LAYOUT ADJUSTMENT ALGORITHM FOR CLASS DIAGRAMS

by

Casey Robinson

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Computing and Information Systems

in the

Computer Science and Information Systems

Program

YOUNGSTOWN STATE UNIVERSITY

May, 2012

Layout Adjustment Algorithm for Class Diagrams

Casey Robinson

Signature:

Casey Robinson, Student Date

Approvals:

Bonita Sharif, Thesis Advisor Date

Kriss A. Schueller, Committee Member Date

Robert Kramer, Committee Member Date

Peter J. Kasvinsky, Dean of School of Graduate Studies and Research Date

Abstract

Unified Modeling Language (UML) class diagrams depict structural design of software. They present a set of classes and relationships between classes. Current state-of-the-art tools generate class diagram layouts solely based on aesthetics and not on how easy the diagram is to comprehend. In prior work, a family of empirical studies investigated the impact of diagram layout on comprehension. The layouts were based on architectural importance of classes defined by entity, control, and boundary stereotypes defined in the UML. The results from prior work show that layout plays a significant role in the comprehension of UML class diagrams and does not favor aesthetics as much. Given a diagram produced by current tools, the goal of this work is to adjust the layout of a class diagram to make it more comprehensible. The thesis presents preliminary work in developing such an algorithm with the goal of clustering classes that are related based on certain user-defined criteria and not just aesthetics. The algorithm takes a dynamic force-based approach. The user-defined criteria are configurable via a script. A prototype of the algorithm was tested on reverse engineered diagrams from open source GUI frameworks Qt and wxWidgets and satisfactory results were produced within seconds.

Contents

List of Figures

List of Tables

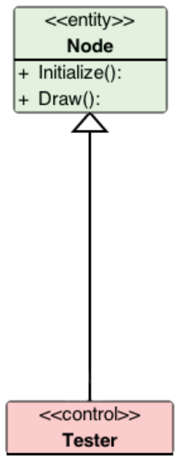
Chapter 1

Introduction

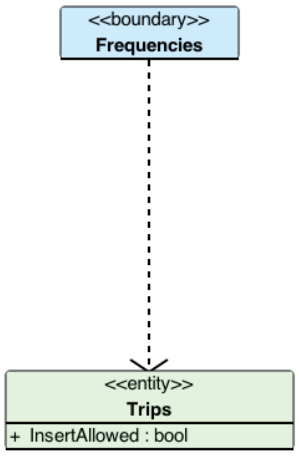
The Unified Modeling Language [7] [3] is the standard notation used in representing design of object oriented systems. The UML consists of many types of diagrams depicting both static and dynamic aspects of design. The thesis focuses on the UML class diagram which represents the static structure of design.

A UML class diagram is a graph that consists of nodes and edges. A node in a UML class diagram is a class in an object oriented system. An edge in a UML class diagram represents a relationship between two classes in the system. There are mainly three different kinds of relationships in UML class diagrams: generalizations, dependencies, and associations. Generalizations are depicted in a class hierarchy with a solid line and an arrow on one end (the superclass end). Dependencies are shown as a dashed line. Plain associations are shown as a solid line connecting two classes. Associations can be further categorized into compositions and aggregations. Compositions are shown with a filled diamond on one end (the class with the whole part of the relationship). Aggregations are shown with a open diamond on one end (usually the whole part of the relationship). Refer to figure 1.1 illustrating these concepts in a simple UML class diagram for illustration purposes.

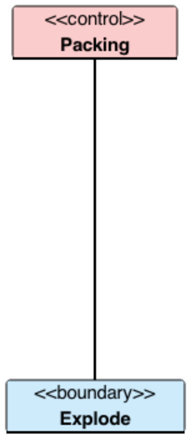
Stereotypes are an extension mechanism provided by the UML that allow users to define additional semantics [16]. In this thesis, three different stereotypes of control, boundary, and entity (based on Booch et al., [16]) are used. A boundary class is responsible for interaction between the system and the external world. An entity class stores persistent information in the system. A control class manages other objects in the system. Stereotypes are shown above the class name. Based on previous work [6] the stereotypes are color coded as follows: red for control classes, green for entity classes, and blue for boundary classes. Refer to figure 1.1 to see how stereotypes are represented visually.



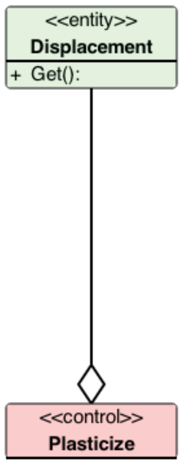
(a) Generalization



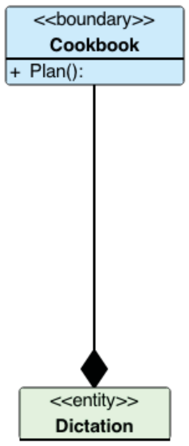
(b) Dependency



(c) Association



(d) Aggregation



(e) Composition

Figure 1.1: Example Drawings of Relationship Types

1.1 Motivation

Even though, a UML class diagram is a graph, there are aspects that make it different and unique from general graphs. General graph drawing algorithms such as the Sugiyama algorithm [26] do not work well for UML class diagrams. A node in a class diagram is not a circle, but consists of a rectangle split into three parts. Each node varies in size across both the width and height dimensions.

There are a handful of drawing algorithms (mainly by two teams of researchers) designed to automatically layout class diagrams. [[10] [9] [11] [12] [14]] Current layout drawing algorithms for UML class diagrams focus on optimizing aesthetic criteria some of which include minimizing whitespace, reducing line crossings and minimizing the angle between classes. Many of these criteria are direct opposites, thus an algorithm based on aesthetics must find a balance. This is not an ideal scenario since it introduces many subjective choices during the algorithm design. See Figure 1.2 for an example diagram produced by Visual Paradigm. an industrial UML tool for development. The class model and the diagram were reverse engineered directly from source code. This diagram is not very easy to understand. It generates layers of classes in no particular semantic way. Two classes that are related are spaced far apart. By focusing on a rectangular layout related classes are on opposite sides of the diagram. Sticking to a rigidly orthogonal hierarchical layout creates a stratified diagram.

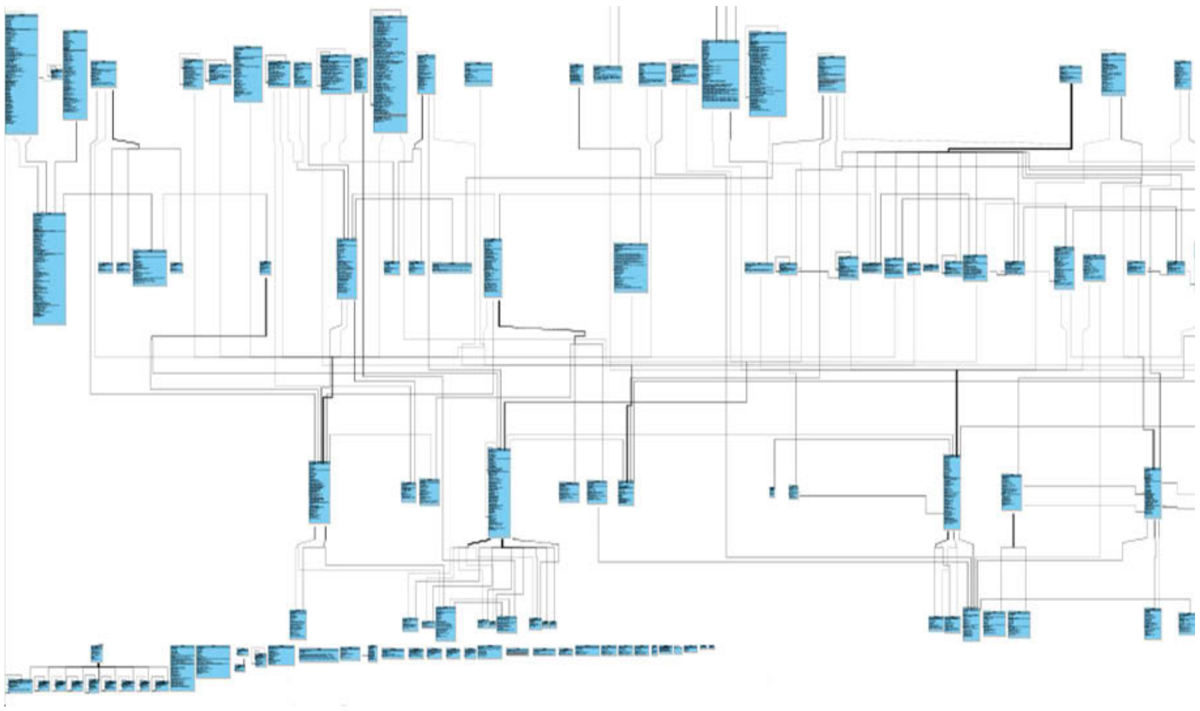


Figure 1.2: A layout generated by Visual Paradigm in hierarchical layers.

In previous work [[6], [23], [22], [24], [25]] , a family of empirical studies have shown that layout of UML class diagrams significantly impacts comprehension. Several types of tasks [22] such as bug fixing, new feature addition, impact analysis, design pattern identification [24] , and refactoring, to name a few were studied in detail. A result of these studies was a set of empirically validated guidelines [21].

1.2 Contributions

The focus of this thesis is to realize the above guidelines into a layout algorithm for UML class diagrams because currently no layout algorithms implement this idea.

The algorithm presented in this thesis attempts to solve the issues presented above by changing the criteria that is most relevant to the task at hand and not focusing mainly on aesthetics. The premise here is that a pretty diagram is not always comprehensible. There are two paradigms that can be used; static and dynamic. Static systems solve a system of equations and use the results to place nodes. This is the basis for most algorithms since the aesthetic criteria are directly translated into a system of equations. Dynamic systems iteratively move nodes based on concepts from physics. This is a more computationally intensive process than static systems.

The purpose is not to generate a class diagram from scratch. Instead, the input to the system is an existing class diagram. This approach is referred to by the literature as a layout adjustment algorithm [8]. We view this as complementary to existing UML class diagram layouts that generate diagrams by reverse engineering source code. The algorithm should be easily pluggable to layouts generated by existing state-of-the-art algorithms, which is the main over-arching goal of the algorithm in its final form. This aspect makes the algorithm easier and accessible to use. The algorithm presented in this thesis is a first attempt at designing a proof-of-concept tool. We illustrate how this can be built upon in the future work section.

1.3 Organization

The thesis is organized as follows. Chapter 2 talks about prior work in the field. The approach used in the proposed algorithm is described in Chapter 3. Chapter 4 shows the output of the algorithm on three diagrams generated from two open source systems. Chapter 5 finally concludes the thesis and presents future work.

Chapter 2

Related Work

This section discusses prior work done in the field of aesthetics and empirical studies to UML class diagrams. Aesthetics for general graphs are also presented.

2.1 Aesthetics for General Graphs

A class diagram is a graph. This means the first obvious direction would be to look at graph algorithms that would adapt to the class diagram. Many algorithms exist for visualizing graphs and Battista et al. provide a detailed analysis [8]. There are three basic categories of algorithms; force-directed, hierarchical, and topology-shape-metrics.

Force-directed algorithms apply forces along each edge and simulate the system with concepts from physics. This is the approach taken in this thesis. The hierarchical algorithms create layers of vertices, reorder vertices to reduce line crossings, and map to a coordinate system for drawing. The topology-shape-metrics algorithms borrow concepts from topology; planarization, orthogonalization, and compaction.

Graph aesthetics focus on the following criteria: total number of edge crossings, area of the drawing, total edge length, maximum length of the edge, large differences between edge lengths, bends, maximum number of bends, large differences between number of bends on edges, aspect ratio of the drawing, smallest angle between two edges incident on the same vertex and symmetry. The latter two are maximized and the former are attempted to be minimized.

Huang et al. claim that current algorithms do not consider node size and therefore produce expanded layouts.[15] The authors present a method for adjusting the layout and state that it can be applied to UML diagrams. However no implementation or concrete description is given. This is additional motivation for creating a layout adjustment algorithm for UML class diagrams.

2.2 Empirical Studies on Aesthetics for General Graphs

Purchase studied the importance of five graph aesthetics [17]. This study aimed to prioritize the criteria based on importance to human understanding. Time and accuracy were used in this study to measure the importance. Reduction of edge crossings was the most important aesthetic, while angles and orthogonal edges were unimportant.

Purchase also studied a set of seven metrics to gauge the aesthetic quality of abstract graphs [18]. The measurement is in the form of a real number between 0 and 1 where 1 determines the positive effect of the aesthetic. The metrics are applied on a set of graphs of varied sizes. They found the value of the edge crossing aesthetic to be high since all the algorithms tried to minimize the number of edge crossings. The presented metrics will help formally quantify output produced by different algorithms.

Ware et al. [27] conducted an experiment that measures cognitive cost of graph aesthetics. Their results only apply to graphs where the task is to find the shortest paths. They looked at edge crossings, sum of the lengths of edges, edge continuity, and number of arcs coming out of a node.

In 2008, Purchase performed another experiment that focused on dynamic layout algorithms and preservation of the user's mental map. The results of this study show that extremes are better and suggest that individual preference may be important. [19]

2.3 Aesthetic Based Layouts for UML class diagrams

Ambler [5] presents a set of guidelines for UML style. They are mainly related to UML notation rather than layout. These are general guidelines for UML diagrams. An example of a general guideline would be to avoid crossing lines, or arrange symbols symmetrically. An example of a relationship guideline would be to model inheritance vertically and place subclasses below superclasses.

Current UML class diagram tools generally fail to produce useful UML layouts [11]. With this work Eichelberger et al. call for UML layout standards to improve the readability and comprehension of class diagrams. They produced fourteen aesthetic criteria designed for this purpose. The hypothesis is that conforming to aesthetics improves the layout of class diagrams. [10]

Eichelberger and Schmid provide an overview of aesthetic and diagramming guidelines for UML class diagrams. [10] They also seek to determine the importance of some of the less tested, but still promising, criteria. This was done with a timed response survey. The survey showed that there were various aspects of the layouts used for testing that were much more significant than the criteria being tested. The authors created a taxonomy of guidelines and classified rules from the literature for UML class diagram layout. They conduct a pilot study to determine the effect of the guideline rules but the study was inconclusive. The study was run on a toy system and did not involve comprehension questions; it was purely notational. On the other hand, the guidelines that drive the algorithm presented in this thesis are based on comprehension studies involving real tasks in real open source software.

Eichelberger's dissertation [9] discusses the aesthetics of UML class diagrams and implements an algorithm based on these findings called SugiBib. SugiBib starts with Sugiyama and incorporates lessons from the Seemann algorithm [20]. These algorithms were extensively modified. Two of the main features from Sugiyama are the related concepts of temporary removal and incremental extension. In temporary removal, all portions of the graph which cannot be used for the inheritance hierarchy are removed to simplify the creation of the hierarchy. Many steps later, after the hierarchy is placed, the removed nodes and edges are successively reinserted and arranged. The intermediate processing steps were introduced to realize the aesthetic criteria. Eichelberger also created a basic set of metrics to objectively compare results of layout algorithms.

Current computer aided software engineering (CASE) tools have three major problems[11]; implementation specific tools make abstractions challenging, packaging mechanisms are only logical, and unique user interfaces reduce usability. These problems create a disconnect between the UML standard and the tool. In short, current tools do not use the state of the art in diagram layout algorithms. Eichelberger et al. propose fourteen criteria based on graph aesthetics and inheritance and association relations, as well as more sophisticated model elements. They have provided a prototype and urge vendors to incorporate the state of the art layout techniques in their CASE tools.

Eiglsperger et al. [12] present two algorithms realizing the topology-shape-metrics paradigm; UML-Kandinsky and GoVisual. The two algorithms join inheritance arcs identically where as the planarization and orthogonalization phases are radially different. The main aesthetic criterion is the number of line crossings; secondarily the number of bends. These algorithms offer greater flexibility over the layered approach since the latter is solely focused upon the flow aesthetic criteria. If the number of line crossings is high or the hierarchy is well formed the layered approach may be advantageous. However, this is a small subset of class diagrams and thus UML-Kandinsky and GoVisual are a better fit with UML.

Gutwenger et al. [14] discuss a topology-shape-metric algorithm blending the aesthetic criteria required for hierarchical (generalization) and non-hierarchical (association) relations. The input to the algorithm is a hierarchical graph based on inheritance. The first step is to replace line crossings with dummy nodes. Next, upward planarization is applied to each hierarchy. This ensures that each relation in the hierarchy points in the same direction. In other words, the superclass for each generalization relation is above the corresponding subclass. To avoid overlapping hierarchies, each hierarchy is clustered together and treated as one node. Then upward planarization is run once more on the clustered graph. The final step is orthogonal drawing. Here GoVisual places each node and routes the relationship lines. This algorithm was implemented as GoVisual which is accessible from many languages with and API or available as a plugin for many commercially available layout packages such as Microsoft Visio.

Eichelberger also presented some visual guidelines and ran a pilot study to determine an effect of the guidelines on comprehension,. This is the only study that checks for comprehension besides prior work by Sharif et al. [[6], [23], [22], [24], [25]] However there are many differences in how Eichelberger's study is different. It is run on a toy application with no real system. Their results did not support the use of any layout rules they used. Their rules were based on general aesthetics and they do not consider semantic closeness of classes but focused solely on UML notation variants.

2.4 Class Diagram Layout Based on Architectural importance

Sharif et al. used both traditional based questionnaires and eye-tracking equipment to determine the comprehensibility of UML class diagram layouts based on architectural importance. Architectural importance was based on entity, control, and boundary stereotype information. The studies [[6], [23], [22], [24], [25]] compare traditional orthogonal layouts produced by state-of-the-art algorithms with the new proposed layouts based on architectural importance. The premise here is placing classes closer to each other actually reduces the cognitive load developers have to deal with when they read a UML class diagram. A set of guidelines were developed to aid in the development of an algorithm that realizes all the findings from these studies. [21]

2.5 Summary

The work presented in this thesis is based on the results derived from Section 2.4 above and expands on them by implementing an algorithm that realizes the guidelines developed. These guidelines are already empirically validated and have shown to be better than the traditional orthogonal layouts as mentioned in Section 2.3. The algorithm presented in this thesis starts with a previously drawn layout and adjusts it based on user options and architectural importance. The algorithm design and implementation is the main contributing factor of this thesis. The previous layouts were taken from [21].

Chapter 3

The Proposed Algorithm

This chapter presents details on the algorithm proposed in the thesis. Each step is explained in a separate section.

3.1 A Force Based Approach to Layout

This section presents the approach used in the algorithm presented in this thesis. A step by step description of the algorithm and calculations involved are given. Force directed layout algorithms use physics to position nodes in a graph. They work by assigning a force to each edge of the graph and simulating the system as if it were a physical system. These forces are proportional to the graph theoretic distance between points. Previous methods use either springs or electrical charges [8]. The algorithm combines these approaches with the additional information about the edges provided by the class diagrams. This section will walk, step by step through the algorithm. A flowchart of the algorithm has been provided as a map. See Figure 3.1.

3.2 Initialization

The green section in the flow chart is initialization. This section includes one input and two processing steps. The input file (config file) contains all options and parameters specified by the user (see Appendix A for more details). There are a handful of options and parameters used to control various aspects of the algorithm. The config file also includes the name of the data file and output directory.

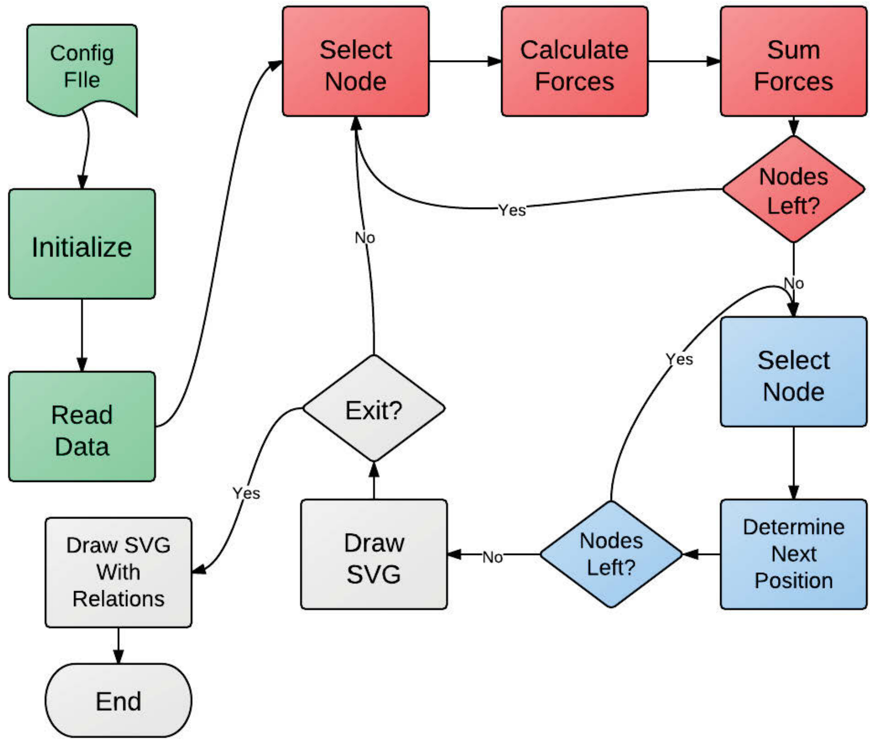


Figure 3.1: Flow Chart describing the steps in algorithm

The first step in initialization, and the algorithm, is reading the configuration file. The configuration file is formatted as a plain text key-value pair system. Each key corresponds to an option in the system. As the configuration file is parsed, the value in each pair is stored in a corresponding variable. If one of the optional parameters is not specified a default value is used. If an unknown, most likely misspelled, option is present in the config file the program displays an error message and terminates.

Once the configuration file is successfully parsed the second processing step - reading data - begins. The data file name is specified in the config file with ”data”. There are currently two formats for data. The first is a simple key-value store similar to the config file. The second, and feature complete, format is ClassML. The next sections describes ClassML.

3.3 ClassML

ClassML is based on XML and includes four container elements; class, relation, attribute, and method. The method element holds all information pertaining to class methods - name, return type, visibility, and parameters - and is stored inside of the class element. The same applies to the attribute element. The class element is the structure which holds all information about a class including methods, attributes, and, optionally, position. The relationship element is the final container element and stores the name, type, start class, and end class. See Figure 3.2 for a UML diagram depicting classML.

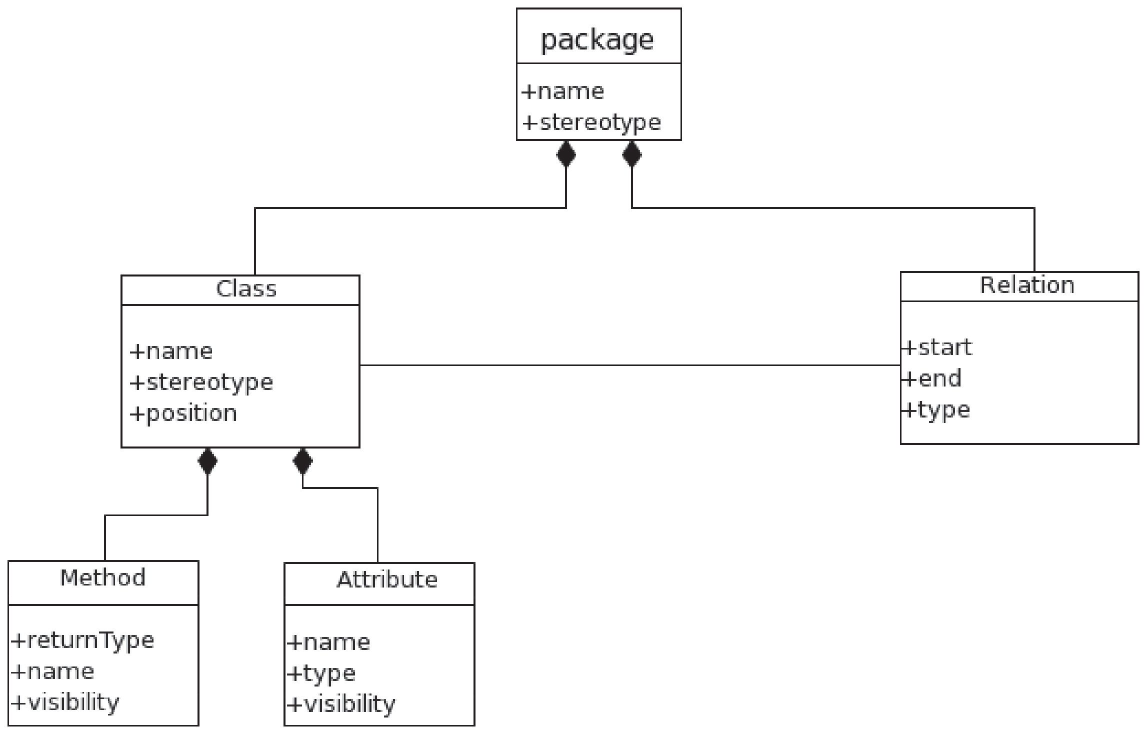


Figure 3.2: A class diagram showing the main elements of ClassML

See Appendix B for an example of a classML file.

3.4 Calculate Forces

The next section in the algorithm is calculating the forces, attractive and repulsive, applied to each node. It is the red portion of the flow chart. This section is also the first portion of the main loop. The forces chosen combine previous work with gravitational and spring systems.

3.4.1 Attractive Force

In order to introduce clustering an attractive force is needed. Two concepts from physics come to mind when thinking of attractive forces; gravity and electricity. Both gravity and electricity have similar equations, thus forming the base for the model.

$$F_g = G\frac{m_1 m_2}{r^2}$$

$$F_e = k\frac{q_1 q_2}{r^2}$$

There are three important features of each equation; universal constant, local values, and inverse square distance. The two types of constants allow for scaling the force and the inverse square of the distance determines the force's magnitude.

The universal constant, G or k, allows global scaling of the force. This way the algorithm can emphasize a specific type of relation and create a diagram that is more useful to the user.

Local values - m_1, m_2 - provide a method for distinction between nodes, used as a mathematical model for the classes. Local constants are also used to fix the position of certain nodes, therefore eliminating the possibility of infinite translation.

The attractive force is proportional to the inverse square of distance between the two nodes. Distance is calculated as the Euclidian distance.

$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

$$F_a = k_a\frac{m_1 m_2}{d^2}$$

Squaring the distance, as opposed to linear, increases the effect distance causes on the force. Inversely proportional means that as the distance between nodes decreases the force increases. So, nodes that are distant have little effect on each other. Yet, nodes that are close strongly attract.

When looking at a class diagram sometimes certain relations are more important than others.

With the algorithm it is possible to distinguish between relationship types by specifying a different k_a value for each type.

A UML diagram is defined as a directed graph. The goal of the algorithm is to group related classes. In order to accomplish this goal, the algorithm treats all edges as undirected edges for force calculation. The directed nature is preserved only to accurately draw the relations in the final diagram.

Suppose that the attractive force were the only force applied to every pair of nodes. The final result would be one pile of nodes at the center of the diagram. Not a usable, let alone improved, diagram. Thus only a subset of pairs is be used. The most obvious, and implemented, selection is only apply the force between pairs of nodes defined by a relation. With this subset of forces the final graph will be multiple clusters of related nodes. This is the reason clustering occurs.

3.4.2 Repulsive Force

At this point all of the nodes are clustered based on relationships. However, all of the clusters are piled in one location making for an unreadable diagram. To solve this problem some form of collision handling is required. A repulsive force is added to counterbalance the attractive force. This force is modeled after Hooke's Law for springs, linearly proportional to the Euclidian distance between nodes.

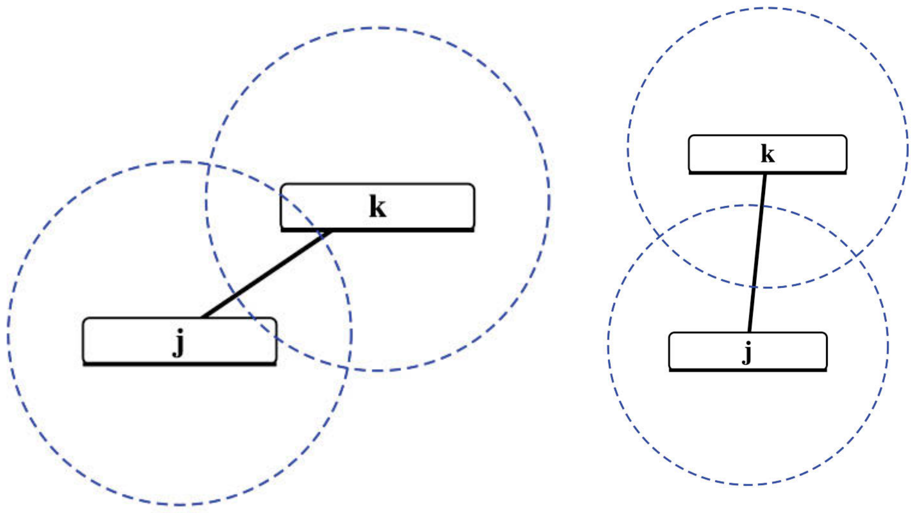
$$F_r = -k_r\sqrt{\Delta x^2 + \Delta y^2}$$

The proportionality constant, k_r, is specified in the configuration. A value should be chosen chosen so that at small distances $F_r > F_a$ and at large distances $F_r < F_a$. This choice creates an equilibrium in which no classes are overlapping.

3.4.3 Bounding Boxes

Up to this point all nodes were treated as singular points. In the diagram each node requires a certain amount of space. To account for this requirement bounding boxes are used to expand the discrete points. The radius, b_i, of the circle which circumscribes the node is calculated. This radius is used as the cutoff point for the forces. If the distance between two nodes is less than the sum of their radii the attractive force is no longer calculated and the repulsive force activates. The opposite is also true.

One of the optional parameters in the config file is the amount of padding to place between nodes. This padding, p, is represented as a multiple of the node radius, i_{radius}. If this option is specified padding times radius is used as the cutoff point.



(a) Repulsive On, Attractive Off

(b) Repulsive Off, Attractive On

Figure 3.3: Bounding Boxes Illustration

Figure 3.3 shows the two states in which a pair of classes exists. When the bounding box overlaps with the class drawing, (a), the repulsive force is applied while the attractive force is turned off. When the overlapping no longer occurs the attractive force is applied and the repulsive force is no longer calculated.

3.5 Final Version of Equations

$$b_i = p i_{radius}$$

$$d_{ij} = \sqrt{x_i - x_j{}^2 + y_i - y_j{}^2}$$

$$F_{a,ij} = k_a \frac{m_i m_j}{d_{ij}^2},\ d_{ij} > b_i + b_j$$

$$F_{r,ij} = -k_r d,\ d_{ij} < b_i + b_j$$

3.6 Update Positions

The blue section of the algorithm (See Figure 3.1) designates the steps for updating positions of nodes. Here the previously calculated forces are applied and nodes moved to their next location. The key equations in this section are Newton's Second Law and displacement.

$$F = ma$$

$$\Delta p = vt + \frac{1}{2}at^2$$

Some assumptions are applied to the displacement equation to improve algorithm performance. First, time (t) is set to one. Time is a scaling factor concerning how far each object moves each iteration. We have other variables that serve the same purpose so this assumption simplifies our calculation. Second, velocity (v) is set to zero. This may seem like an odd approach to take, but it is helpful in avoiding some potential pitfalls discussed in the Algorithm Completion section. This is a valid approach since we are not concerned with simulating the real world.

So after our assumptions are applied the final equations for calculating the next position are

$$\Delta x_i = \frac{F_{x,i}}{2m_i}$$

$$\Delta y_i = \frac{F_{y,i}}{2m_i}$$

The same process is applied to each node to obtain an updated graph.

3.7 Draw Results

The final section of the algorithm is draw results and corresponds to the silver section of the flow chart. In this step an SVG[2] image of each class and relation at their respective current locations is drawn.

One of the options in the config file - fpd - allows the user to specify the number of frames per drawing. If this parameter is set, an image will only be drawn if the current iteration is a multiple of this value. This way a user can limit the number of intermediate images, and consequently reducing the disk space used.

The flow chart depicts two different SVG drawing steps, Draw SVG and Draw SVG with Relations. Both steps draw each class at their respective location and lines connecting related classes. Draw SVG uses solid straight lines to connect classes, while With Relations uses arrows depending on relation type. The use of solid straight lines serves two purposes, speed and comprehension. Drawing relation type specific arrows requires more calculation as well as more disk space, thus solid lines are faster and more efficient. Solid lines also provide a graphical representation of the direction that the forces will be applied. This increases comprehension when the output is viewed as an animation. To provide further comprehension the bounding boxes are also drawn during the intermediate iterations.

Exiting is another important function of the draw results section. On each iteration, regardless of fdp value, draw results checks if any of the exit conditions has been met. If so, the diagram is drawn a final time without bounding boxes and relations drawn as specified by UML instead of plain lines.

3.8 Algorithm Completion

Ideally, during an iteration all nodes remain stationary. When this occurs any subsequent iterations of the main loop will also result with no movement. Allowing the algorithm to continue will waste CPU cycles, so the main loop is exited and the final result is written to disk. This is the first - and preferred - exit condition, stationary termination.

A rigorous mathematical proof of the algorithm is impossible due to the three body problem from physics. The three body problem arises numerous places in physics, and perhaps the most famous comes from the motion of Earth, the Moon, and the Sun. In the three body problem each object is given a mass, position, and velocity. From here the goal is to determine the trajectory each body will follow. If the system is restricted to two bodies it is possible to solve the system of equations analytically. Once a third body is introduced infinite geometric series are required to calculate the trajectories. This allows for unstable trajectories to exist - an unsatisfactory state for the system. The three body problem is generalizable to the n-body problem. The impact of the three body problem is reduced by resetting velocity.

These unsatisfactory states will never reach a static equilibrium and alternative exit conditions must be designed. Rotation is one scenario in which static equilibrium will never occur. A diagram under rotation is typically readable and an improvement over the original diagram. Theoretically, once a diagram reaches a rotational state any iteration will produce a satisfactory image. A possible exit condition would be to detect rotation. However, detecting rotation is computationally intensive and the state is uncommon (based on empirical observation) thus it is not worth the expense.

Another possible unsatisfactory state is clumping - a scenario where multiple nodes are stuck together near the middle of the diagram. With clumping, the diagram would be unreadable. See Figure 3.4.

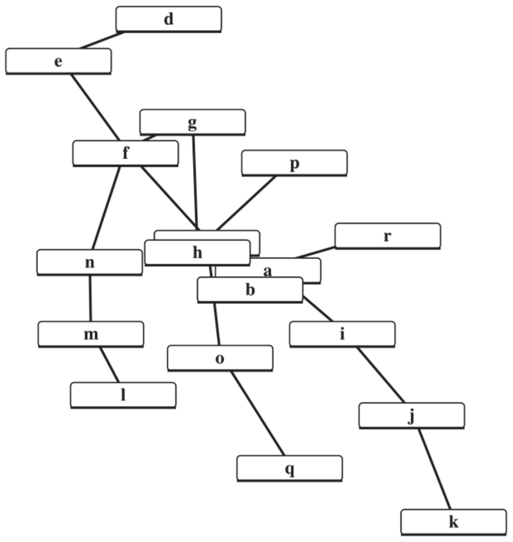


Figure 3.4: Example of Clumping

Therefore we must design a mechanism to handle this condition. One possibility would be to apply a large force, in differing directions, to each of the overlapping nodes then continue to run the algorithm. To continue our physics analog, this is akin to detonating an explosive charge at the centroid of the overlapping nodes. While this method solves the problem it adds unnecessary complexity to the algorithm.

Detecting and resolving both of these issues is possible with one intriguingly simple mechanism. Randomize all of the node positions if clumping is detected. The goal with randomization is that the new initial conditions will produce a state in which static equilibrium is obtainable. According to testing, the number of iterations to completion for a collection of randomly placed nodes follows the gamma distribution (See Appendix C for statistical analysis of randomize). One of the defining characteristics of the gamma distribution is the concentration of probability near the origin. Therefore it is likely that randomization will result with a system in static equilibrium. Randomization is triggered after a number of iterations - specified in the config file with random - occur without obtaining static equilibrium.

If all else fails, a threshold is set to terminate the algorithm after a given number of iterations have run. This failsafe is the final exit condition.

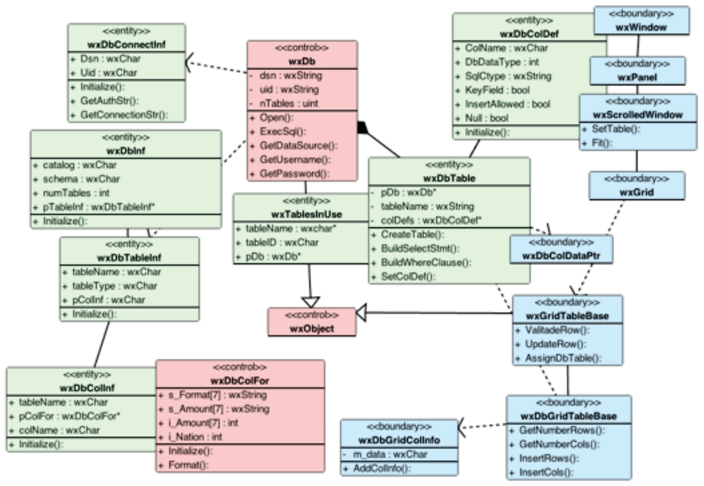
Chapter 4

Results

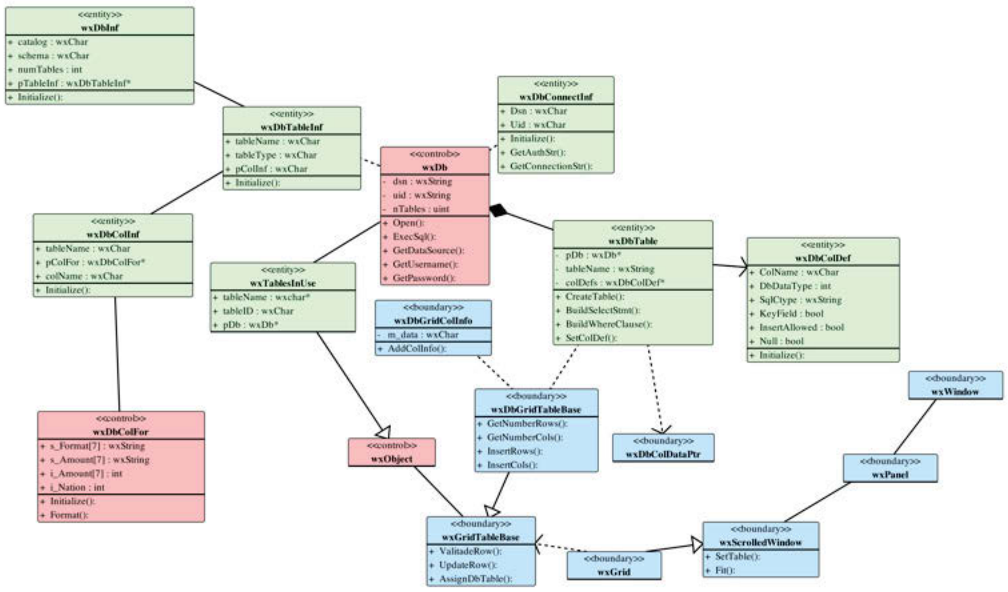
wxWidgets[4] and Qt[1] are C++ open-source GUI frameworks that allow developers to create cross platform applications. The database portion of both libraries was used as input to test the algorithm. With both systems the algorithm placed semantically related classes closer together taking into consideration the user-defined criteria. This has been empirically shown to improve diagram comprehension [21]. The input diagrams were reverse engineered directly from source and were taken from [21].

The next two sections show a before and after look at each diagram from both packages. The same config file (Appendix A) was use for each system.

4.1 wxWidgets



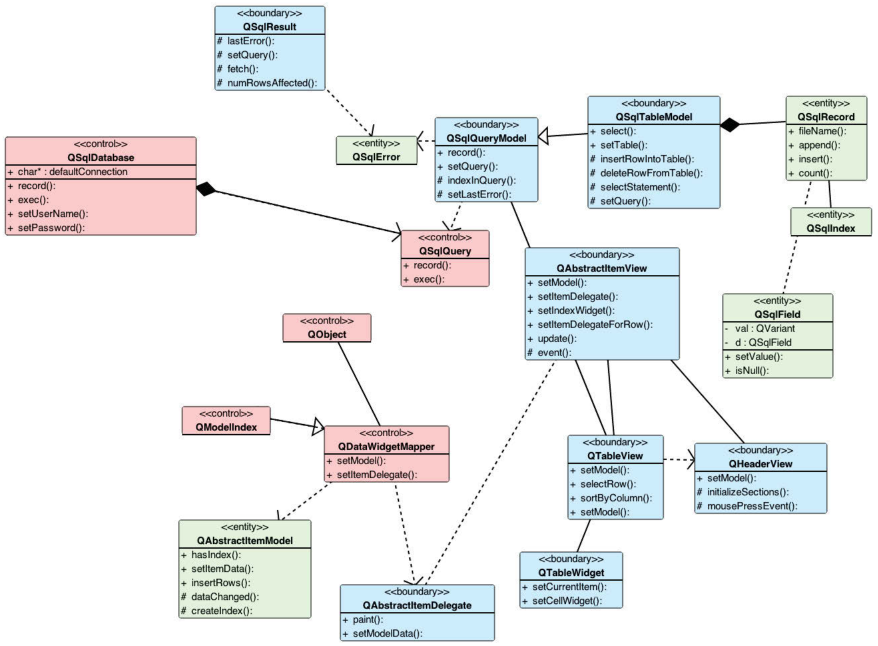
(a) Input



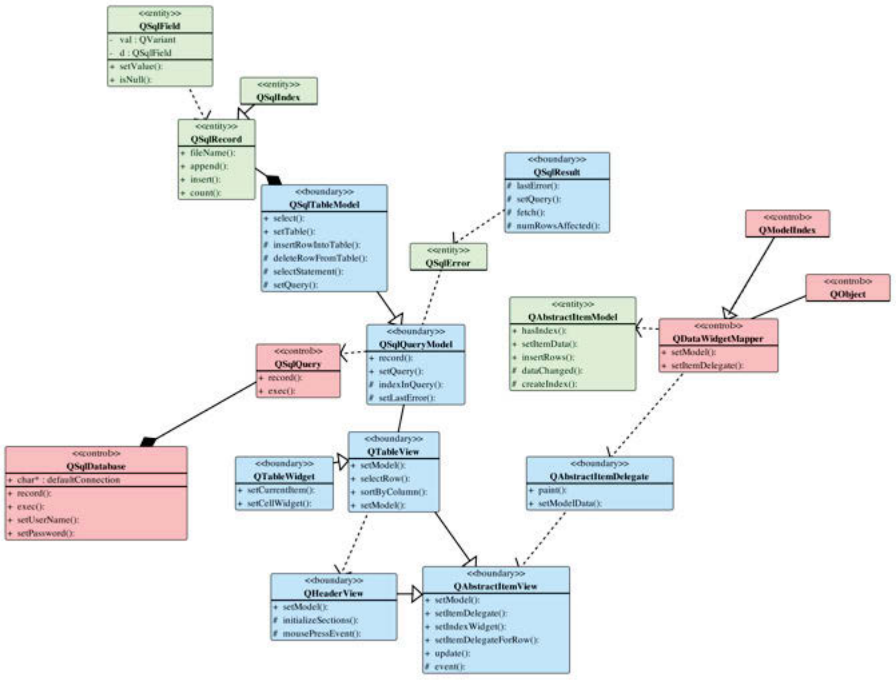
(b) Output

Figure 4.1: WxWidgets Before and After (314 iterations)

4.2 Qt



(a) Input



(b) Output

Figure 4.2: Qt Before and After (2,163 iterations)

4.3 Layouts Optimized for Design Patterns

This section discusses how layout can be modified to highlight design patterns in class diagrams. Classes that are typically not shown near each other in traditional layouts are placed closer due to the semantic meaning they have e.g., participating in a design pattern. First, the general idea of a design pattern is presented and an example is then given with respect to the strategy pattern.

Design patterns [13] are useful abstractions in Object Oriented software design. They represent recurring solutions to problems in specific contexts. One specific design pattern is called the Strategy pattern. Strategy is used when multiple algorithms exist for a specific function, like in sorting. Here the Strategy pattern hides the specific details of sorting from other objects using the Strategy. This allows the program to specify which algorithm is used dynamically on run time conditions in the program. The Strategy pattern has three roles associated with it: Context, Strategy and Concrete Strategy.

A class diagram depicting the Strategy pattern is shown in Figure 4.3.

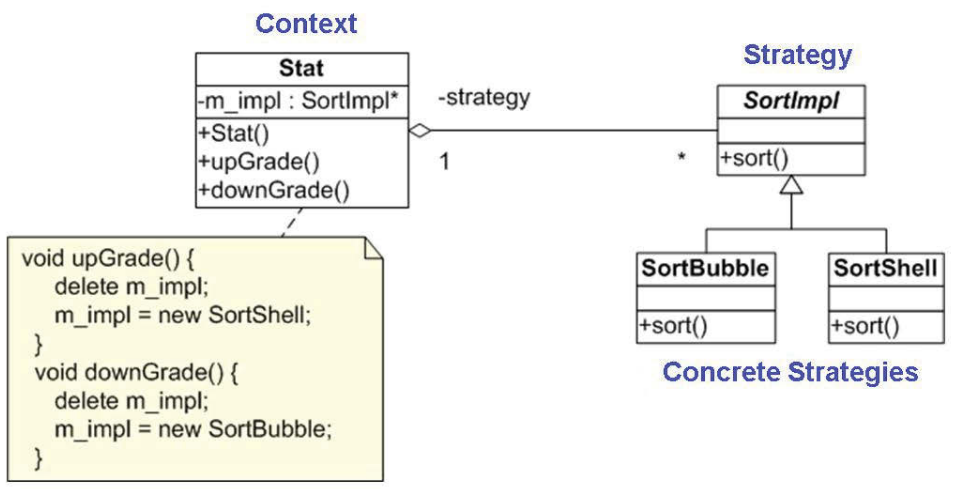


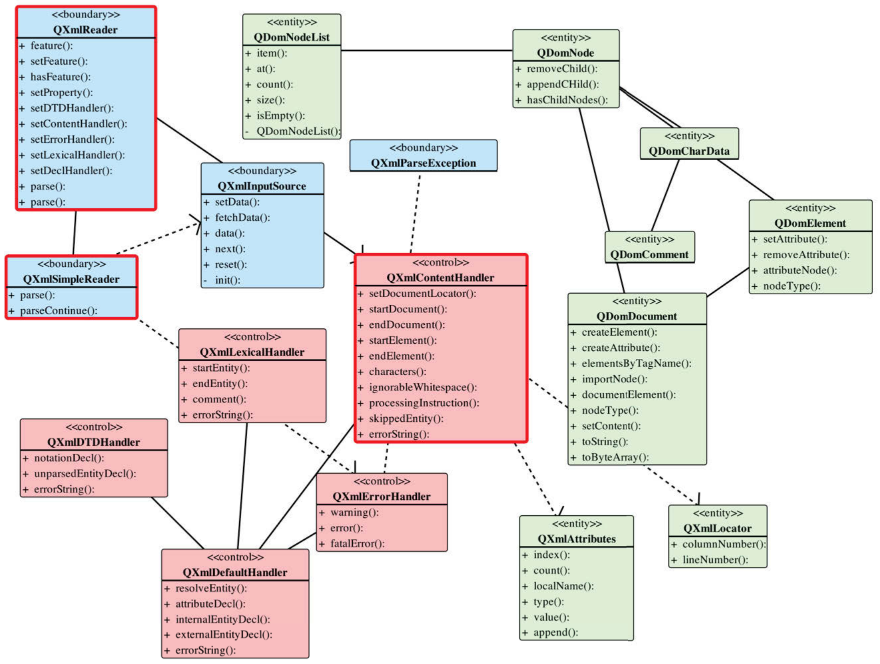
Figure 4.3: Example Strategy Pattern

In this example, the Stat class plays the role of Context, the SortImpl class plays the role of Strategy, and the SortBubble and SortShell classes play the role of Concrete Strategies.

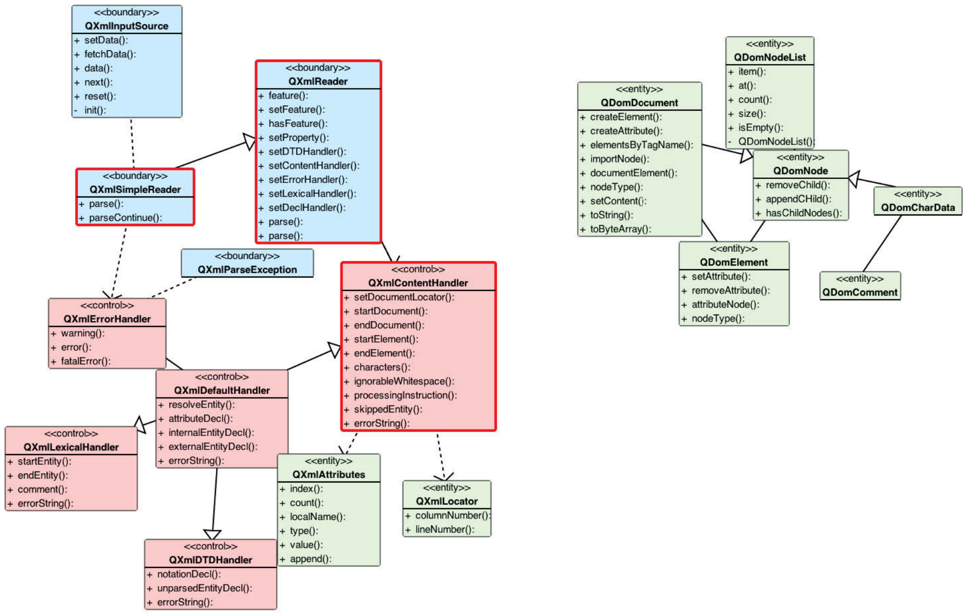
Many systems employ the strategy pattern including Qt. In the example there are three classes involved in the strategy pattern, highlighted with a red border. The class QXmlContentHandler plays the role of Context, the class QXmlReader plays the role of Strategy, and in this case there is only one concrete strategy class QXmlSimpleReader.

Using an orthogonal layout obscures the nature of this pattern. Figure 4.4 shows the results. In the output diagram classes participating in the strategy pattern are shown closer and in a pattern that is similar to it's canonical form shown in Figure 4.3.

The red border was overlaid on top of the classes to show which classes participate in the Strategy design pattern.



(a) Input



(b) Output

Figure 4.4: Strategy Pattern in Qt Before and After (257 iterations).

Chapter 5

Conclusions and Future Work

The thesis presents a layout adjustment algorithm and its corresponding implementation for class diagrams that takes into account semantic information between classes. It takes a force-based approach in implementing the algorithm. The guidelines used as input to the algorithm have been previously validated extensively on real systems and real software tasks [21]. A ClassML format was designed and a parser was written to input details into the algorithm. A configuration file was also created that is easily expandable in the future. The algorithm in it's current state is far from done. It is viewed as the first step in realizing the guidelines. Future work is discussed next.

There is still a lot of space for improvement. By focusing on positioning the classes, the implementation only draws the relations as straight lines. Many users prefer curved or orthogonal lines connecting the classes. This can be included with the implementation. Currently, in generalizations, the superclass is not necessarily placed on top of the subclass. Even though this violates UML layout aesthetics, it needs to be seen whether this is truly a hindrance in understanding.

The configuration file is cryptic and can only be understood in conjunction with the table of options. Defining an xml format could solve this problem. Using command line parameters in combination with the configuration file is another possibility. This problem can be hidden with a nice user interface. A well designed graphical user interface (GUI) would reduce friction and make the algorithm more accessible to the user. Ideally the GUI would allow a user to specify their data file using the system file browser and push one button to generate a better layout. An advanced menu to adjust the other options in the config file would be an added bonus.

The GUI should also include an interface for manipulating the diagram based on design patterns.

Each pattern contains certain types of relations. To search for a given pattern the attractive force constant corresponding to these relation types would be increased. A method for averaging over multiple design patters should also be explored.

To further increase usability, a plugin for Eclipse or Visio is envisioned in the near future. This plugin would integrate the algorithm into the user's environment thereby removing the need for an additional program.

The algorithm's performance can be improved. Selecting a different physical model, e.g. annealing, or modifying the force functions may increase performance but empirical validation is necessary. Computing each connected components in parallel and composing the final images would provide a decent speed up.

Appendix A

Configuration File

command	*variable type*	*default*	*description*
ak[acdgs]	double	3000	Value of the attractive constant. The optional parameters correspond to relationship type. If not specified the constant is used for all types.
css	string	Write custom css to out/svg.css	CSS file (file path) containing styles for images
data	string	*null*	File containing classes and relations. Must have xml or dat extension.
fpd	positive integer	10	Number of frames to skip per drawing.
iter	positive integer	2000	Maximum number of iterations before aborting.
out	string	res/	Directory to store results
restart	integer	-1	Number of iterations before triggering randomization mechanism.
rk	real number	0.1	Value of the repulsive constant.

Table A.1: List of possible configuration file commands.

```
data data/qt_sql.xml
out res/
iter 100000
fpd 100
ak 3450
akg 3450
akd 3450
aka 3450
rk 0.175
```

Figure A.1: Example Configuration File

Appendix B

ClassML Data File

```
<classML>
 <class>
  <name>QSqlIndex</name>
  <stereotype>entity</stereotype>
  <position>950,180</position>
 </class>
  ...
 <class>
  <name>QSqlField</name>
  <stereotype>entity</stereotype>
  <position>910,270</position>
  <att>
   <type>QVariant</type>
   <name>val</name>
   <visibility>-</visibility>
  </att>
  <att>
   ...
  </att>
  <meth>
   <visibility>+</visibility>
   <name>setValue</name>
  </meth>
   ...
 </class>
 <relation>
  <start>QTableView</start>
  <end>QAbstractItemView</end>
  <type>generalization</type>
 </relation>
</classML>
```

Figure B.1: Example ClassML File

Appendix C

Statistical Analysis of Randomize

In order to begin analyzing the viability of randomize data needed to be generated. These inputs should have varying numbers of classes and relationships. Each data set should also have random starting positions. After looking at existing diagrams, nine to eighteen appeared to be a reasonable number of classes to test. Five data sets were created, each adding extra classes and relations to the previous set, starting with nine and working up to eighteen.

Each input was run one thousand times with different random positions. The number of iterations to completion was recorded for every run. Iterations to completion was chosen since it is less dependent on implementation and system specifications.

The first step of analyzing the data is to plot the density for each run. This was accomplished with an R script.

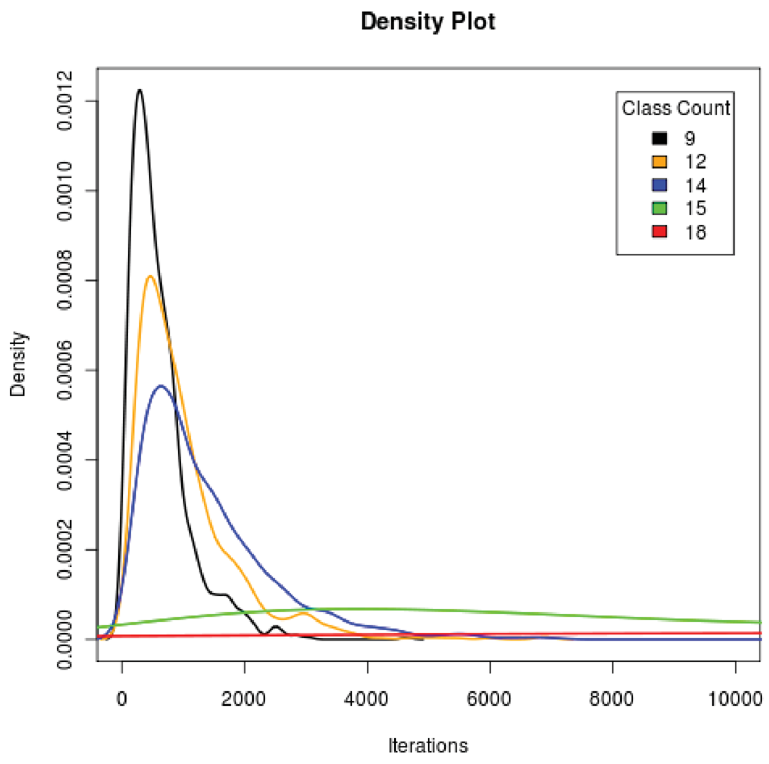


Figure C.1: Experimental Density

Looking at these curves, a pattern emerges. Curves starting near the origin with a peak shortly after, and exponentially decaying. This implies an underlying relationship from the exponential family and it is possible to find the type of distribution that models the system. The next step is to estimate parameters for specific distributions, then calculate the goodness of the fit.

The method used for estimating parameters is the Method of Moments. With this method we estimate the parameters as functions of the moments. In this case we are looking at the exponential family so only the first two moments, mean (μ) and variance (σ^2), will be necessary.

Three distributions will be explored - normal, gamma, and exponential. These continuous functions were chosen despite the data being discrete. Since the scale is large continuous is a valid approximation.

Estimating parameters for the normal distribution is straight forward. Calculate the mean and variance then plug into equation.

$$f_{normal}(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$$

The gamma distribution is characterized by shape ($\alpha > 0$) and rate ($\beta > 0$) parameters. In order to estimate these parameters an equation relating variance and mean must be derived for each.

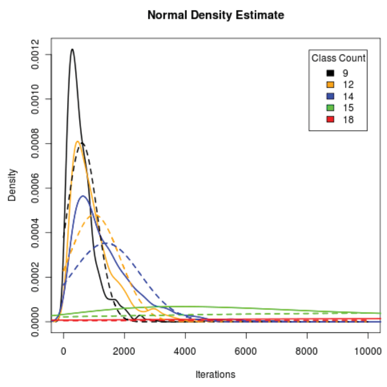
$$\alpha = \frac{\mu^2}{\sigma}$$

$$\beta = \frac{\mu}{\sigma}$$

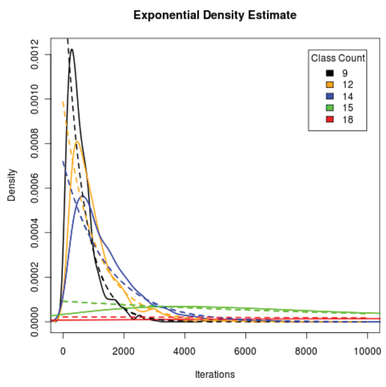
$$f_{gamma}(x; \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

The exponential distribution only has one parameter, $\lambda = \frac{1}{\mu}$, and is described below. $H(x)$ is the Heaviside step function, the integral of the Dirac Delta function.

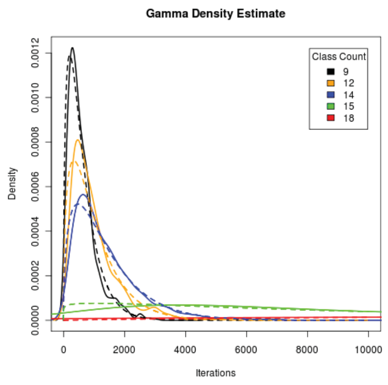
$$f_{exponential}(x; \lambda) = \lambda e^{-\lambda x} H(x)$$



(a) Normal



(b) Exponential



(c) Gamma

Figure C.2: Density plots with estimated parameters

Now that we have estimated the parameters it is time to determine which distribution has the best fit. There are many methods for measuring the goodness of fit and the Kolmogorov-Smirnov test was chosen for this system. After running the test the gamma distribution has the best p-value, followed by the exponential distribution, and the normal in dead last. Therefore it is reasonable to conclude that the data follows the gamma distribution.

It would be interesting to search for a relation between the estimated parameters and relation count, or even k_a and k_r.

Bibliography

[1] Qt website. `http://qt.nokia.com/products/`, 28 April 2012.

[2] Svg website. `http://www.w3.org/TR/SVG/Overview.html`, April 2012.

[3] Uml website. `http://www.uml.org/`, 29 April 2012.

[4] wxwidgets website. `http://www.wxwidgets.org/`, 27 April 2012.

[5] Ambler, S. W. *The Elements of UML Style*. Cambridge University Press, 2002.

[6] Andriyevska, O., Dragan, N., Simoes, B., and Maletic, J. Evaluating uml class diagram layout based on architectural importance. In *Proceedings of 3rd IEEE International Workship on Visualizing Software for Understanding and analysis* (2005), pp. 14–19.

[7] Booch, G., Rumbah, J., and Jacobson, I. *Unified Modeling Language User Guide*, 2nd ed. Addison Wesley, 2005.

[8] Di Battista, G., Eades, P., Tamassia, R., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[9] Eichelberger, H. *Aesthetics and Automatic Layout of UML Class Diagrams*. PhD thesis, Universitat Wuerzburg, 2005.

[10] Eichelberger, H., and Schmid, K. Guidelines on the aesthetic quality of uml class diagrams. *Information and Software Technology* (2009).

[11] Eichelberger, H., and von Gudenberg, J. W. Uml class diagrams - state of the art in layout techniques.

[12] Eiglsperger, M. Automatic layout of uml class diagrams in orthogonal style. In *in Proceedings of Information Visualization* (2004), p. 2004.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[14] Gutwenger, C., Junger, M., Klien, K., Kupke, J., Leipert, S., and Mutzel, P. A new approach for visualizing uml class diagrams. *Software Visualization* (2003).

[15] Huang, X., Sajeev, A. S. M., and Lai, W. A scalable algorithm for adjusting node-node overlaps. In *International Conference on Computer Graphics, Imaging, and Visualization* (2006).

[16] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, 1999.

[17] Purchase, H. Performance of layout algorithms: Comprehension, not computation. *Journal of Visual Languages and Computing 9*, 647-657 (1998).

[18] Purchase, H. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing 13* (2002), 501–516.

[19] Purchase, H., and Samara, A. Extremes are better: Investigating mental map preservation in dynamic graphs. 60–73.

[20] Seemann, J. Extending the sugiyama algorithm for drawing uml class diagrams: Towards automatic layout of object-oriented software diagrams. *GD* (1997).

[21] Sharif, B. *Empirical Assesment of UML Class Diagram Layouts Based on Architectural Importance*. PhD thesis, Kent State University, 2010.

[22] Sharif, B., and Maletic, J. The effect of layout on the comprehension of uml class diagrams: A controlled experiment. *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis* (2009).

[23] Sharif, B., and Maletic, J. An empirial study on the comprehension of stereotyped uml class diagram layouts. *17th IEEE International Conference on Program Comprehension* (2009).

[24] Sharif, B., and Maletic, J. The effects of layout on detecting the role of design patterns. *23rd IEEE-CS International Conference on Software Engineering Education and Training* (2010).

[25] Sharif, B., and Maletic, J. An eye tracking study on the effects of layout in understanding the role of design patterns. *26th IEEE International Conference on Software Maintenance* (2010).

[26] Sugiyama, K., Tagawa, S., and Toda, M. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2 (Feb 1981), 109–125.

[27] Ware, C., Purchase, H., Colpoys, L., and McGill, M. Cognitive measurements of graph aesthetics. *Information Visualization 1*, 2 (2002), 103–110.