 3341 ms.
Running time: 3124 ms.
Running time: 4920 ms.
Running time: 26317 ms.(way off, excluded)
Running time: 55221 ms.(way off, excluded)
Running time: 5557 ms.
Running time: 1757 ms.
Running time: 2148 ms.
Running time: 3566 ms.
```

[2] http://en.wikipedia.org/wiki/Law_of_large_numbers

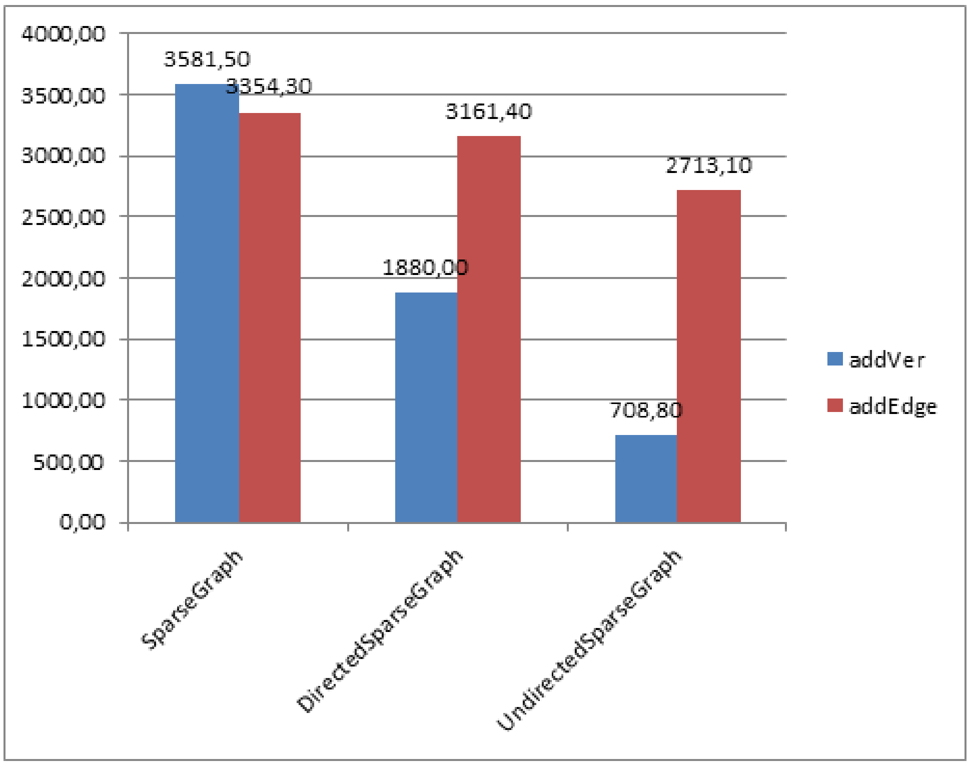


Figure 3.1: Result for add edge. Add one million edge into the graph. Time in milliseconds.

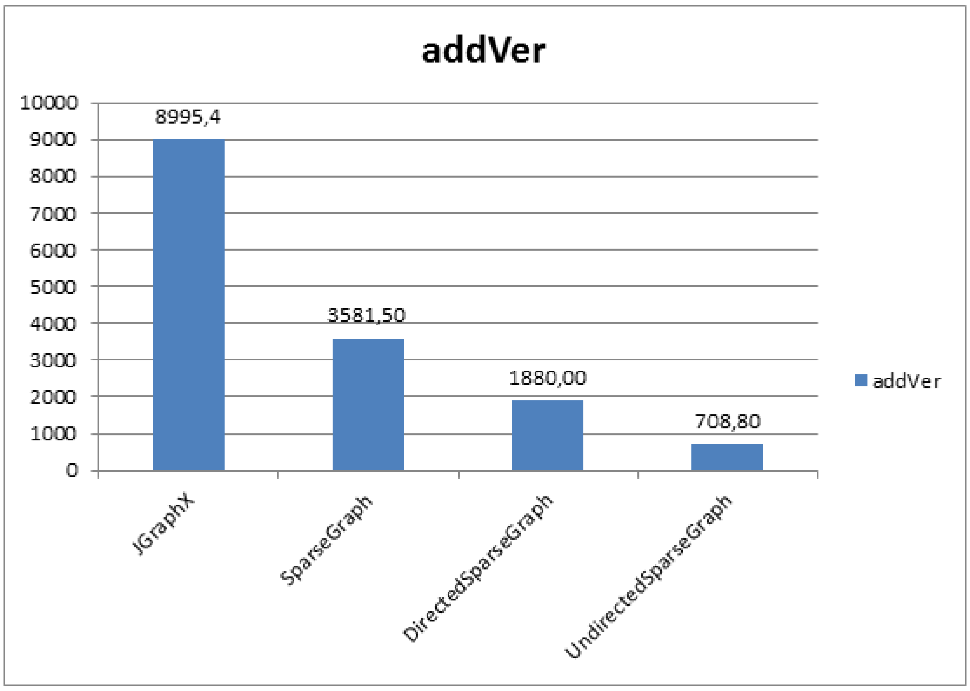


Figure 3.2: Result for add vertex. Add one million vertex into the graph. Time in milliseconds.

```
===Average time: 11019.0 ms.===

DirectedSparseGraph
** Benchmarking add vertex. **
Running time: 1882 ms.
Running time: 1312 ms.
Running time: 1770 ms.
Running time: 1886 ms.
Running time: 1911 ms.
Running time: 1990 ms.
Running time: 1279 ms.
Running time: 3162 ms.
Running time: 1946 ms.
Running time: 1662 ms.
===Average time: 1880.0 ms.===

UndirectedSparseGraph
** Benchmarking add vertex. **
Running time: 1352 ms.
Running time: 424 ms.
Running time: 714 ms.
Running time: 976 ms.
Running time: 402 ms.
Running time: 546 ms.
Running time: 904 ms.
Running time: 407 ms.
Running time: 514 ms.
Running time: 849 ms.
===Average time: 708.8 ms.===
```

In the first part of benchmark, where we add vertex to SparseGraph, there occurred two values that are around ten times more than other measured values. These two results are inconsistent, hence excluded from the final result and are not likely to be random error.

Random errors[3] are errors in measurement that lead to measurable values being inconsistent when repeated measures of a constant attribute or quantity are taken. The word random indicates that they are inherently unpredictable, and have null expected value, namely, they are scattered about the true value, and tend to have null arithmetic mean when a measurement is repeated several times with the same instrument. All measurements are prone to random error.

These two results are caused by unpredictable fluctuations, these fluctuations may be in part due to interference of the environment with the measurement process. Or perhaps resulted from the Java virtual machine with the garbage collection mechanism mentioned earlier. But it is worth a mention that we have kept the testing system strictly consistent during the testing process.

[3] `http://en.wikipedia.org/wiki/Random_error`

Add edge benchmark

One million vertexes added into different kinds of graph, namely three: SparseGraph, DirectedSparseGraph, UndirectedSparseGraph. And then edges are created following the rule of linking the adjacent numbered vertexes. So it is quite a sparse graph indeed.

```
SparseGraph
** Benchmarking add edge. **
Running time: 3261 ms.
Running time: 3011 ms.
Running time: 3216 ms.
Running time: 5508 ms.
Running time: 3480 ms.
Running time: 1990 ms.
Running time: 3770 ms.
Running time: 3494 ms.
Running time: 2050 ms.
Running time: 3763 ms.
===Average time: 3354.3 ms.===

DirectedSparseGraph
** Benchmarking add edge. **
Running time: 3486 ms.
Running time: 1710 ms.
Running time: 1662 ms.
Running time: 3375 ms.
Running time: 3688 ms.
Running time: 3942 ms.
Running time: 4170 ms.
Running time: 4547 ms.
Running time: 1648 ms.
Running time: 3386 ms.
===Average time: 3161.4 ms.===

UndirectedSparseGraph
** Benchmarking add edge. **
Running time: 1618 ms.
Running time: 3070 ms.
Running time: 3266 ms.
Running time: 1610 ms.
Running time: 3157 ms.
Running time: 1668 ms.
Running time: 3216 ms.
Running time: 2775 ms.
Running time: 3062 ms.
Running time: 3689 ms.
===Average time: 2713.1 ms.===
```

3.3.2 jGraphX Library Results

Similar to above benchmark with JUNG library,

Add vertex to graph.

It is very inconvenient if one just wants to add vertex to graph of JGraphX library. One has to specifically point out the coordinates of the point to be added. Admittedly, this helps to make graph layout easier, but not always come in handy in the requirement of this project. And it is natural that the performance of simply adding one million vertexes is very much below JUNG's.

The code line and results:

```
//command to add
graph.insertVertex(parent, null, String.valueOf(i), 0, 0, 0, 0);

Running time: 11885 ms.
Running time: 9390 ms.
Running time: 8727 ms.
Running time: 8412 ms.
Running time: 8526 ms.
Running time: 8325 ms.
Running time: 8816 ms.
Running time: 8309 ms.
Running time: 8703 ms.
Running time: 8861 ms.
===Average time: 8995.4 ms.===
```

Add edge

Similar with add vertex, the insert edge function is not efficient, and hard to benchmark. The line looks like this:

```
Object v1 = graph.insertVertex(parent, null, "Hello", 20, 20, 80, 30);
Object v2 = graph.insertVertex(parent, null, "World!", 240, 150, 80, 30);
graph.insertEdge(parent, null, "Edge", v1, v2);
```

Thus to add one million edges, I have to keep track of all the nodes added in the graph. And when even testing for only 100000 vertex and edge connecting one from the other, it took unbearable time:

```
Running time: 159719 ms.
```

As for the reason of this result, in my mind, is that to hold all the nodes in memory is just not possible and thus the operating system automatically activates its paging memory scheme, which provides virtual memory using the disk storage for data that does not fit into physical random-access memory (RAM). And because the store and retrieve of data from secondary storage for use in main memory is significantly slower, thus the operation becomes really slow.

Chapter 4
Design and Implementation

As we have discussed in the previous chapter about the advantage and disadvantages of each available libraries, and came to the conclusion that overall JUNG library is more suitable to this project and thus are chosen accordingly. Now, we would like to show how each requirement is met in the details of our implementation, and why we are doing it.

Designing any object-oriented software is hard, and even more difficult if it is to be a library like ours that provides satisfiable, reusable functionalities. This chapter consists of the following parts: first we list out the specific design problems we are facing in this project, second section gives overview of the design of our library, the third section discuss the detailed design of some class and functions. We will show how our design capture solutions to design problems in our application.

4.1 API

We are building an library, and it is only through the API that users can access the functionalities of this library. So before we look at the problems and design, it is important to investigate the meaning and properties regarding API (with reference to NetBeans (2012)).

The need for API

API is an abbreviation that stands for *Application Programming Interface*. Interface here takes the general meaning rather than the Java language specific one. It means that there are two different subjects involved at least. One being the provider of various services/functionalities, while the other is the one left for applications to make calls into it. And from another perspective, the producer of the code and other programmers using it are also on two separated sides. They intrinsically differs in many aspects, like their goals, needs, expectations and schedules, etc. This observation is fundamental to understands what is to come.

It is exactly this separation that implies the rules for designing and maintaining an API. Suppose that there was no separation and the whole product was developed by one single tight team and build at once, there would be less need for bothering with API (as it is definitively more work). But as the real world products are composed from a set of independent projects developed by teams that do not necessarily know about each other, have completely different schedules and build their projects independently, but still want to communicate among themselves there is a need for a stable contract that can be used for such communication.

What is API

API is developed to enable communication between teams and applications within separated and distributed development situations. The answer to "what is API" shall include every factor that can influence such kind of development. Thus we should look into these factors before we get started on developing our own.

The API can and should take these factors into consideration (see NetBeans (2012)):

method and field signatures : communication between applications is usually about calling functions and passing data structures between each other. If there is a change in the names of the methods, in their arguments or in structure of exchanged data, the whole program often does not even link well, nor it can run.

files and their content : many applications read various files and their content can influence their behavior. Imagine application relying on the other one to read its configuration file and modifying its content prior to invoking the application. If the format of the file changes or the file is completely ignored, the communication between those applications gets broken.

behavior : a bit harder to grip, but important for the separation as well is the the dynamic behavior. How the program flow looks like - what is the order of execution, what locks are being held during calls, in which threads a call can happen, etc.

The important thing with respect to distributed development is to be aware of possible APIs - of possible things other code can depend on. Only by identifying such aspects of own application one can develop it in a way that will not hurt cooperation with separately developed applications.

4.2 Design Problems

A good design can deal with a bunch of challenges, as Erich Gamma (1994) noted, we need find the pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. The design should be specific to the problem at hand but also general enough to address future problems and requirements.

In our project, we will examine the following problems in our design:

1. Finding the appropriate objects. The "right" objects to include in the library. Our goal is to have the objects represents the problem space at the right granularity. We need to find out not so obvious abstractions of the target system and the objects that can capture them.

2. Structure of our library, inheritance hierarchies, the graphs and specialized vertexes and edges. This affects all aspects of our library design. We are not writing a completely new library, but rather to build upon the existing library of JUNG. With techniques like inheritance and composition, we need to both reuse the functionalities and build them flexible.

3. User operations. Our intention is to build a library for the application in the field of system analysis. These requirement demands us to keep our implementation details transparent to the user.

4. Layout algorithm. JUNG provides various layout algorithms, yet we lack those that are easy to use and understand. We provided the most simple and straightforward one: grid layout.

Since we are designing an API library, we also have the following goals(from Harold (2004) that we tries to meet, though the actual effect is hard to quantify:

- It must be absolutely correct. In the our case, this meant that the API could never produce unexpected result no matter what the caller did.

- It must be easy to use. This is hard to quantify. A good way to get an idea is to write lots of example code. Are there groups of operations that you keep having to repeat? Do you have to keep looking up your own API because you forget what things are called? Are there cases where the API does not do what you might expect?

- It must be easy to learn. This overlaps considerably with ease of use. But there are some obvious principles to make learning easier. The smaller the API, the less there is to learn. Documentation should include examples. Where appropriate, the API should look like familiar APIs.

- It must be fast enough. Make sure the API is simple and correct. Then think about performance. You might be inclined to make API changes because the original API could only be implemented in an inefficient way. By all means change it to allow a more efficient implementation, provided you do not compromise correctness or simplicity. Do not rely on your intuition to know what performs well. Measure. Then tweak the API if you have determined that it really matters.

We will discuss the above listed problems and tries to meet the goals presented above in the sections that follow. Each problem has an associated set of goals plus constraints on how we achieve those goals. We explain these goals and constraints in detail before proposing a specific solution.

4.2.1 Finding the right objects

Object-oriented programs are all about objects, and indeed, the programs are made up of objects. It might be easy and straightforward to identify some objects in a system, yet the hard part is to decomposing a system into objects in a good way. It is difficult because many factors came into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on.Erich Gamma (1994)

The design methodologies of object-oriented design has many established and different approaches. As pointed out in Erich Gamma (1994), one can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations. Or one can put some focus on the responsibility and collaborations within the target system. Or even model the real world using ’role play’ or ’card game’ methods, and translate the objects found during analysis into design. Yet it is hard to tell which are the best approach.

In this project, as noted before, has its own special requirements. We are building on the existing library of JUNG and thus follow its intrinsic way of abstraction. For example, we set our own graph class *MyGraph* to extend the *SparseGraph* of JUNG. As shown in the figure 4.4, subclass relationship are usually indicated with a vertical line and a white triangle at the end. The class at the triangle end is the *parent class*, and the class on the other end is the *subclass*.

```
public class MyGraph<T> extends SparseGraph<Vertex<T>, Edge> implements
    MoveData<T, Edge>, CopyData<T, Edge>
```

4.2.2 Determine object granularity

In general, granularity is the extent to which a system is broken down into small parts, either the system itself or its description or observation. It is the "extent to which a larger entity is subdivided." [1]

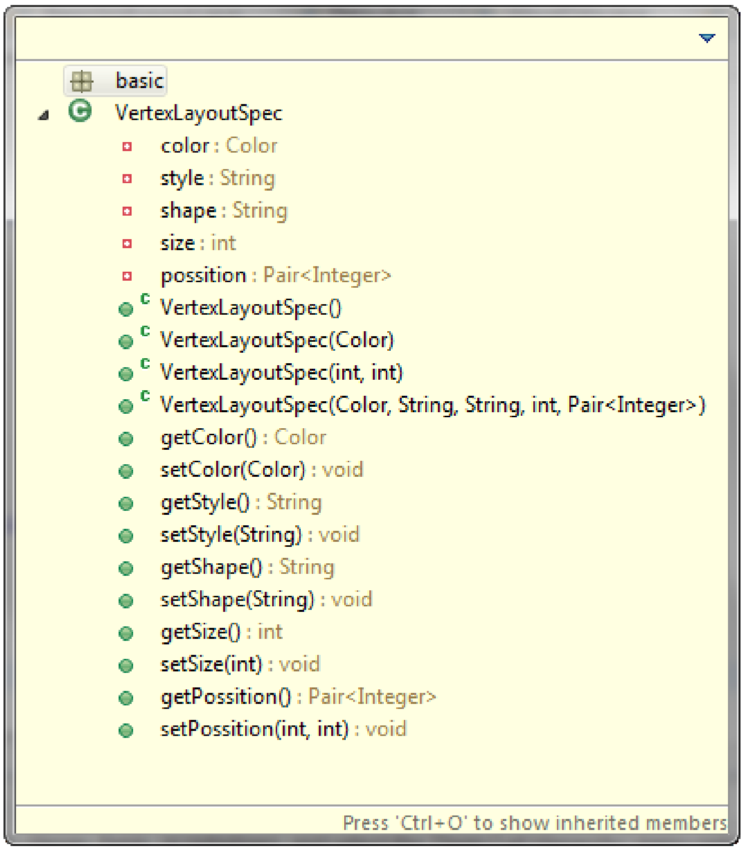
In object-oriented program, objects can vary tremendously in size and number. They can represent everything from a hardware or all the way up to entire applications. In this project for instance, we are required to store information of the layout specifications of a vertex, we could list one by one in the field of vertex or we can abstract out a layout class to encapsulate the variables. We opt for the latter choice, to encapsulate the layout specifications and isolate them from the rest of the program. The encapsulation here keep the parts that stay the same separate from the parts that might change(which is the layout specifications) and thus it becomes easy to make changes.

Figure 4.1 depicts what the class *VertexLayoutSpec* consists of, it naturally holds the private parts of fields about the color, style, shape, position, etc as the layout specifications. There are naturally also constructors, getters and setters that provide access to the data. The encapsulation has the advantage of remove the dependencies on layout information from the vertex itself. It also made it easy to adjust to future modifications of requirement, for example, if we are required to add a new layout specifications for vertex like the absolute position of each vertex in double, we does not need to change the constructors of *Vertex*, but only the implementation detail of the class *VertexLayoutSpec*. These made them low in coupling and higher in cohesion.

4.2.3 Determine method signature

A method signature defines a functions inputs and outputs, includes at least the function name and the number of its arguments. In some programming languages, it may also specify the function's return type, the types of its arguments, or errors it may pass back. The set of all signatures defined by an objet's operations is called the *interface* to the object. By the way, the interface here is different from the Java programming language's keyword *interface*, it means the generic sense of the term.

[1] `http://en.wikipedia.org/wiki/Granularity`



```
basic
VertexLayoutSpec
    color : Color
    style : String
    shape : String
    size : int
    possition : Pair<Integer>
    VertexLayoutSpec()
    VertexLayoutSpec(Color)
    VertexLayoutSpec(int, int)
    VertexLayoutSpec(Color, String, String, int, Pair<Integer>)
    getColor() : Color
    setColor(Color) : void
    getStyle() : String
    setStyle(String) : void
    getShape() : String
    setShape(String) : void
    getSize() : int
    setSize(int) : void
    getPosition() : Pair<Integer>
    setPossition(int, int) : void
```

Figure 4.1: Class detail for *VertexLayoutSpec*

Considering that we are writing a customized API for others to use, it is important to keep in mind one of the principles of object-oriented programming:

Program to an interface, not an implementation.

Following this guideline, our system supports the developer in coding against the interface and we can change the internal workings without impacting the usage of them. And there is no way to know anything about an object or to ask it to do any operations without going through the interface. Knowing this, we have kept only those classes and methods public or protected that are intended to be used by others, other implementation details and private fields is kept away. We provides extensive documentation as well.

Generally speaking, we have two techniques that would enable us to "program to interface", either through an abstract class or a interface.

Abstract class or Interface

This is not intended solely to compare the advantages and disadvantages of abstract class and interface, but to demonstrate the logic behind designing - when to apply abstract class and when to use interface.

It is easy to tell abstract class from concrete class, if some class should never be initialized, never make instance of, then it should be declared as abstract class(or consider interface). So if one base class will never be made instance of, then Abstract class and Interface are the more appropriate choice. As for the decision of abstract class or interface, basically abstract class is a abstract view of any real world entity, whereas interface is more abstract one, it sometimes expresses an ability or relationship.

Another consideration is enforced single inheritance in Java programming language, compared with other programming languages like C++. For example in our case, we want a vertex that can be moved around, it is natural to let the vertex be a subclass of some abstract vertex class and implement a interface that enables it to move between edges. As we do not have the mechanism of multiple inheritance, subclassing to both vertex and moveable object is not possible.

These are the in general idea of taking decision between abstract class, interface and normal class. There are more to it. As there is one constant factor in software that is the change of requirement. If a class is made as *interface* then it is difficult to accommodate changes in this class in future. Because if I add new method in the interface, then it is demanded that in every implementing class the newly added method are implemented.

CAN-DO and IS-A relationship can also help define the difference between Interface and abstract class. As we already discussed Interface can be use for multiple inheritance for example we have another interface named copyData which having behavior copy. IS-A is for "generalization" and "specialization", denote the extension relationship. CAN-DO offers a way to see the implement interface relationship.

Abstract Class

An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer its implementation of some or all of the operations to its subclass. For example, our *AbstractVertex<T>* class implements the interface *implements Comparable<Vertex<T»* , the operation

```
public abstract int compareTo(Vertex<T> v2);
```

is declared *abstract*, since we have no idea as to how each vertex should rank against the other. And thus implementation detail is pushed to its subclass like *Vertex*, where the are ranked by their unique index.

```
        /**
         *
         * compares two vertex according to there index.
         * @param v2 is the vertex to compare with
         * @return 0 if index are the same. 1 if this vertex's index is larger
              than v2's. -1 otherwise.
         */
        @Override
        public int compareTo(Vertex<T> v2) {
                if (this.index == v2.index)
                        return 0;
                else if (this.index > v2.index)
                        return 1;
                else
                        return -1;
        }
```

The Advantages of Abstract Classes[2]

The main reason that there is preference to the application of abstract classes is their ability to evolve in a time - it is possible to add a new method with a default implementation without breaking existing clients or implementors (here we talk about runtime compatibility, not compile time one). Interfaces lack such functionality, so it is necessary to introduce another interface to provide future extensions. So you end up with a lot of interfaces.

A second very useful feature of abstract classes is the possibility of restricting access rights. Every method in a public interface is public and everybody can implement the interface. That for example means anybody can implement such interface, but in real life, one often wants to restrict that and have the creation under control. Interfaces lack such restrictions.

Another thing that is possible with abstract classes is that they can contain static methods. Of course that with interface one can create separate classes with factory methods, but the truth is that a class is usually the most natural and reasonable place for factory methods that return instances.

Interface

What JUNG library provides here is also applicable to our library. So listed here as a reference. JUNG makes use of Java interfaces, abstract classes, and implementation classes in its type definitions. There are a few reasons that JUNG uses combinations of these layers of abstraction.

The Advantages of Interfaces

The most obvious one is that usage of the type, if implemented as an abstract class, is limited as java does not allow multiple inheritance of classes. This only becomes a problem when a type is huge, or when it significantly enhances developer productivity to be able to subclass and reuse a base implementations. We will call these support classes, where one is expected to subclass and reuse a base class's implementation.

The second advantage of interfaces is that there is an enforced separation between the API and the implementation. But this can be achieved with abstract classes too, with a bit of self control, while in interfaces that is enforced by the compiler.

[2]`http://wiki.netbeans.org/API_Design`

Interface design

One unique requirement from the user is the ability to spread information between vertexes. Specifically, there are three different functionalities that we need to provide: to move the data, to copy the data and to pass the data. The name might be a little bit confusing or misleading, so the fact is: move means that the data in the source vertex should be removed if the process is successful in moving the data to the target vertex, and a directed edge should be established in that direction between the to endpoints. On the contrary, copy does not delete the original data. For pass, it is the move function without the need to create a edge between them, but have to make sure beforehand that between the source node and target node there is a pass connecting them. If directed edges are involved, then with direction there is only one way reachable. For undirected edges, they can go both ways. Let us first look at the move functionalities in detail:

- Move a specific data from one vertex(startNode) to another vertex(targetNode), and if successful creates an directed edge from startNode to targetNode.
  Precondition: startNode contains data d, which is the one to be moved.
  Postcondition: startNode does not contain data d, and targetNode contains data d, and there is an edge from startNode to targetNode.

- Move all the data from startNode to the targetNode.
  Precondition: startNode contains some data.
  Postcondition: startNode does not contain any data, and targetNode contains the data from startNode, and an edge from startNode to targetNode.

```
moveData(Vertex<T> startNode, Vertex<T> targetNode, T d, Graph<Vertex<T>,E> g)
     throws Exception;
moveAll(Vertex<T> startNode,Vertex<T> targetNode, Graph<Vertex<T>,E> g) throws
     Exception;
```

Let us wrap the two combined API in one structure moveData. Should the moveData be interface or abstract class? Simple analysis can show that firstly, the concept does not fall into the realm of an entity, much less an object, but it provides a functionality. Moreover the moveData itself contains just two methods that are open. All of the that leads to answer that the type should be an interface. We have the ability for multiple inheritance, and there is no fear of evolving the interface because it has just one method that does it all, no need for static factory methods, no need to prevent subclassing. Thus an interface is the right choice.

Similarly, for copyData and passData we also has the method signatures listed below. Notice that copyData do not require a reference to the original graph since nothing other than the information inside the endpoints are altered.

```
//copyData
copyData(Vertex<T> sourceNode, Vertex<T> targetNode, T d)        throws
    Exception;
```

```
copyAll(Vertex<T> sourceNode, Vertex<T> targetNode)          throws
    Exception;

//passData
passData(Vertex<T> startNode, Vertex<T> targetNode, T d,Graph<Vertex<T>,E> g)
    throws Exception;
passAll(Vertex<T> startNode, Vertex<T> targetNode,Graph<Vertex<T>,E> g) throws
     Exception;
```

The specifications are also similar to moveData with a little difference:

CopyData:

- **copyData:** Copy a specific data from one vertex(startNode) to another vertex(targetNode).
  Precondition: startNode contains data d, which is the one to be copied.
  Postcondition: startNode contains data d, and targetNode contains data d.

- **copyAll:** Copy all the data from startNode to the targetNode.
  Precondition: startNode contains some data
  Postcondition: startNode remains intact, and targetNode contains all the data from startNode.

PassData:

- **passData:** Pass a specific data from one vertex(startNode) to another vertex(targetNode).
  Precondition: startNode contains data d, which is the one to be copied. It must be reachable from startNode to targetNode.
  Postcondition: startNode do not contain data d, and targetNode contains data d.

- **passAll:** Pass all the data from startNode to the targetNode.
  Precondition: startNode contains some data, it must be reachable from startNode to targetN-ode.
  Postcondition: startNode does not contain any data, and targetNode contains all the data from startNode.

The above listed specifications comes intuitively as we analyzed our case, however we now face a problem of whether we should design three separated interfaces with each representing the above mentioned functionalities? Or should we put all six methods into one interface? We need to realize one fact that if a concrete class were to implement that interface, it would be forced by the compiler to implement all the interface methods. So wrapping all methods into one interface would be a bad idea, it will decrease the cohesion of this interface. Thus we have three separated interface: moveData, passData and copyData.

4.2.4 Inheritance and Composition

There are two most common techniques in reusing functionalities in object-oriented system design, they are class inheritance and object composition. Each have their own advantage and disadvantage, we shall give a brief introduction and brief about their application in our library.

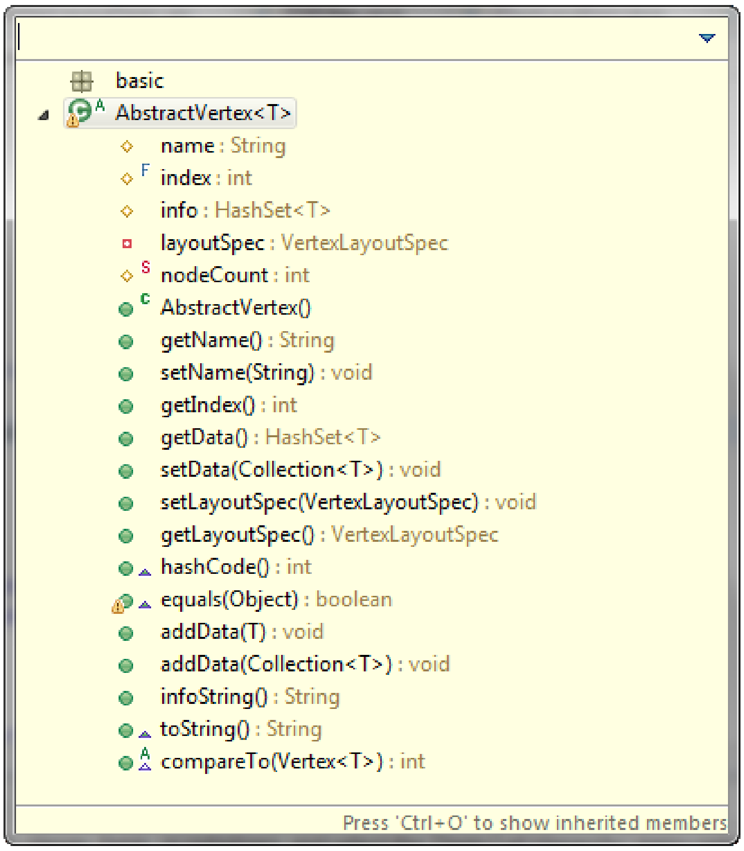
Inheritance

One major feature we are using here is the *class inheritance* of the object-oriented programming. When such a subclass inherits from a parent class, it includes the definition, data and functionalities that the parent class holds. But not necessarily all of them, as in Java and many other object-oriented languages, there are access specifiers that control access to class members. The primary purpose is to separate the interface of a class from its implementation, so as to support the concept of information hiding and encapsulation.

A common set of access specifiers that many object-oriented languages support is[3]:

- private (or class-private) restricts the access to the class itself. Only methods that are part of the same class can access private members.
- protected (or class-protected) allows the class itself and all its subclasses to access the member.
- public means that any code can access the member by its name.

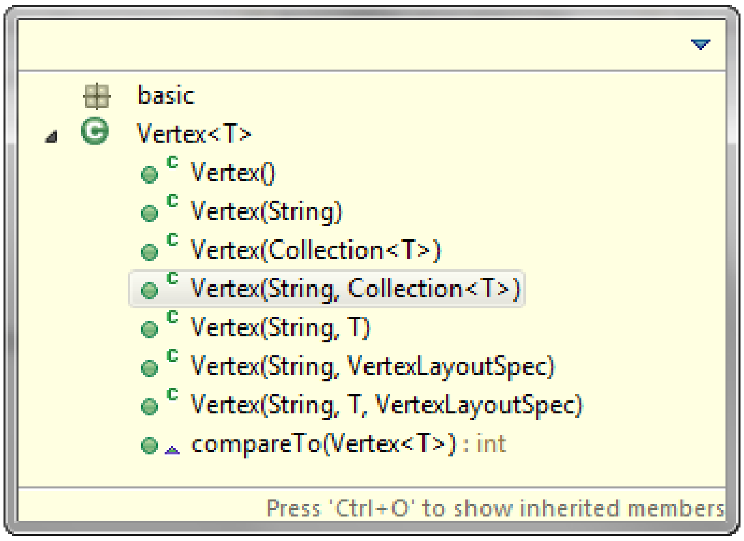
In our project for example, *Vertex* is a subclass of *AbstractVertex*, the former contains the data and operations defined by the latter ones, as shown in figure 4.2 and figure 4.3.[4]



```
basic
AbstractVertex<T>
    name : String
    index : int
    info : HashSet<T>
    layoutSpec : VertexLayoutSpec
    nodeCount : int
    AbstractVertex()
    getName() : String
    setName(String) : void
    getIndex() : int
    getData() : HashSet<T>
    setData(Collection<T>) : void
    setLayoutSpec(VertexLayoutSpec) : void
    getLayoutSpec() : VertexLayoutSpec
    hashCode() : int
    equals(Object) : boolean
    addData(T) : void
    addData(Collection<T>) : void
    infoString() : String
    toString() : String
    compareTo(Vertex<T>) : int
Press 'Ctrl+O' to show inherited members
```

Figure 4.2: Class diagram detail for *AbstractVertex*

[3]`http://en.wikipedia.org/wiki/Class_(computer_programming)`
[4]The figure is generated by Eclipse.



```
basic
Vertex<T>
    Vertex()
    Vertex(String)
    Vertex(Collection<T>)
    Vertex(String, Collection<T>)
    Vertex(String, T)
    Vertex(String, VertexLayoutSpec)
    Vertex(String, T, VertexLayoutSpec)
    compareTo(Vertex<T>) : int
Press 'Ctrl+O' to show inherited members
```

Figure 4.3: Class diagram detail for *Vertex*

The relationships including the interfaces they implemented and the two classes mentioned above are showed in figure 4.4, it displays explicitly that *Vertex* extends *AbstractVertex* whereas both classes implement the interface *Comparable*.

Composition

Classes can be composed of other classes, thereby establishing a compositional relationship between the enclosing class and its embedded classes. Compositional relationship between classes is also commonly known as a has-a relationship. Object composition is an alternative to inheritance. New functionality is obtained by assembling or composing objects to get more complex functionalities. There is another principle for object-oriented designErich Gamma (1994):

Favor object composition over class inheritance.

Compared with inheritance, composition has some advantages over inheritance:

- It helps to keep each class encapsulated and focused on one task. The class and class hierarchy will be less likely to grow into huge unmanageable monsters. Whereas inheritance expose a subclass to details of its parent's implementation, and may force the subclass to change should the parent class has any changes. And since Java is a enforced single inheritance language, where a class can only derive from one base class, composition provides a more flexible way out.
- Object composition is defined dynamically at run time through objects acquirement references to other objects. Whereas object inheritance is defined statically at compile time. And because the reference is solely through their interfaces, an object can be replaced at run time by another as long as it has the same type (or subclasses through polymorphism), thus fewer implementation dependencies.

In UML, composition is depicted as a filled diamond and a solid line. It always implies a multiplicity of 1 or 0..1, as no more than one object at a time can have lifetime responsibility

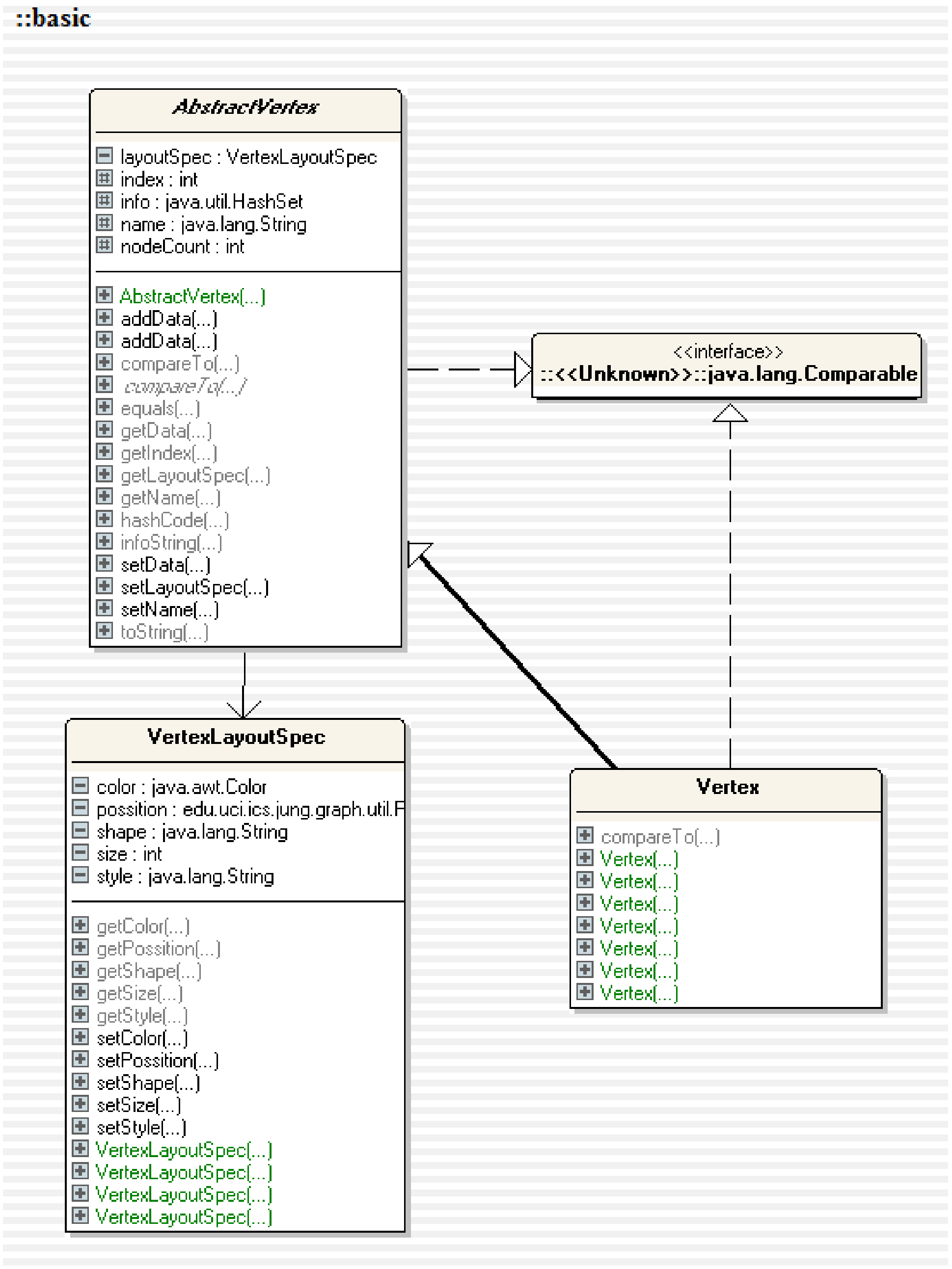


Figure 4.4: Class diagram detail for both *AbstractVertex* and *Vertex*

for another object. (Note that it is different from the more general form, aggregation, which is depicted as an unfilled diamond and a solid line.) In the figure 4.4 as we have shown previously, the composition is depicted as an arrow and a solid line.

Let us take the example of layout class for vertex, where in the *AbstractVertex* class, the VertexLayoutSpec and other primitive types *int* and *String* are combined to form the member of the class. Or in other words, the object *AbstractVertex* contains a reference or pointer to the object VertexLayoutSpec and thus has an arrow from the former one pointing the latter.

4.3 Detailed design

Now that we have the whole picture described, we will discuss concretely how each class is implemented and how each requirement is met.

4.3.1 Choice of data structure

When we are to design *AbstractVertex* and *AbstractEdge*, a choice need to be made on which data structure we should use to store the data. The data structure that first comes to mind is arrays and collections. Obviously, it ought to be a class that implements the *Interface Collection* of java.util. Arrays do not fit here as we do not know how many elements we are expected to hold. Java provides wide variety of implementations and we need to choose carefully.

Our requirements:

- we want the data structure to hold a group of objects
- we do not need to allow duplicate elements
- the order of elements is not important
- we need to be able to look up given element at a low time cost

These requirements calls for a hash-based solution. It satisfies the above listed requirements. The fact that we do not allow duplicate elements even lowered the possibility of collision, thus improve the efficiency of search and retrieving operations. And most importantly, its overall performance are excellent compared with all the linear data structure.(A linear data structure is one in which, while traversing sequentially, we can reach only one element directly from another.) Indeed the performance depends on the choice of hash function, and for the best possible choice of hash function, a table of size n with open addressing has no collisions and holds up to n elements, with a single comparison for successful lookup, and a table of size n with chaining and k keys has the minimum max(0, k-n) collisions and O(1 + k/n) comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(k)$ amortized comparisons per insertion and up to k comparisons for a successful lookup.[5]

We considered these three classes HashTable, HashMap and HashSet as candidate. And we chose HashSet as the one. We shall explain the reasoning behind it.

[5] `http://en.wikipedia.org/wiki/Hash_table`

```
protected HashSet<T> info;
```

HashTable and HashMap is similar, both provides key-value access to data. The major difference lies in the fact that Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones. And, another difference is that HashMap permits null values in it, while Hashtable does not. So for these reasons, HashMap is preferred here.

For the class of Hashset, it implements the Set interface, yet actually run a HashMap instance at the background. It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. It is not synchronized.[6] The major difference lies in the fact that HashMap keeps a (key,value) pair whereas HashSet only a set of non-duplicate elements. So according to the nature of this problem, HashSet fit here. One of HashSet's subclasses is LinkedHashSet, so in the event that we want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This would not be as easy if you were using Hashtable.

4.3.2 Layout Algorithm

The problem comes from one of the demos for the application of the library. We want to have several vertexes that spread in a grid like two dimensional space, where each vertex is given its position. In other words, we need a simple layout algorithm similar to the GridLayout in java.awt package[7]. More specifically, want a algorithm that creates a grid layout with the specified number of rows and columns. All components in the layout are given equal size and have equal gaps between them.

Background

First, we need to get straight where JUNG library place the layout class and how it works in the system.

JUNG provides mechanisms for laying out and rendering graphs. The current renderer implementations use the Java Swing API to display graphs, but they may be implemented using other toolkits (such as SWT).

In general, a visualization requires one of each of the following as noted in Joshua O Madadhain (2005):

- A **Layout**, which takes a graph and determines the location at which each of its vertices will be drawn.

[6] `http://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html`
[7] `http://docs.oracle.com/javase/7/docs/api/java/awt/GridLayout.html`

- A (Swing)**Component**, which provides a "drawing area" upon which the data is rendered. JUNG provides a VisualizationViewerclass for this purpose, which is an extension of the Swing JPanel class. A currently available experimental version of VisualizationViewer allows the user to create a "window" on the graph visualization, which can be used to magnify (zoom in on) portions of the graph, and to select different areas for magnification (panning).
- A **Renderer**, which takes the data provided by the Layout and paints the vertices and edges into the provided Component.

As for the different layout algorithms, the JUNG library also provides a great number of them (see the classes that implementing the interface on `http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/algorithms/layout/Layout.html`). They all implement the interface in the package *edu.uci.ics.jung.algorithms.layout*. Notice that JUNG's documentation[8] noted:" A generalized interface is a mechanism for returning (x,y) coordinates from vertices. In general, most of these methods are used to both control and get information from the layout algorithm."

```
public interface Layout<V,E> extends org.apache.commons.collections15.
    Transformer<V,Point2D>
```

To name some of them, there are the widely used force-based algorithms, like SpringLayout or KKLayout, for drawing graphs in an aesthetically pleasing way. There is also the CircleLayout and TreeLayout for different purposes. But all in all, is to position the nodes of a graph in two-dimensional or three-dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible in the two dimensional space[9].

Solution

However, these fancy layout algorithms do not give us much help, because what we are looking for is a simple grid like layout, like the one shown in figure 5.2. In the figure, the vertexes are placed into a three row by two column grid. In it the vertexes *a1,a2,a3* would be in the first column and *b1,b2,b3* would be in the second column, where as *a1* and *b1* shall be on the first row. Following this fashion, *a2,b2* and *a3,b3* are on the seconde and third row respectively.

This class implements the basic grid layout, according to the index of each vertex in the graph. It extends the StaticLayout, by default fill the static grid row by row according to the index. Can be altered to fill the column first. The class diagram is shown in figure 4.6.

The essence of this class is to take a *Vertex* in and return a *Point2D* object, in which stored the layout position information. This is the method that has been inherited from the parent class, and need to be overridden to achieve the designed functionality. There are two situations, as shown in the flowchart in figure 4.7. If the vertex passed in a the parameter knows where it should be in the panel, then the position is calculated for it and returned wrapped in a *Point2D* object. If the vertex does not hold that information, then the class calculates the layout for all of the vertexes in the graph and store them back into each vertexes.

[8] `http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/algorithms/layout/Layout.html`

[9] `http://en.wikipedia.org/w/index.php?title=Force-based_algorithms_(graph_drawing)&oldid=477528895`

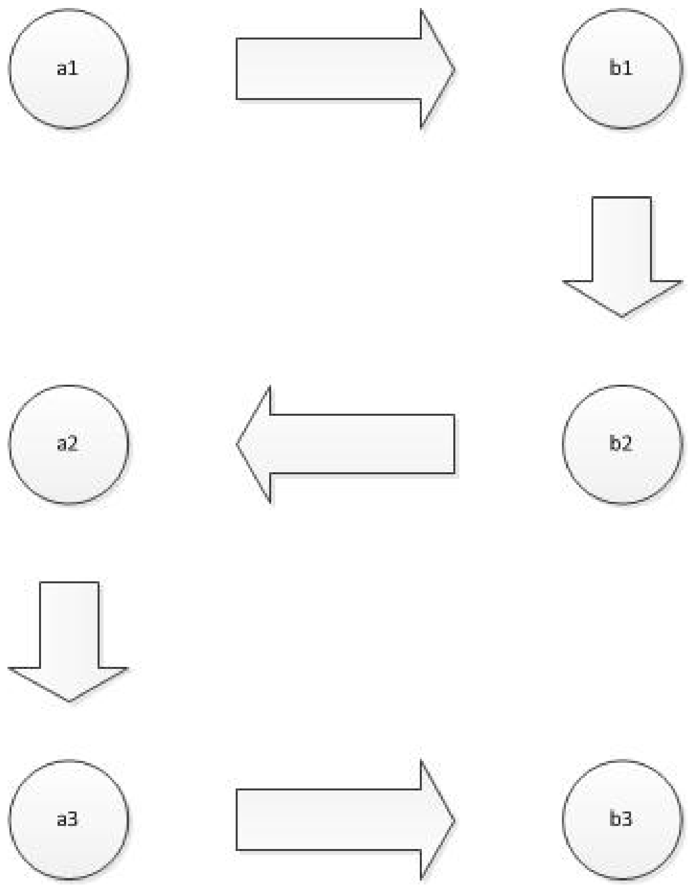
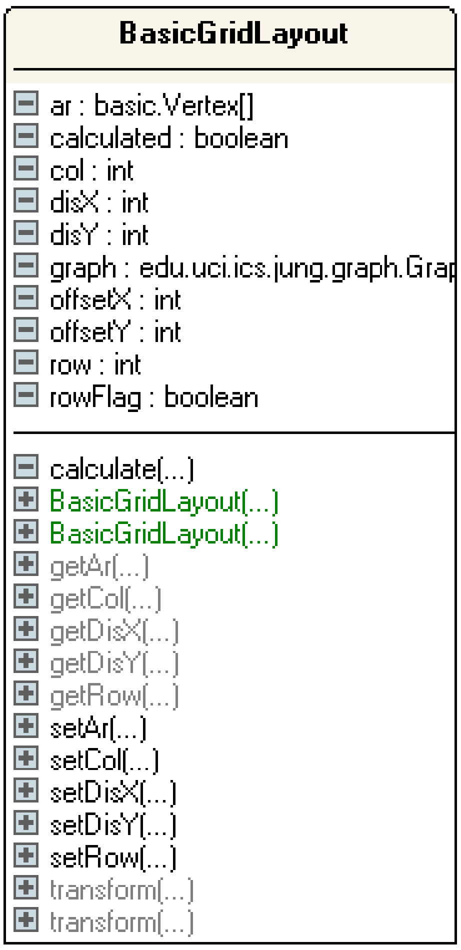


Figure 4.5: Desired simple layout

The code snippet below are the details of calculating the vertex layout position if the vertex stores specific *row*, *column* information in the VertexLayoutSpec class. It is known by examining the non-null of its fields in line 9. However, if it fails to find its layout information, then we call in the function *calculate()* to calculate the positions for each vertex.

```
1    @Override
         public Point2D transform(Vertex v) {
3                // the location is calculated for each vertex
                 if (calculated == false) {
5                        calculate();
                         calculated = true;
7                }
                 // the vertex knows where it will be in the grid
9                if (v.getLayoutSpec()!= null && v.getLayoutSpec().getPossition
                     () != null) {
                         this.setLocation(v, v.getLayoutSpec().getPossition().
                             getSecond()
11                                       * disX + offsetX, v.getLayoutSpec().
                                             getPossition()
                                         .getFirst()
13                                       * disY + offsetY);
```



```
BasicGridLayout
ar : basic.Vertex[]
calculated : boolean
col : int
disX : int
disY : int
graph : edu.uci.ics.jung.graph.Grap
offsetX : int
offsetY : int
row : int
rowFlag : boolean
calculate(...)
BasicGridLayout(...)
BasicGridLayout(...)
getAr(...)
getCol(...)
getDisX(...)
getDisY(...)
getRow(...)
setAr(...)
setCol(...)
setDisX(...)
setDisY(...)
setRow(...)
transform(...)
transform(...)
```

Figure 4.6: Class diagram for BasicGridLayout

```
                }
                return new Point2D.Double(this.getX(v), this.getY(v));
        }
```

The code snippet below shows the calculation for each vertex. It is only excused for once, this is a static process. We assume that each vertex that has already been in the graph should be similar in their states concerning their layout information, in other words they either know where they should be, or they do not know at all. And the newly added vertex through the interactive GUI does not apply here. Their position is decided by the canvas itself.

The few lines of code creates a local copy of the vertexes that the graph has, and then sorts the vertexes according to the index(internal unique number), and lay them at corresponding position. Because the scale of the grid is set and there are small chances that the number of vertexes is equal to the number of grids. With this possibility of space left, there are two ways of filling the grid, so if *rowFlag* == *true* (which is the default value), then it follows the fashion of filling the row first. Otherwise, it will fill the column first.

```
        private void calculate() {

```

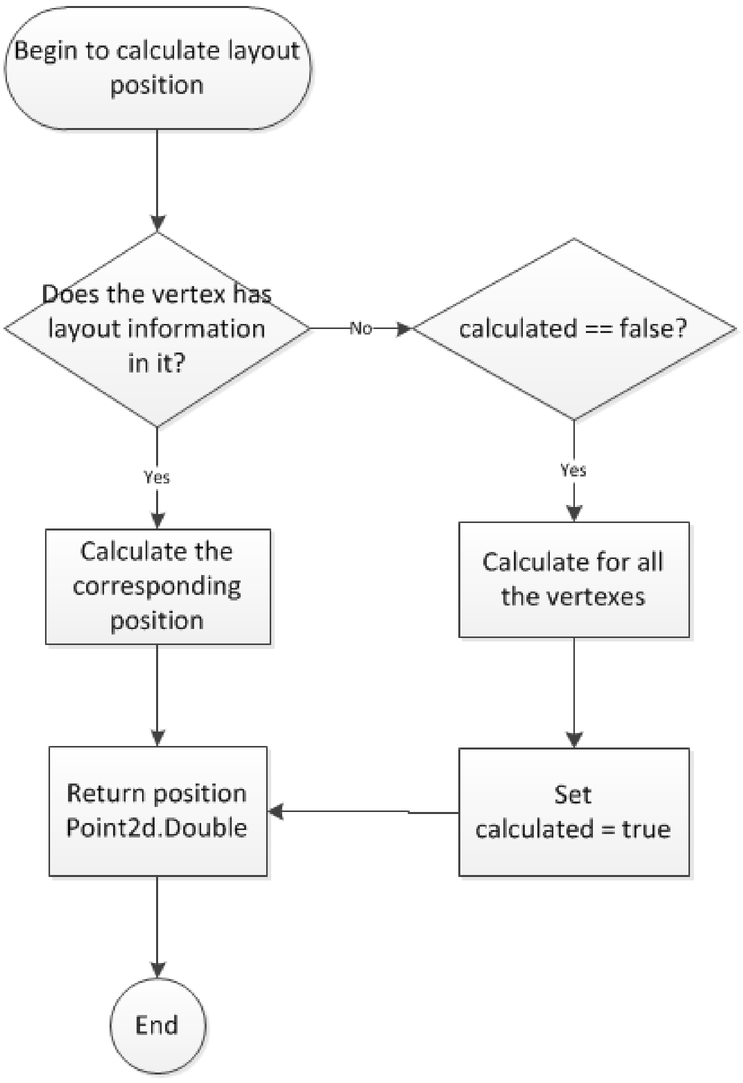

Begin to calculate layout position
Does the vertex has layout information in it?
No
calculated == false?
Yes
Yes
Calculate the corresponding position
Calculate for all the vertexes
Return position Point2d.Double
Set calculated = true
End

Figure 4.7: Flowchart of BasicGridLayout

```
ar = (Vertex[]) new basic.Vertex[graph.getVertexCount()];
int count = 0;

// copy into array
for (Vertex v : graph.getVertices()) {
        ar[count] = v;
        count++;
}
// sort
Arrays.sort(ar);

count = 0;
while (count < ar.length) {
        if (rowFlag) {
                // row first
                this.setLocation(ar[count],
                                                     disY * (count % col) + offsetY
                                                        ,disX * (count / col) +
                                                        offsetX);
                          } else {
                               // column first
                               this.setLocation(ar[count], disY * (count /
                                   row) + offsetY ,disX * (count % row) +
                                   offsetX
                                                     );

                          }
                          count++;
                     }
                }
```

Calculation process

VertexLayoutSpec has a private field to save the coordinates of the vertex, as shown in its class diagram in figure 4.1 and the code snippet below.

```
private Pair<Integer> position
```

The reason that position takes *Integer* as argument other than *Double* or *Float* is that position indicates where the vertex is by storing the relative coordinates in the (row,column) pair. The design originally intended to hold (row,column) information for the BasicGridLayout class. And thus kept the Integer as always. Strictly speaking, the vertex should not care much about the actual position in detail, it should be the canvas/container's responsibility.

Other than the constructors for the BasicGridLayout class, the API signatures below enable users to have access to its layout information.

```
public Pair<Integer> getPosition()
public void setPosition(int row,int col)
```

The calculation process goes like this: suppose the pair of double number (x, y) represents the coordinates of the vertex in the user space, which is also the value to be returned. And suppose that the VertexLayoutSpec holds the position of the vertex to be (a, b), the pair of numbers stored in type integer. Then the actual lay out position should be calculated following the equation 4.1.

$$(x, y) = (horizontalGap \times b + offsetX,\ verticalGap \times a + offsetY) \tag{4.1}$$

The BasicGridLayout API

As shown in the table 4.1, aside from the getters and setters of the BasicGridLayout class, it also provides four constructors. The following table lists constructors of the BasicGridLayout class that specify the number of rows and columns, and the additional parameter of horizontal gaps and vertical gaps.

Constructor	Purpose
BasicGridLayout(*Graph<Vertex, E> graph, int row, int col)*	Creates a grid layout with the specified number of rows and columns. All components in the layout are given equal size.
BasicGridLayout(*Graph<Vertex, E> graph, int row, int col, int hgap, int vgap)*	Creates a grid layout with the specified number of rows and columns. In addition, the horizontal and vertical gaps are set to the specified values. Horizontal gaps are places between each of columns. Vertical gaps are placed between each of the rows.
BasicGridLayout(*Graph<Vertex, E> graph, int row, int col,boolean rowFlag)*	Creates a grid layout with the specified number of rows and columns. With a boolean to set the filling by row or column.
BasicGridLayout(*Graph<Vertex, E> graph, int row, int col,int hgap, int vgap,boolean rowFlag)*	Creates a grid layout with the specified number of rows and columns, and the horizontal and vertical gaps. With a boolean to set the filling by row or column

Table 4.1: The BasicGridLayout class constructors

4.3.3 Exceptions

Exception design is also a part of the whole system. With the help of the Eclipse IDE, it is easy to get things through without rasing errors or warnings. We have developed some exceptions of ourselves, as Thinking in Java Eckel (2006) noted: "To create your own exception class, you¡¯re forced to inherit from an existing type of exception, preferably one that is close in meaning to your new exception." This is easy and straightforward. However, when choosing the right exception to use (or extend), it becomes more than just reading the documentation.

IllegalArgumentException vs. NullPointerException

The problem is from the interface shown below that defines the signature where null parameter is not appropriate for the situation. So we wonder for this specific question, should we use the IllegalArgumentException or NullPointerException.

```
public boolean moveData(Vertex<T> startNode,
                    Vertex<T> targetNode, T d, Graph<Vertex<T>,E> g)
                        throws Exception;
```

With the help of references, we come to the conclusion that a NullPointerException should be the one to be thrown. Like the one shown below.

```
if (d == null || targetNode == null || sourceNode == null) {
                    throw new NullPointerException();
```

There are several reasons for this. Yet first and foremost, we shall look at where in the class hierarchy these two class lie. For IllegalArgumentException and NullPointerException, the class hierarchy of each class is listed in the code snippet below.(from Java Documentation see Oracle (2011)) Obviously, these two classes belongs to the same category as they both are subclass of the java.lang.RuntimeException. From the description in the previous chapter: they all belongs to the unchecked exception. So they should be treated the same under rules.

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.IllegalArgumentException
```

```
java.lang.Object
  java.lang.Throwable
      java.lang.Exception
          java.lang.RuntimeException
              java.lang.NullPointerException
```

Some thoughts that supports the decision of preference of IllegalArgumentException over NullPointerException [10]:

First, the Java Documentation explicitly lists the cases where NullPointerException is appropriate. Notice that all of them are thrown by the runtime when null is used inappropriately. In contrast, the IllegalArgumentException JavaDoc could not be more clear: "Thrown to indicate that a method has been passed an illegal or inappropriate argument."

Second, NullPointerException in a stack trace is prone to indicate that someone dereferenced a null. Whereas IllegalArgumentException might lead to the assumption that the caller of the method at the top of the stack passed in an illegal value. Again, the latter assumption is true, the former is misleading.

Third, the name tells everything. Since IllegalArgumentException is clearly designed for validating parameters, as we can find out in the its name, you have to assume it as the default choice of exception.

Fourth, all other incorrect parameter data will be IllegalArgumentException, so why not be consistent? An illegal null is not so special that it deserves a separate exception from all other types of illegal arguments.

By referencing the "book of rule" Effective Java 2nd Edition (Bloch (2008)), we could also easily come to the conclusion, as the book puts:"Arguably, all erroneous method invocations boil down to an illegal argument or illegal state, but other exceptions are standardly used for certain kinds of illegal arguments and states. If a caller passes null in some parameter for which null values are prohibited, convention dictates that NullPointerException be thrown rather than IllegalArgumentException. Similarly, if a caller passes an out-of-range value in a parameter representing an index into a sequence, IndexOutOfBoundsException should be thrown rather than IllegalArgumentException."

[10]http://stackoverflow.com/questions/3881/illegalargumentexception-or-nullpointerexception-for-a-null-parameter

Chapter **5**

Testing, Evaluation and Case studies

In this chapter, we are going to test the graph library by applying it to two non-trivial case studies. And following that, we will discuss the testing and evaluation of our library, in the hope to present a more comprehensive coverage of the library.

The two cases are:

- Mutual authentication using public key
- To simulate a lock system

These two cases fall right into the designed field of application of the library.

5.1 Case 1: Mutual authentication

The first case is a mutual authentication case, where two parties authenticate each other. According to wikipedia, in technology terms, it refers to a client or user authenticating themselves to a server and that server authenticating itself to the user in such a way that both parties are assured of the others' identity. When describing online authentication processes, mutual authentication is often referred to as website-to-user authentication, or site-to-user authentication[1].

National Institute of Standards and Technology also has a standard regarding this topic, and there definition goes:"This standard specifies two challenge-response protocols by which entities in a computer system may authenticate their identities to one another."U.S. DEPARTMENT OF COMMERCE and Technology (1997)

There are several different kinds of authentication mechanisms, the discussion of which is beyond the scope of this thesis, but here we would like to show one. The figure 5.1 shows what occurs during username and password-based mutual authentication[2].

5.1.1 Analysis, Design and Implementation

The requirement is to use a graph to show the interactions, which can be a typical application of the library. To make things simple, we decided to use the very basic undirected simple graph (contrary to the multi-graph), and the data type to be stored in vertexes is set to *String*. And it is out of the same intuition, as this should not be a major concern in this case and thus we do not put

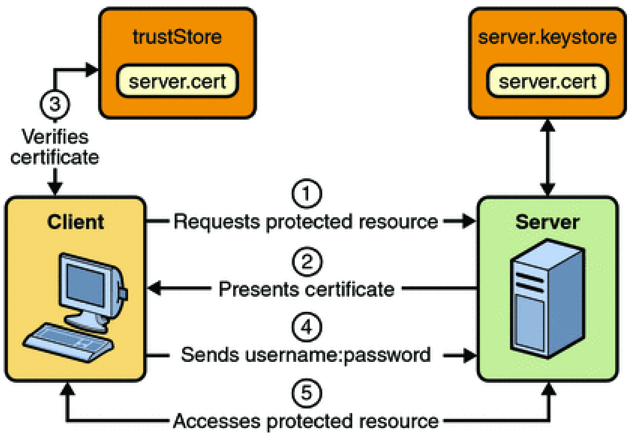


Figure 5.1: User Name- and Password-Based Mutual Authentication.

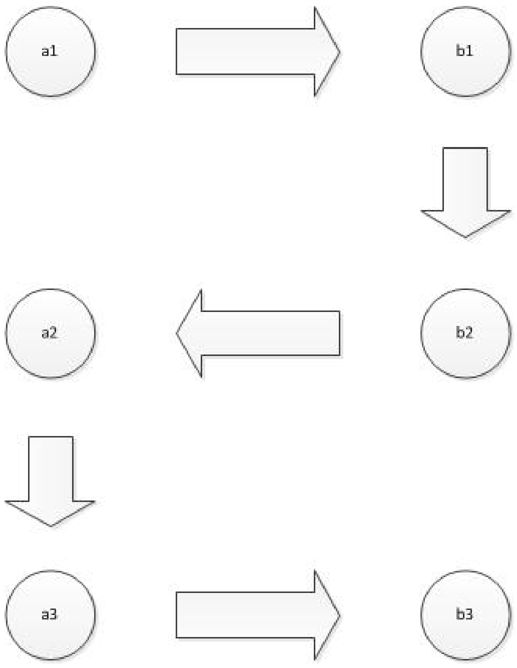


Figure 5.2: Information flow path denoted in the figure, from **a1** to **b3**.

further effort in here. The figure 5.2 shows the interactions between two parties as time develops.

As for this case, a two way authenticating situation, we introduce two series of vertexes into the graph, one of them in the **a** series and the other the **b** series, there trailing number indicates the development over time. And to be consistent with the scenario, the a actually stands for the Client side of the authentication and the b series represents the Server side. There communication and calculation on each side is abstracted away, with only the communication between them left. Concretely, we add four vertex of one side, namely **[a1,a2,a3,a4]**, to the graph, and set their color to be green; on the other side, four vertex **[b1,b2,b3,b4]** are added to the graph. Each vertex holds some random information represented in String. The figure shows the result after several round of data exchange between vertexes. Concretely, **a1** holds the information in *String* that is to be passed around: *"to be moved"*. And the data is passed between vertexes in the following order:

```
a1 -> b1 -> a2 -> b2 -> a3 -> b3
```

And we can verify the result by visual output in figure 5.4 and the console output that **b3** holds the data *"to be moved"*.

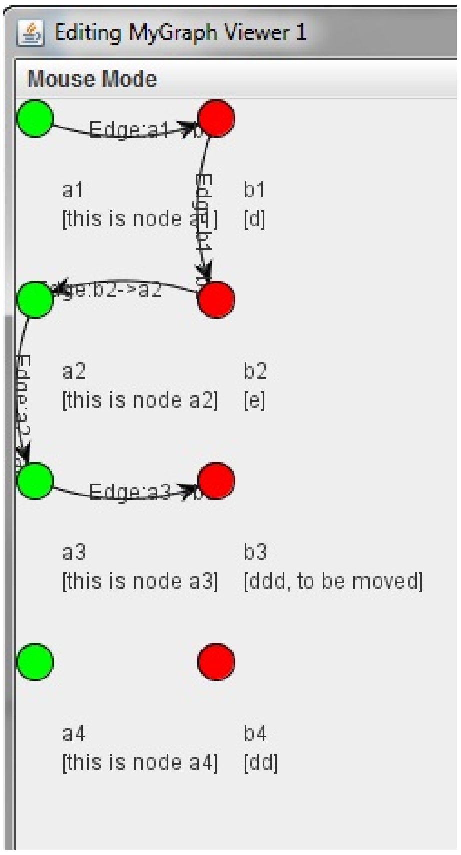


Figure 5.3: Print screen result of the first case.

[1]http://en.wikipedia.org/wiki/Mutual_authentication
[2]http://docs.oracle.com/cd/E19226-01/820-7627/bncbs/index.html

Implementation

The case is implemented using the library and wrapped into the class *Demo1.java* .

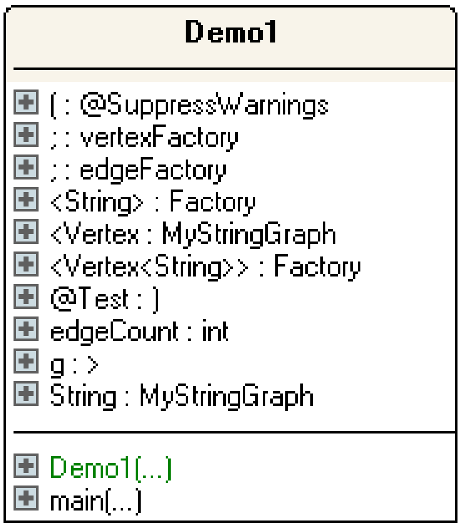


Figure 5.4: Class diagram of demo1 for the above discussed case.

The code snippet below creates the graph that holds the vertexes and the edges. The two factory objects are for the purpose of creating Vertex<String> and Edge<String> objects on the fly, when user operates on the graphical user interface. The static variable edgeCount is kept only for the naming of the edge.

```
        MyStringGraph g;
        static int edgeCount;
        Factory<Vertex<String>> vertexFactory;
        Factory<Edge<String>> edgeFactory;
```

The following initialize the fields of this class and set the layout convention to be a 4 by 2 grid layout with the column filling first. We set the string of "to be moved" as the data that needs to be passed around in the system and initially it is added to the vertex a1.

```
    Demo1 sgv = new Demo1();
        Layout<Vertex, String> layout = new BasicGridLayout(sgv.g,4,2,false);
        layout.setSize(new Dimension(500, 500));

        //a
        Vertex a1 = new MyVertex("a1");
        Vertex a2 = new MyVertex("a2");
        Vertex a3 = new MyVertex("a3");
        Vertex a4 = new MyVertex("a4");

        sgv.g.addVertex(a1);
        sgv.g.addVertex(a2);
        sgv.g.addVertex(a3);
        sgv.g.addVertex(a4);

        String dataString = "to␣be␣moved";
        a1.addData(dataString);
```

Following this fashion we get the b series of vertex added into the graph. With a little manipulation of the color of the b series to differentiate them from a series. Shown in the code below.

```
        for(AbstractVertex iter: sgv.g.getVertices()){
                        iter.setLayoutSpec(new VertexLayoutSpec(Color.green));
                        iter.addData("this␣is␣node␣" + iter.getName());
                }
```

And now we are ready for the information exchange operation.

```
    //move data
        sgv.g.moveData(a1, b1, dataString, sgv.g);
        sgv.g.moveData(b1, b2, dataString, sgv.g);
        sgv.g.moveData(b2, a2, dataString, sgv.g);
        sgv.g.moveData(a2, a3, dataString, sgv.g);
        sgv.g.moveData(a3, b3, dataString, sgv.g);
```

We do the following simple print out to verify the result, we check every vertex the graph contains and look for the dataString in it.

```
    //local verification that the nodes holds the data.
        for(Vertex<String> v: sgv.g.getVertices()){
                System.out.println(v.getName() +"␣containts␣the␣data:␣" +v.
                    getData().contains(dataString) );
        }
```

Result should be: b3 should contain the dataString whereas all other vertexes should not. And are shown below.

```
    a4 containts the data: false
    b1 containts the data: false
    a2 containts the data: false
    a3 containts the data: false
    b4 containts the data: false
    b2 containts the data: false
    b3 containts the data: true
    a1 containts the data: false
```

In this part we shall briefly introduce the structure of user interface. We first have a VisualizationViewer that extends Java panel class, so this is the canvas for display.

```
        // actually visualizationViewer is a subclass of panel
                VisualizationViewer<Vertex, String> vv = new
                    VisualizationViewer<Vertex, String>(
                            layout);
```

5.2 Case 2: Lock system

The second case is about an simulation of a lock system, for example an access control system that identifies people exerting control over who can interact with a resource. Access control is, in reality, an everyday phenomenon. A lock on a car door is essentially a form of access control. A PIN on an ATM system at a bank is another means of access control. The possession of access control is of prime importance when persons seek to secure important, confidential, or sensitive information and equipment.[3]

In general there are two categories of lock systems: physical securities and computer securities. Physical access by a person may be allowed depending on payment, authorization, etc. Also there may be one-way traffic of people. In physical security, the term access control refers to the practice of restricting entrance to a property, a building, or a room to authorized persons.

In computer security, access control includes authentication, authorization and audit. It also includes measures such as physical devices, including biometric scans and metal locks, hidden paths, digital signatures, encryption, social barriers, and monitoring by humans and automated systems.

The above description of the problem cries for an representation using graph/networks. And that is what we do next, to model a lock system.

5.2.1 Analysis, Design and Implementation

To model a lock system, we need first to have a map of it. The figure 5.5 below shows the map. There are few points that worth noticing:

- On the first row, the path is bidirectional, which means one can move back and forward as one wishes;
- On the second row, the "CL" stands for Cypher Lock, which is a lock itself, and a method for the access control;
- On the second, third and the last rows, there are some abbreviations that needs clarification: srv = surveillance, usr = user, jan = janitor;
- On the last line, the entities represented are somehow different from previous ones. They represent PCs and thus are locations on the net.

These information are crucial not only in the case itself, but also in designing the library. First we need to find the right objects. There are clearly two different entities that should be abstracted out, net location and physical location. The vertexes on the rows except the last are physical locations, whereas the last row represents the net location. We here make them simple subclass of MyVertex class. It is though that they do not provide its own functionality or properties, it is open for change and evolve. The code snippet are listed below.

[3] http://en.wikipedia.org/wiki/Access_control

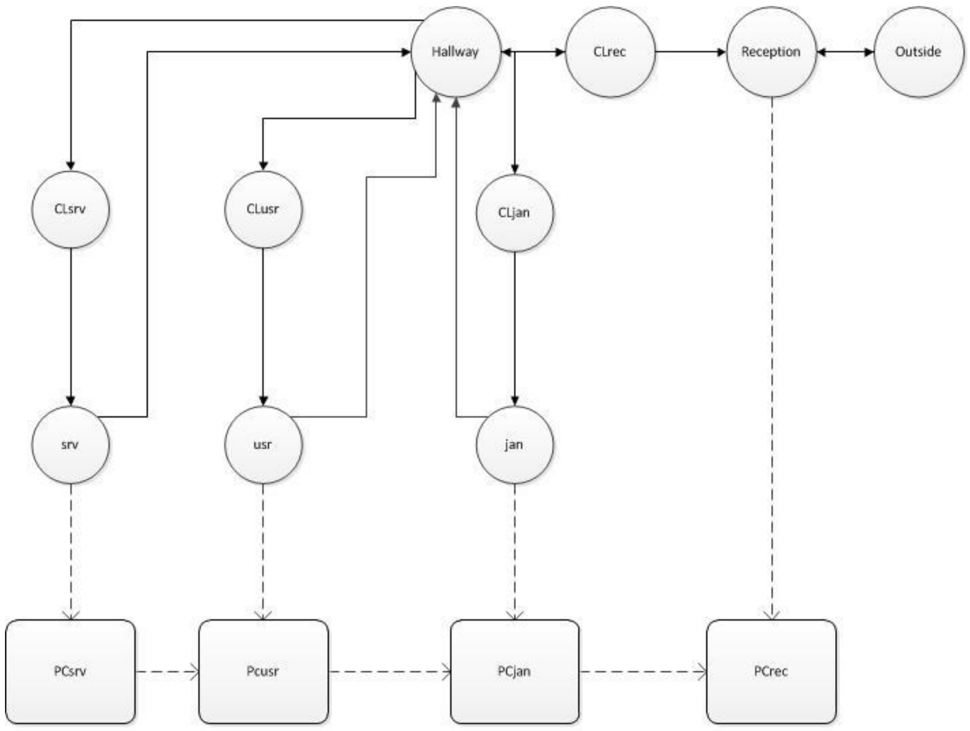


Figure 5.5: object relationship for case 2.

```
public class PhysLocation extends MyVertex
public class NetLocation extends MyVertex
```

And through these, we then have four different kinds of edge entities as shown in the figure 5.6. We have the class called NPEdge which stands for net location to physical location edge. And so do we have NNEdge, PNEdge, PPEdge. Again these classes showed no more than properties of the edge class, but they should be essential for future development, they can be a control or restriction over the accessability of personnel. Especially combined with the power of predicate of JUNG library.

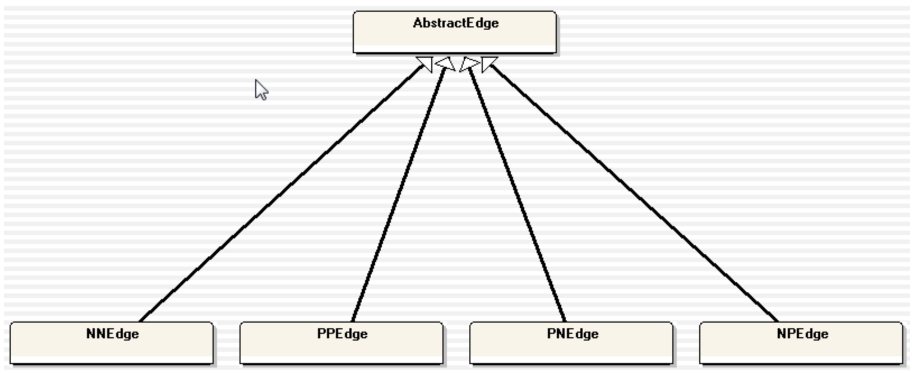


Figure 5.6: Class diagram of edge and its subclasses

Like the NPEdge class below, it depicts an edge with string data having source node being NetLocation and target node being PhysLocation.

```
public class NPEdge extends AbstractEdge<NetLocation, PhysLocation, String>
```

Now let us look at the demo class it self. All the initialization and operations are here in demo 2 class, as shown below in figure 5.7.

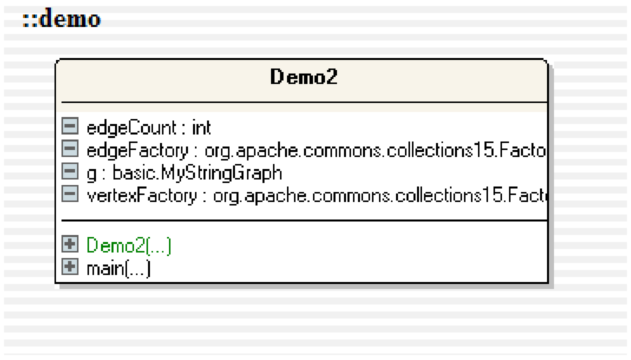

::demo
Demo2
edgeCount : int
edgeFactory : org.apache.commons.collections15.Facto
g : basic.MyStringGraph
vertexFactory : org.apache.commons.collections15.Fact
Demo2(...)
main(...)

Figure 5.7: demo2 class diagram

And similar to the previous case, there are factory methods to create objects of vertexes and edges.

```

                vertexFactory = new Factory<Vertex<String>>() { // My vertex
                    factory
                        public Vertex<String> create() {
                                Vertex<String> n1 = new MyVertex();
                                n1.setLayoutSpec(new VertexLayoutSpec(Color.
                                    black));
                                return n1;
                        }
                };

                edgeFactory = new Factory<Edge<String>>() { // My edge factory
                        public Edge<String> create() {
                                return new Edge<String>("E" + String.valueOf(
                                    edgeCount));
                        }
                };
```

We set the space layout to be 4 by 4. And initialize each vertex row by row. The difference of layout lies in here, we explicitly tell these vertex where to stay. For example, a1 is set to be at first row, first column. They after all the vertexes are added to the graph, we carefully establish edges between them as shown in the graph. Note that we use undirected edge to represent bidirectional directed edges, this on one hand lower the workload, on the other hand, also simplify the generated user interface.

```
                Layout<Vertex, String> layout = new BasicGridLayout(sgv.g,4,4)
                    ;
        //first row
                Vertex<String> a1 = new NetLocation("Hallway");
        //here we set the layout spec of each node
                a1.setLayoutSpec(new VertexLayoutSpec(1,1));
        ...
        //set edges
                sgv.g.addEdge(sgv.edgeFactory.create(), new Pair<Vertex<String
                    >>(a3,a4));
                sgv.g.addEdge(sgv.edgeFactory.create(), new Pair<Vertex<String
                    >>(a1,b1),EdgeType.DIRECTED);
```

Lastly, we pass the data around, and copied at the NetLocation in the figure, at the end successfully pass the data back to the start point. With the code snippet below and result of demo2 in figure 5.8

```
                sgv.g.passData(a4, a3, "Janitor", sgv.g);
                sgv.g.passData(a3, a2, "Janitor", sgv.g);
                sgv.g.passData(a2, a1, "Janitor", sgv.g);
                sgv.g.passData(a1, b3, "Janitor", sgv.g);
                sgv.g.passData(b3, c3, "Janitor", sgv.g);
                sgv.g.copyData(c3, d3, "Janitor");
                sgv.g.copyData(d3, d4, "Janitor");
                sgv.g.copyData(d4, a3, "Janitor");
                sgv.g.passData(a3, a4, "Janitor", sgv.g);
```

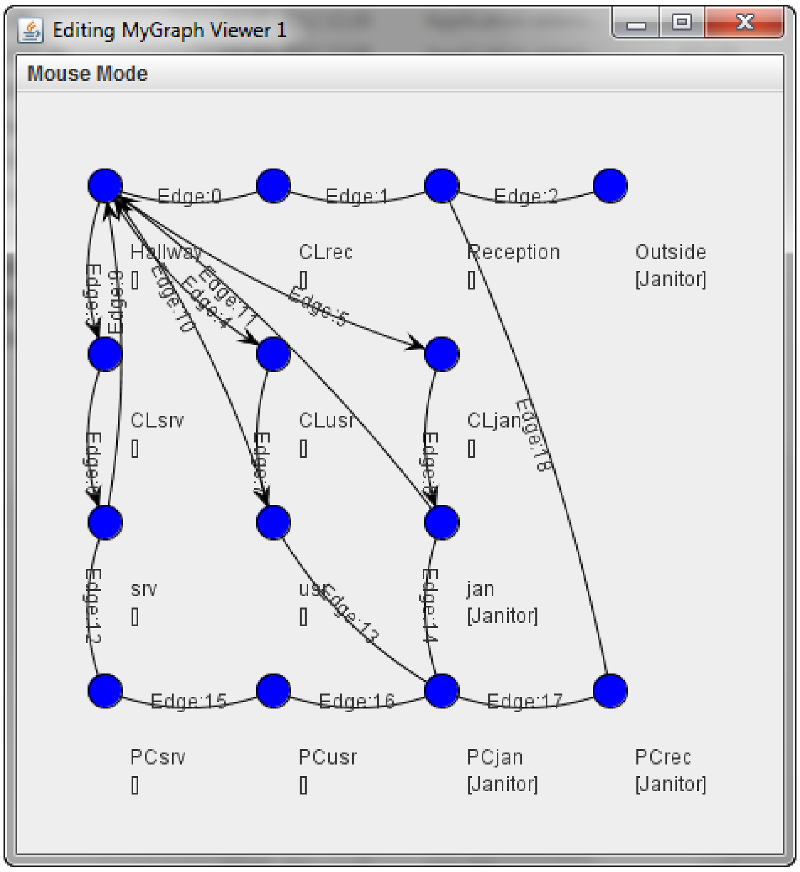


Figure 5.8: The result of case 2

Chapter **6**

Conclusion and Future Work

This project has achieved the goal we mentioned in the introduction chapter, which is "to build a library for the analyze, visualize and manipulation of graph and networks", as we can see from the demos given. Through the process of doing this project, I have acquired a great deal of knowledge in the field of software engineering, especially in writing an API library. Starting from the choice of base library, to determine interfaces and operation signatures, every step on the way requires deliberation on decisions to be made.

Aside from the achievement, there are and always will be something that can be improved. So here we list some thoughts on future work:

- The evaluation of libraries is based on some intuitive assumptions about what kind of operations are of the first priority in our domain. It would be advisable if we can make an detailed investigation into the requirements and list out the operations needed, and comes with a more informed decision.
- More cases are required, not only to demonstrate usage of this library, but to provide requirements that can enrich the content of this library.
- Better coding styles and techniques would make building this library more effortless. It is though that implementation details can be kept transparent to users, a "beauty inside" would not only make things pleasant, but also reduce the difficulties when refactor and evolving of the library are needed in future.

Hopefully, this work shall be applied elsewhere as a step stone towards a better and more comprehensive library.

A. Benchmark Program

A.1. BasicBenchmark.java

```java
package Sample.Benchmark;
import edu.uci.ics.jung.graph.*;

/**
 *
 * @author Logan
 */
public class BasicBenchmark {

    public long addVer(int num) {
        System.out.println("Benchmarking␣add␣vertex.");
        SparseGraph<Integer, String> g = new SparseGraph<Integer, String>();
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < num; i++) {
            g.addVertex((Integer) i);
        }

        long endTime = System.currentTimeMillis();
        long time = endTime - startTime;
        System.out.println("Total␣time:␣" + time + "␣ms.");
        return time;
    }

    public static void main(String[] args) {
        BasicBenchmark b = new BasicBenchmark();
        long total = 0;
        double times = 10;
        for (int i = 0; i < times; i++) {
            total+=b.addVer(1000000);
        }
        System.out.println("Average␣time:␣" + total/times + "␣ms.");
    }
}
```

A.2. BasicBenchmarkJung.java

```
package benchmark;

import edu.uci.ics.jung.graph.Graph;
import edu.uci.ics.jung.graph.SparseGraph;


/**
 *
 * @author Logan
 */
public class BasicBenchmarkJung implements BasicBechmark {
        /* (non-Javadoc)
         * @see benchmark.BasicBechmark#addVer(int, edu.uci.ics.jung.graph.
             Graph)
         */
        @Override
        public long addVer(int num, Graph<Integer, String> g) {
                long startTime = System.currentTimeMillis();
                for (int i = 0; i < num; i++) {
                        g.addVertex((Integer) i);
                }

                long endTime = System.currentTimeMillis();
                long time = endTime - startTime;
                System.out.println("Running␣time:␣" + time + "␣ms.");
                return time;

        }

        /* (non-Javadoc)
         * @see benchmark.BasicBechmark#addEdge(int, edu.uci.ics.jung.graph.
             Graph)
         */
        @Override
        public long addEdge(int num, Graph<Integer, String> g) {
                for (int i = 0; i < num; i++) {
                        g.addVertex((Integer) i);
                }

                long startTime = System.currentTimeMillis();

                for (int j = 0; j < num - 1; j++) {
                        g.addEdge(String.valueOf(j), j, j + 1);
                }

                long endTime = System.currentTimeMillis();
                long time = endTime - startTime;
                System.out.println("Running␣time:␣" + time + "␣ms.");
                return time;

        }
}
```

A.3. BasicBenchmarkJung.java

```
package benchmark;

import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.view.mxGraph;

public class BasicBecnhmarkjGraphX {

        final static int scale = 100000;
        final static double times = 10.0;

        public static long addVer() {
                final mxGraph graph = new mxGraph();
                Object parent = graph.getDefaultParent();
                graph.getModel().beginUpdate();
                long startTime = System.currentTimeMillis();
                long time;
                try {
                        for (int i = 0; i < scale; i++) {
                                graph.insertVertex(parent, null, String.
                                    valueOf(i), 0, 0, 0, 0);
                        }

                        long endTime = System.currentTimeMillis();
                        time = endTime - startTime;
                        System.out.println("Running␣time:␣" + time + "␣ms.");
                } finally {
                        graph.getModel().endUpdate();
                }
                return time;
        }

        public static long addEdge() {
                final mxGraph graph = new mxGraph();
                Object parent = graph.getDefaultParent();
                graph.getModel().beginUpdate();
                Object[] ar = new Object[scale];
                long startTime;
                long time;
                try {
                        for (int i = 0; i < scale; i++) {
                                ar[i] = graph.insertVertex(parent, null,
                                    String.valueOf(i), 0, 0, 0, 0);
                        }

                        startTime = System.currentTimeMillis();
                for (int i = 1; i < scale; i++) {
                        graph.insertEdge(parent, null, String.valueOf(
                            i), ar[i] , ar[i-1]);
                }
                long endTime = System.currentTimeMillis();
                time = endTime - startTime;
                System.out.println("Running␣time:␣" + time + "␣ms.");
        } finally {
                graph.getModel().endUpdate();
        }
        return time;

    }

    public static void main(String[] args) {
        long total = 0;
        for (int i = 0; i < times; i++)
                total += addVer();
        System.out.println("===Average␣time:␣" + total / times + "␣ms
            .===\n");


        long Edgetotal = 0;
        for (int i = 0; i < times; i++)
                Edgetotal += addEdge();
        System.out.println("===Average␣time:␣" + Edgetotal / times + "
            ␣ms.===\n");

    }
}
```

A.4. BasicBechmark.java

```
package benchmark;

import edu.uci.ics.jung.graph.Graph;

public interface BasicBechmark {

    /**
     * Basic sparse graph testing: add 1 million nodes into the graph.
     *
     * @param num
     * @return processing time
     */
    public abstract long addVer(int num, Graph<Integer, String> g);

    public abstract long addEdge(int num, Graph<Integer, String> g);

}
```

Bibliography

Bloch, J.: 2008, Effective java: Second edition.

Cormen Thomas H., Leiserson Charles E., R. R. L. S. C.: 2001, *Introduction to Algorithms (Second ed.)*, MIT Press and McGraw-Hill.

Eckel, B.: 2006, *Thinking In Java (Fourth ed.)*, Prentice Hall; 4 edition.

Erich Gamma, Richard Helm, R. J. J. V.: 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Harold, E. R.: 2004, Xom design principles.

Joshua O Madadhain, Danyel Fisher, P. S.: 2005, Analysis and visualization of network data using jung.

NetBeans, W.: 2012, Api design.

Oracle: 2011, Java 2 platform api specification standard edition (version 1.6).

Oracle: 2012, The java tutorials.

Shenoy, S.: 2002, Best practices in ejb exception handling.

U.S. DEPARTMENT OF COMMERCE, N. I. o. S. and Technology: 1997, Entity authentication using public key cryptography, *Federal Information Processing Standards Publication* **1**96, 3974–3981.