GGraph: Uma ferramenta para aplicações que envolvem grafos.

Luiz Carlos Lucca

GGraph: Uma ferramenta para aplicações que envolvem grafos.

Luiz Carlos Lucca

Orientador: **Prof. Dr. Antonio Castelo Filho**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Janeiro de 2013

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

L934g

Lucca, Luiz Carlos
GGraph: Uma ferramenta para aplicações que envolvem grafos. / Luiz Carlos Lucca; orientador Antonio Castelo Filho. -- São Carlos, 2012.
89 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2012.

1. Manipulação de Grafos. 2. Visualização de Grafos. 3. Ferramenta de Grafos. I. Filho, Antonio Castelo, orient. II. Título.

Agradecimentos

Acima de tudo, gostaria de agradecer a Deus, quem me guia e fortalece, por ser fiel e estar sempre ao meu lado durante todo este período, sendo abrigo sempre que preciso e dando forças para seguir adiante. A Ele seja dada toda a honra, a glória e o louvor.

Agradeço à minha namorada Daiana, que me apoiou na decisão de concorrer ao programa de mestrado, me incentivou durante todos esses anos, parte deles à distância, tendo por vezes abdicado de seu tempo em detrimento desse trabalho. Seu amor, seu carinho e sua compreensão são motivadores em tudo o que eu faço e foram decisivos para que os resultados deste trabalho fossem alçados.

A toda a minha família, e em especial aos meus pais Luiz e Leonice, pelo amor, pela paciência, pela formação como pessoa e pelo apoio de sempre, principalmente durante os anos em que vivi em São Carlos.

Ao Prof. Dr. Antonio Castelo Filho, por acreditar em mim, pela atenção e pelo apoio durante o processo de definição e orientação.

À Universidade de São Paulo, onde construí minha vida acadêmica, pela oportunidade de realização do curso de mestrado.

Aos meus amigos, que sempre me apoiaram no percurso do mestrado. Entre eles, gostaria de agradecer em especial aos meus amigos Erick Mazieiro, Ricardo Wandré, Thiago Gotti, Leandro Arrivetti, Ricardo Kramer, Caio Kramer, Tiago Augusto, Victor Dutra, Diego Iritani, Allan Pugliese, Madjer Oliveira, Isaac Oliveira, Luiz Henrique, Luiz Eduardo, Thiago Galvão, Marcos Vieira e Rafael Heleno que, além do contínuo apoio, ajudaram ativamente na conclusão deste trabalho.

Resumo

LUCCA, L. C. **GGraph: Uma ferramenta para aplicações que envolvem grafos.** 2012. 97 p. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, 2012.

Diversas são as aplicações que podem ser expressas por meio de grafos [2]. Algoritmos [3] e modelos de visualização [15] podem ser encontrados amplamente na literatura. Todos os problemas de grafos possuem uma base em comum: um modelo genérico que nasce da própria natureza dos elementos e das relações que podem ser expressas entre eles, diferindo apenas pelo tipo de resposta que queremos obter desta complexa malha. Além disso, é natural que, para problemas que sejam de áreas distintas, mas que sejam semelhantes quanto ao processamento interno, apenas o que mude, seja a visualização dos elementos que o compõe (nós, arestas, etc.). Da mesma forma, independente do tipo de processamento interno, os grafos devem manter a estrutura original de grafos, ou seja, ainda deve haver uma malha que descreve os nós e suas ligações. Neste aspecto, fundamentamos nosso estudo: propomos neste trabalho, desenvolver uma API que possa ser estendida para os mais diversos problemas na área de grafos, tanto na parte visual como na representação matemática do modelo e dos algoritmos, porém, robusta, no sentido de manter a complexidade dos algoritmos envolvidos na área de grafos, além de ser completamente dirigida as necessidades de cada aplicação, podendo-se alterar apenas algumas partes da aplicação para obter um produto específico ao trabalho do usuário.

Abstract

LUCCA, L. C. **GGraph: Uma ferramenta para aplicações que envolvem grafos.** 2012. 97 p. Dissertação (Mestrado) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, 2012.

There are several applications that can be expressed by means of graphs [2]. Algorithms [3] and visualization models [15] can be widely found in the literature. All graph problems have a common base: create a generic model that arises not only from the nature of their elements, but also from the relationships which these elements can express, differing just by the type of response we want to get from this complex mesh. Moreover, it is natural for problems that are in different fields, but similar in internal processing, that the only change is related to how elements are visualized (nodes, edges, and so on). Likewise, regardless the internal processing, the graphs must keep their original structure, i.e., they must still be a mesh that describes the nodes and their connections. Based on that, this study proposes to develop an API that is generic enough to be extended to several problems in the graphs area. This API can be applied in both visual and mathematical representation of models and algorithms. Besides that, it must be robust to maintain the complexity of the algorithms involved in the graph. Also, it has to be flexible so that only some parts of the application can be changed to get a specific product to the user´s need.

Lista de Figuras

Listagens

Sumário

1. Introdução

Muitas são as aplicações que necessitam uma representação por grafos, seja para otimização ou para simples visualização. Com o desenvolvimento da teoria de grafos, estruturas de dados, da área de HCI (Human Computer Interaction) e do próprio poder computacional, vários problemas puderam ser resolvidos devido a maior interatividade com o usuário e a maior velocidade de processamento dos computadores: grandes sistemas podem ser modelados de modo que as relações expressas pelos elementos envolvidos são completamente exploradas em tempos aceitáveis de processamento. Como sabemos, vários problemas relacionados a grafos pertencem à classe NP-Completo; isso não mudou com o desenvolvimento dos computadores, porém, podemos explorar soluções que sejam viáveis em menor tempo, e assim, desenvolver boas heurísticas para aproximar o resultado da solução ideal do problema.

O desenvolvimento de técnicas de manipulação de grafos e, da própria área de visualização, permitiu que novos paradigmas de representação surgissem. Bons exemplos de plataformas que manipulam este conjunto de dados podem ser encontrados sem nenhuma dificuldade pela internet; como exemplo, podemos citar a PEX (Projection Explorer) [18], que possui várias técnicas de visualização e clusterização implementadas. Além disso, novas ferramentas e bibliotecas (como Walrus e JGraphT) permitem a exploração e desenvolvimento de grafos sem exigir muito do usuário ou programador.

Apesar do amplo conjunto de bibliotecas e ferramentas disponíveis no mercado (livres ou comerciais), nenhuma é tão robusta ao ponto de poder representar uma ampla gama de aplicações apenas mudando um núcleo genérico.

Neste contexto, desenvolvemos nossa pesquisa, propondo uma ferramenta que seja capaz de representar diversas aplicações, mudando apenas um núcleo genérico, sem aumentar o custo computacional que é exigido pelos algoritmos que serão empregados. O projeto foi feito utilizando o NetBeans, uma plataforma que fornece diversas ferramentas para desenvolvimento de código, suportando Java, C, C++ e outros; além de poder organizar o projeto logicamente em pastas, subpastas, etc., ele

permite que o grupo desenvolva no mesmo padrão, podendo compartilhar o projeto, suportando também integração com o *subversion,* ferramenta para controle de versão do código.

1.1 Organizaçao do Trabalho

O trabalho está organizado da seguinte forma: **revisão bibliográfica**, onde um breve estudo de grafos e algoritmos é apresentado; **desenvolvimento do projeto**, onde as idéias centrais, assim como um minucioso estudo do desenvolvimento é discutido; **MVC** (Model – View – Controller), que além de discutir o padrão de projeto adotado, mostra como foi desenvolvido no projeto; **Interface gráfica**, em que discutimos o conceito e suas implicações para um bom projeto, além de descrever como ela foi implementada no projeto**; adaptação do código para o Hydra**, onde descrevemos a adaptação das classes base para o projeto Hydra; **Anexos e Referências**, onde apresentamos o UML das classes críticas, e as referencias da literatura utilizadas.

O motivo desta organização foi levar o leitor a entender os conceitos básicos das técnicas que serão empregadas no desenvolvimento do projeto, introduzindo e relembrando algumas definições que serão empregadas no decorrer do documento. Em cada seção, o leitor poderá encontrar as referências utilizadas, identificadas por um número que pode ser pesquisada na área de Referências ao final do texto.

Após a introdução dos conceitos básicos, a seção de objetivos descreve qual o objetivo principal do projeto, além de citar algumas referências a bons paradigmas na área de visualização e interatividade com grafos.

2. Revisao Bibliográfica

2.1 Grafos

Diversas são as áreas da ciência que se depara com problemas onde existem relações diretas e implícitas entre elementos de um domínio específico. Essas relações podem ser descritas, muitas vezes, matematicamente como um grafo *G*, ou seja, um conjunto não vazio de elementos, chamados de vértices *V* e um conjunto de pares não ordenados de vértices, chamado arestas *E*: *G(V, E)*.

Acredita-se que o primeiro resultado sobre a teoria de grafos tenha sido apresentado em 1736, pelo matemático e físico suíço Leonhard Euler (1707 - 1783); o problema “Sete Pontes de Konigsberg” (figura 1, esquerda) continha uma ilha cercada por quatro regiões e sete pontes. Antes de Euler, acreditava-se que era possível atravessar todas as pontes sem repetir nenhuma. Euler provou que não era possível realizar tal tarefa, mapeando as regiões e as pontes num grafo (figura 1, direita), onde as quatro regiões se transformaram em vértices e as sete pontes em arestas; percebeu então que só era possível resolver o problema se de um vértice que possui um número ímpar de arestas, se pudesse atingir zero ou dois vértices distintos.

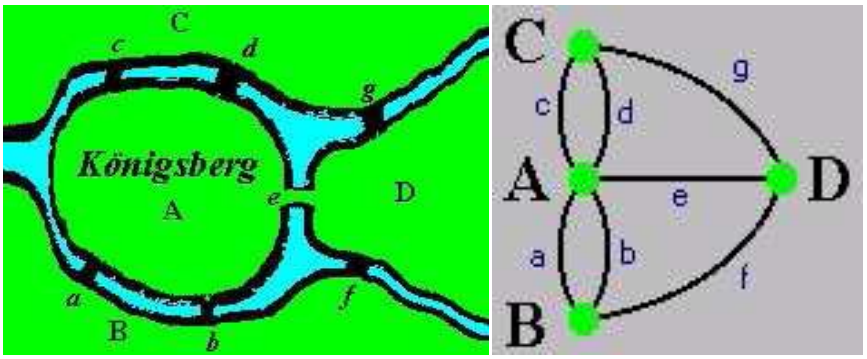


Figura 1: À esquerda, a região de Konigsberg; à direita o mapeamento realizado para grafos.

No século XIX surgiram vários outros problemas isolados:

Em 1852, *Francis Guthris* aluno de *Augustus De Morgan* enquanto coloria os municípios da Inglaterra se deparou com a seguinte questão: qual o número mínimo de

cores necessárias para que todos os municípios fossem pintados sem que dois adjacentes possuíssem a mesma cor; o mapeamento é direto sobre grafos: as regiões se tornam vértices e as fronteiras arestas.

Apesar de hoje sabermos que apenas quatro cores são suficientes para pintar um mapa plano (figura 2), apenas uma demonstração em computador em 1976 por *Kenneth Appel* e*Wolfgang Haken*[11] (Universidade de Illinois) foi capaz de comprovar tal teorema. A idéia básica é assumir que haveria ao menos um mapa que o teorema não fosse válido, provando que isso não é verdade. Utilizaram um conjunto de teoremas matemáticos reduzindo um mapa grande em mapas menores. Resolvendo o problema para os pequenos mapas, expande-se gradualmente a solução até atingir o mapa original e, assim, provar que qualquer mapa pode ser colorido com quatro cores.

Figura 2: A figura acima ilustra o problema: como colorir o mapa dos EUA com quatro cores?

Em 1856 o matemático irlandês *Sir William Rowan Hamilton* propôs um jogo baseado em um dodecaedro (sólido regular com 20 vértices, 30 arestas e 12 faces) (figura 3 esquerda), objeto de um dos seus estudos[14]. O jogo mapeava várias cidades que o matemático conhecia, sendo que o objetivo era encontrar um caminho (que iniciasse e terminasse em um vértice *v* qualquer) que percorria todas as cidades apenas uma vez (figura 3, direita). O mapeamento em grafos se dá transformando as cidades

em vértices e as ligações entre elas em arestas. Esse caminho é conhecido como caminho Hamiltoniano [12].

Outros problemas semelhantes ao proposto por Hamilton surgiram na mesma época: o problema do cavalo no tabuleiro de xadrez consiste em procurar um conjunto de jogadas em que o cavalo passe por todas as casas uma única vez e que chegue, após a seqüência de movimentos, na casa de partida; o problema do caixeiro viajante consiste em achar um conjunto de estradas entre cidades em que a distância seja a mínima e que retorne ao ponto inicial, passando apenas uma vez em cada uma.

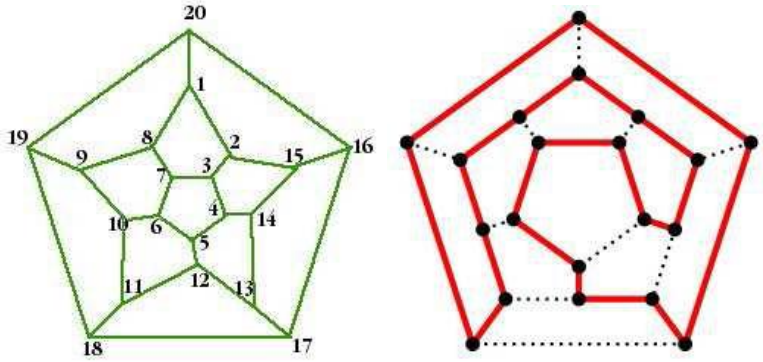


Figura 3: À esquerda o problema proposto por Hamilton; à direita, a solução do problema.

Os vários problemas que surgiram no século XIX, sua complexidade e importância, motivaram vários pesquisadores a criarem a Teoria dos Grafos.

A classe de complexidade dos algoritmos [10] implica diretamente no mapeamento computacional da solução, sendo que, em muitos casos, apenas uma solução próxima da ideal pode ser obtida.

Apesar do exponencial crescimento do processamento das máquinas[8], ainda hoje não é possível explorar soluções ideais sobre alguns problemas. Isso decorre do fato de muitos problemas pertencerem à classe NP-Completo. A classe NP-Completo é subconjunto da classe NP, que esta contida na classe de complexidade P (com resolução polinomial). A classe de complexidade P possui problemas que são resolvidos em tempo polinomial, como explorar um vetor de elementos. A classe NP

define problemas que podem ser resolvidos em geral, em tempo polinomial, ou muito próximo ao exponencial.

Já a classe NP-Completo possui problemas que, muito provavelmente, não possuem solução polinomial, apenas solução exponencial: se um problema na classe NP-Completo for resolvido em tempo polinomial, então, todos os problemas relacionados, pertencentes a esta classe, poderão ser resolvidos nesta escala de tempo. Ainda há muita discussão sobre a divisão dessas classes e ainda mais, sobre a classe NP e P[10].

Em grafos, muitos problemas se encontram na classe de complexidade NP-Completo, como é o caso do problema do caixeiro viajante (prova não se aplica ao contexto deste texto) [1]; nestes casos, heurísticas são aplicadas para que, em um tempo razoável, se possa chegar a uma solução aceitável.

Outros problemas (como o das quatro cores) ainda não possuem uma prova matemática teórica, apesar de sabermos que se pode aplicar a solução à problemas específicos.

O mapeamento computacional dos grafos, de modo geral, pode ser realizado de duas maneiras: por vetores (matrizes) ou por listas (vetores ou ponteiros): cada aplicação fornece uma otimização (desempenho ou memória, respectivamente) para determinado tipo de problema.

As implementações por matrizes, mapeia cada aresta *E* como um *booleano*, sendo que a posição na matriz de vértices A[i,j], só é verdadeira, se existir uma aresta sobre os vértices i,j. Se o grafo for esparço e apresentar um grande número de vértices a representação por matriz gera um desperdício de memória. Entretanto, o acesso as posições é direta e constante O (1). Para grafos pequenos, pode-se optar pela representação matricial dos elementos [4]. Para grafos ponderados, ao invés de armazenarmos apenas um booleano sobre a posição A [i,j], armazena-se uma estrutura, que contém os pesos sobre cada ligação dos vértices i,j.

O mapeamento por listas (ponteiros ou arranjos) mantém uma lista ou vetor de arestas sobre cada vértice. Apenas as arestas que estão presentes no grafo são representadas. A economia de memória implica em um gasto maior de processamento sobre cada vértice; porém, sendo *n* o número de vértices do grafo, o máximo de

elementos sobre cada vértice é O(n), no caso de um grafo completo (seção 2.2), ou seja, se gasta tempo O(n) para determinar se existe uma arestas que conecta o vértice atual a algum outro que possa estar presente na lista[4]. Do mesmo modo que na representação matricial, pode-se utilizar uma estrutura que armazene os pesos de cada ligação entre vértices i,j quaisquer.

Ainda com o mapeamento por listas, pode-se optar pela representação conhecida como matriz esparsa: apenas elementos que não são nulos são armazenados, de modo que se deve manter ponteiros entre os elementos que se relacionam na linha e coluna, permitindo a navegação pelos valores armazenados; há um gasto maior com ponteiros, porém se a matriz contém muitos zeros e possui uma grande dimensão, pode-se obter economia de memória. Como a navegação depende agora, não somente dos índices na matriz, mas sim de ponteiros, a lógica de acesso deve ser modificada, a fim de manter estável o acesso aos elementos; sendo *n* o número de elementos da matriz, para se chegar ao último elemento, levando em consideração a matriz completamente cheia, deve-se acessar *n* elementos (O(n) no acesso); porém, como utilizamos este modelo para matrizes que contém muitos zeros, o número de acessos deve ser muito reduzido, comparando-se com a modelagem por listas.

Se utilizarmos esta implementação para matrizes que tendem a ser complemente preenchidas, o gasto de memória tende a ser maior, devido ao grande número de ponteiros que serão criados: isso, além de consumir mais recursos de memória, tende a se tornar eficiente, dado que o acesso deve ser feito elemento à elemento.

Em ambas as modelagens (matrizes e listas), pode-se adotar um modelo de representação de grafos conhecido como listas ou matizes de adjacência: a matriz ou listas contém apenas um valor binário (um ou zero) que indica se há ou não uma ligação entre os vértices do grafo. Apesar de ser uma representação simples, podem-se armazenar, além deste valor binário, outros campos em uma estrutura, de modo a estender a representação, possibilitando que outros algoritmos (como os que trabalham sobre arestas ponderadas) possam ser executados sobre a estrutura.

Muitos algoritmos não necessitam uma representação ponderada das arestas, como é o caso da busca de todos os caminhos entre dois nós, ordenação topológica, árvore geradora mínima, etc.

Porém, muitos problemas envolvidos, principalmente na área de logística, necessitam que o peso das arestas seja armazenado para que se possa obter melhores caminhos ou rotas mais curtas sobre um trajeto. Em geral estes problemas estão classificados como NP-Completos, sendo necessário aplicar algoritmos “gulosos” ou heurísticos para a solução.

Algoritmos gulosos sempre fazem a escolha mais conveniente a cada passo de execução. Isso significa que a solução encontrada depois de *n* iterações pode não ser a melhor solução para o problema, porém, garante-se que a um tempo polinomial se encontre a solução. Heurísticas são funções aplicadas sobre o problema, considerando ou não os resultados obtidos em iterações anteriores. Como são funções aplicadas sobre um conjunto de possibilidades, há uma probabilidade, muitas vezes alta, de não se obter a melhor solução; porém, podem-se obter soluções razoáveis em tempo polinomial. Heurísticas e algoritmos gulosos também são amplamente utilizados na área de programação inteira, onde, para não se explorar de forma exaustiva o conjunto completo de soluções de um problema, aplicam-se os conceitos descritos acima para aproximar a solução da ótima [20].

Como pode ser observado acima, existem muito algoritmos que se destinam a determinados problemas. A aplicação demanda uma estrutura específica que deve ser abordada com algoritmos específicos; uma vez que a aplicação esteja complementa definida, adota-se o modelo mais conveniente, ou seja, aquele que responda melhor (em desempenho ou memória) às expectativas do usuário.

Muitas são as ferramentas que programam essas estruturas e um conjunto de algoritmos (busca, caminho mínimo, etc.). A maioria das ferramentas já prontas são disponibilizadas em bibliotecas (conjunto de classes e métodos para manipular problemas específicos). Em C++, podemos citar:

- **Boost**[9]; Boost é conjunto de bibliotecas dividida em módulos: cada módulo é responsável por fornecer estruturas e métodos para determinados tipos de

problemas. No caso específico de grafos, a Boost Graph é quem fornece essas funcionalidades; como a programação é feita sobre templates, se torna simples estender esse conjunto para os requisitos do usuário (*www.boost.org*).

- **Tom Sawyer Layout**: Conjunto de bibliotecas para otimização e pesquisa de relações, baseado em grafos. Possui um aplicativo que é adaptável com as necessidades do usuário. Ferramenta comercial (*http://www.tomsawyer.com/*).

- **NGraph**: Biblioteca livre, porém restrita. O código foi desenvolvido tomando como meta a simplicidade das operações, dado que o autor julga as ferramentas acima muito complexas e de difícil extensão. Programada com listas de adjacências, disponibilizando algoritmos como clique, máximo e mínimo grau do grafo, etc.(http://math.nist.gov/ RPozo/ngraph/index.html).

- **LEDA**: é um conjunto de algoritmos voltados para aperfeiçoar o desenvolvimento de software, contando com uma biblioteca otimizada de grafos. É uma ferramenta comercial, possuindo uma versão livre, com várias restrições, tanto quanto na utilização, como nas possíveis aplicações. (http://www.algorithmic-solutions.com/leda/index.htm).

Em Java, muitas são as possibilidades de escolha. A maioria das ferramentas é livre e possuem tópicos ou fóruns de ajuda ao usuário. Além disso, como Java tem sido amplamente explorado, dado sua facilidade de implementação e execução nativa em aplicações WEB, o número de fóruns e usuários que contribuem para o desenvolvimento e suporte é muito alto; esse conjunto de fatores, contribuem para o grande número de aplicações que utilizam Java e, em geral, algumas das ferramentas listadas abaixo.

- **GraphViz**: ferramenta completa de grafos, com algoritmos e visualização prontos, disponibilizada em bibliotecas (http://www.graphviz.org/);

- **JUNG** - (Java Universal Network/Graph Framework) é uma biblioteca para modelagem, análise e visualização de grafos, sendo utilizada por diversos projetos como em *Google Cartography* (http://jung.sourceforge.net/);
- **Java Graph Framework:** software de visualização e modelagem de problemas em grafos. (*http://www.tensegrity-software.com*).
- **JGraphT e JGraph: JGraphT** é um conjunto de bibliotecas livre, voltada para o desenvolvimento de algoritmos relacionados a grafos, como grafos ponderados, sub-grafos, caminhos mínimos, etc (http://www.jgrapht.org/). JGraph é uma ferramenta voltada para visualização dos grafos, sendo que utiliza a JGraphT para manipulação matemática e lógica destas estruturas (http://www.jgraph.com/jgraph.html).

2.2 Definições

- **grafo**: Conjunto não vazio de elementos, chamados de vértices *V* e um conjunto de pares não ordenados de vértices, chamado arestas *E(VxV)*: *G(V,E)*.
- **grafo direcionado:** Grafo onde existe uma ordem nas arestas: *E(U,V)*, implica que *U* precede *V*.
- **grafo não direcionado:** Grafo onde E(U,V) = E(V,U).
- **vértices adjacentes:** São vértices que possuem uma aresta que os liga: *U* é adjacente a *V* se existe E(U,V).
- **grau de um vértice**: É o número de aresta que o vértice possui.
- **vértice isolado**:É um vértice que não aparece em E, ou seja, seu grau é zero.
- **vértice conectado:** Vértice com grau diferente de zero.
- **caminho:** Conjunto de vértices *V* e arestas *E(V xV)* que os ligam.
- **comprimento de caminho:** Número de arestas presentes no caminho.
- **caminho simples:** Caminho que não possui vértices repetidos.
- **ciclo:** Caminho onde o vértice inicial é o mesmo que o vértice inicial.
- **caminho acíclico:** aquele que é cíclico.
- **grafo conectado:** Aquele que apresenta caminhos entre cada par de nós.

- **grafo fortemente conectado:** Aquele que dado dois vértices quaisquer, eles são alcançáveis a partir de um outro.
- **grafos isomorfos:** Dois grafos G=(V, A) e G'=(V', A') são isomorfos se existir uma bijeção f : V-> V' tal que (u, v) E A se e somente se (f(u), f(v)) E A'.
- **sub-grafo:** H=(V',E') é sub-grafo de G=(V,E) se V' _ V e E' _ E.
- **grafo ponderado:** Grafo onde as arestas possuem peso.
- **grafo completo:** Grafo simples onde todos os vértices são adjacentes.
- **grafo conexo:** Grafo onde há pelo menos um caminho entre cada par de vértices.
- **back-tracking** É um algoritmo geral para encontrar todas (ou algumas soluções) para alguns problemas, construindo incrementalmente candidatos para as soluções, abandonando cada candidato parcial *c* (recuando), logo que determina que *c* não pode ser eventualmente uma solução válida.

2.3 Principais Algoritmos

Nesta seção discutimos brevemente alguns dos algoritmos mais utilizados na área de grafos.

Busca em profundidade: Utilizado para explorar em profundidade um grafo a partir de um nó *v*.

A idéia é ir o mais profundo possível a partir de um vértice *v*: visita-se cada aresta do nó; a cada nó descoberto, o processo se repete recursivamente. A cada visita, um novo nível do grafo é explorado. Quando se chega ao final, ocorre back-tracking, explorando-se as arestas do vértice que ainda não foram processadas. A complexidade, uma vez que se percorre todos os nós e todas as arestas é O(|E| + |V|), onde |E| é o número de arestas e |V| o número de vértices.

Busca em largura: Ao contrário da busca em profundidade, a idéia é explorar o grafo horizontalmente, explorando-se cada aresta do nó; somente depois de visitadas todas as arestas do um nó *v*, parte-se para os seus vizinhos. Usa-se recursividade para retornar aos vértices que já foram explorados na largura. A complexidade, uma vez que se percorre todos os nós e todas as arestas é O(|E| + |V|), onde |E| é o número de

arestas e |V| o número de vértices. O algoritmo de busca em largura pode ser utilizado para verificar se ocorre ciclos (loops) no grafo: quando encontrada uma aresta que leva de um nó *v* `a um nó inicial *u* (aresta de retorno) pode-se concluir que há um ciclo no grafo. O caminho mais curto entre dois nós pode ser encontrado utilizando este algoritmo: como a busca ocorre em largura, basta armazenar os nós antecessores ao nó explorado, imprimindo o caminho ao achar o nó de destino.

Ordenação topológica: Este algoritmo é utilizado para ordenar um grafo que seja direcionado e acícliquico (seção 2.2); grafos com este aspecto representam tarefas que devem ser cumpridas antes de outras, ou seja, que necessitam de alguma precedência. Se existe uma aresta do vértice *u* a *v*, então, necessariamente, *u* deve aparecer antes de v. A busca em profundidade pode ser utilizada para encontrar a solução: parte-se de um nó *u* até encontrar o nó mais profundo da busca, adicionando este ao final de um vetor de nós; repete-se o procedimento para nó na pilha de recursão. Ao final, no vetor de nós, teremos os nós do grafo ordenado, por ordem de precedência.

Componentes fortemente conectados: Um componente fortemente conectado em um grafo dirigido G=(V,E), é um caminho maximal de nós $C \in V$, tal que para todo par de vértices$(u,v) \in C$, existe um caminho de *u* para *v* e de *v* para *u*. Para tal, utiliza-se a busca em profundidade para o grafo *G* armazenando todos os nós que são alcançáveis por *u*, ordenando os vértices decrescentemente, pelo tamanho do caminho. Após isso se aplica o mesmo processo, porém no grafo transposto G^T , partindo-se dos vértices que foram armazenados; como os vértices foram armazenados em ordem pelo maior caminho, se o nó *u* for atingido pelo caminho gerado por *v* em G^T, então temos uma componente fortemente conectada.

Árvore geradora: Uma arvore geradora de um grafo *G=(V,E)* é uma árvore, ou sub-árvore (sub-grafo de G) que contenha todos os vértices de G. A busca em profundidade ou em largura podem ser utilizadas para criar a esta árvore. Logicamente, uma árvore geradora só pode ser criada se o grafo for conexo.

Árvore geradora mínima: Este algoritmo se aplica a grafos não direcionados e ponderados. Árvore geradora mínima (AGM) é um subconjunto de aresta $T \in$ A de um

grafo *G=(V,E)*, cuja somatória dos pesos é mínima, conectando todos os vértices. Várias abordagens são passíveis a este algoritmo:

- Gulosa: Parte-se de um nó *u* avaliando o peso das arestas; adiciona-se à AGM esta aresta apenas se ela não inclui um nó de uma aresta já inserida. Este algoritmo é guloso porque avalia apenas as arestas do momento.
- Prim: Parte-se de um nó *u* até que os nós da AGM gerem todos os nós de G. A cada passo uma aresta (a de menor peso) é inserida na AGM, isso significa que o algoritmo considera todas as arestas incidentes ao nó em questão.
- Kruskal:A idéia deste algoritmo é ir adicionando aresta de tal forma que não se formem ciclos, ou seja, é um algoritmo guloso, dado que faz a melhor escolha baseado apenas nas arestas que partem de um determinado nó.

Caminho mais curto: O algoritmo de Dijkstra pode ser utilizado para calcular a caminho mais curto: utiliza uma técnica chamada relaxamento que toma o valor p(v) como um limitante superior para o tamanho do caminho, ou seja, se a soma de uma nova aresta ao caminho exceder p(v), então outro caminho deve ser tomado (back-tracking). Mantém-se uma lista de antecessores e pesos máximos *p(v*) em cada vértice *v* de um Grafo G. A cada iteração, uma nova aresta que não excede o peso máximo do caminho é adicionada ao caminho mais curto.

2.4 Interface

Interface é um termo utilizado para descrever a comunicação entre dois meios distintos quaisquer.

Em computação o primeiro sentido à este termo foi dado entre a comunicação dos componentes de hardware e software, de modo a possibilitar a troca de informações e o processamento interativo entre eles.

Em HCI, o termo interface se refere ao estudo dos meios de interação entre os usuários e o sistema que eles operam [5]. HCI é a ciência que estuda a representação da informação e sua interatividade com o usuário, de modo a permitir que o usuário possa aprender e obter de forma agradável o maior número de informações possíveis; é uma ciência complexa, pois estuda os aspectos físicos do corpo humano, como

representação das cores, sinais auditivos, etc. e, sociais, de modo a reconhecer o ambiente em que o usuário esta inserido. Alguns exemplos desta representação são os próprios Sistemas Operacionais utilizados hoje em dia: toda a interface foi criada de modo a reproduzir o ambiente de trabalho que o usuário esta inserido: a área de trabalho é um local onde se pode colocar vários documentos, arrastá-los e jogá-lo fora (na lixeira); as pastas seriam como as gavetas de um armário: armazenam documentos que são, em geral, relacionados; o sistema tende a ser paralelo, pois permite que vários programas sejam executados ao mesmo tempo, representando as várias tarefas que o usuário pode realizar em uma situação real de trabalho.

Tipicamente, sistemas que exigem interação com o usuário tendem a seguir os padrões de desenvolvimento descrito em HCI. Dado que usuários reais manipularão a interface, é inconsistente apenas focar o desenvolvimento matemático da aplicação, reprimindo sua representação gráfica e interativa.

Como exemplo, podemos citar as primeiras versões dos sistemas operacionais, mais especificamente o Linux: Linux é sistema operacional derivado do UNIX (criado em 1969 pela AT&T's Bell Laboratories): as primeiras versões possuíam uma ínfima interface gráfica e eram utilizadas apenas em servidores e por alguns profissionais na área de informática; com o avanço da tecnologia, várias melhorias foram inseridas, tanto no kernel (núcleo do SO), quando na interface gráfica (GNOME e KDE, principalmente), possibilitando maior interatividade, impulsionando sua disseminação entre empresas e entre usuários comuns; não somente este aspecto impulsionou o seu crescimento, mas também, por ser uma iniciativa livre, se tornou uma alternativa ao Windows, sistema proprietário da Microsoft. Outras informações sobre Sistemas Operacionais, sua filosofia e implementação podem ser encontrados em [7].

Seguindo este modelo, buscamos criar uma interface que seja agradável ao usuário. Como na maioria das aplicações que envolvem grafos, muitos elementos devem se mapeados na tela, fazendo-se necessário estudar estratégias de representação, tanto dos elementos, como de suas relações; uma vez que estes estejam bem representados graficamente, a análise e aplicação de algoritmos específicos se tornam naturais, uma vez que saberemos quais são as informações que estamos buscando.

Muitas são as possibilidades de algoritmos para criar *clusters*, ou seja, relacionar dados específicos sobre um domínio em comum. Em grafos ou redes complexas, quando a dimensão e comprimento das arestas é trivial, vários são os métodos que podem ser aplicados para o agrupamento de nós relacionados sob algum critério: FDP (*Force Directed Placement*)[13], Star Coordinates [16], MDS (*Multimensional Scaling*) [19], LSP (*Least Square Projection*) [17]. Além dessas técnicas, bons exemplos de customização podem ser adquiridos analisando os programas disponíveis no mercado, tanto comerciais, quanto livres, como por exemplo o PEX (Projection Explorer) [18] (figura 4); PEX é um sistema capaz de explorar projeções tanto no espaço bidimensional como no espaço tridimensional. O sistema permite trabalhar com tabelas estruturadas, dados de distância entre os elementos, conjunto de textos e consultas na WEB. Possui vários algoritmos de visualização já implementados, facilitando a exploração dos dados; podem-se dividir os resultados em duas janelas distintas, com tipos de projeções diferentes, sendo que as operações realizadas sobre uma janela são refletidas sobre a outra; é uma boa ferramenta neste contexto (redes complexas e grafos) devido às funcionalidades que ela possui, como os diversos tipos de projeção (no sentido visualizar agrupamentos) e as técnicas empregadas para exploração dos dados projetados; zoom, rotação, transparência, criação de superfícies sobre dados do mesmo *cluster* (*Convex Hull*), entre outras.

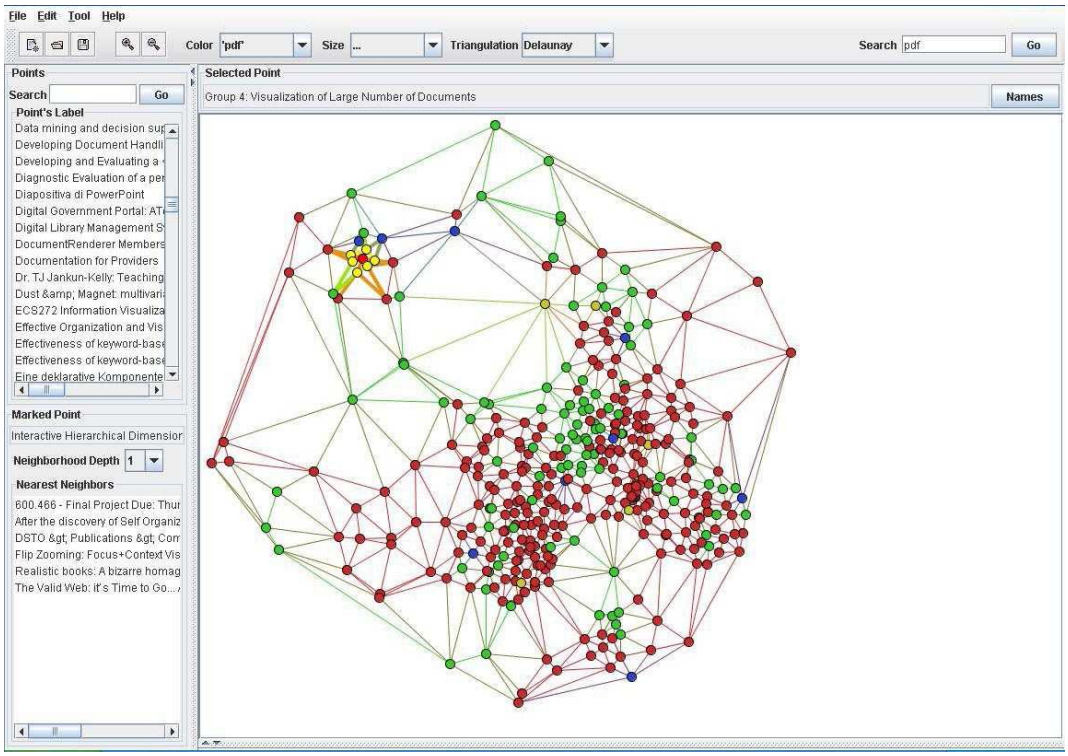


Figura 4: PEX - Projection Explorer: cluster sobre o conjunto de dados.

2.4.1 Interfaces Relacionadas a Grafos

Abaixo, algumas interfaces que manipulam grafos; pretendemos seguir os bons paradigmas destas interfaces, a fim de construir a nossa, baseado nas boas funcionalidades, buscando corrigir os problemas e as dificuldades de manipulação de cada uma.

- Walrus: Graph Visualization Tool (figura 5).

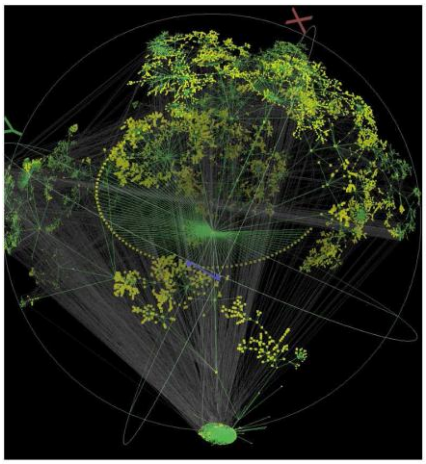


Figura 5: Walrus - Explora projeções em grafos, etc.

- Graphviz: Graph Visualization Software (figura 6).

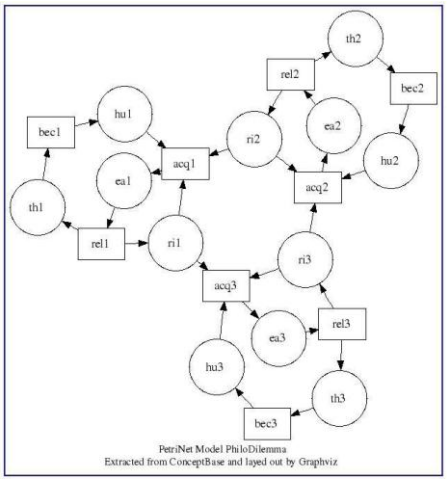


Figura 6: Graphviz: ferramenta para visualização de grafos, baseado numa estrutura definida em seu código.

- JGraph: The Java Open Source Graph Drawing Component (figura 7).

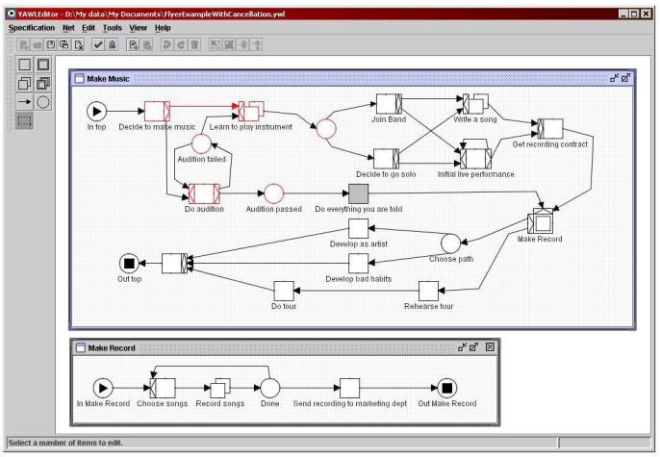


Figura 7: JGraph: interface para visualização de grafos; estrutura mapeada em seu código

- Jung: Java Universal Network/Graph (figura 8).

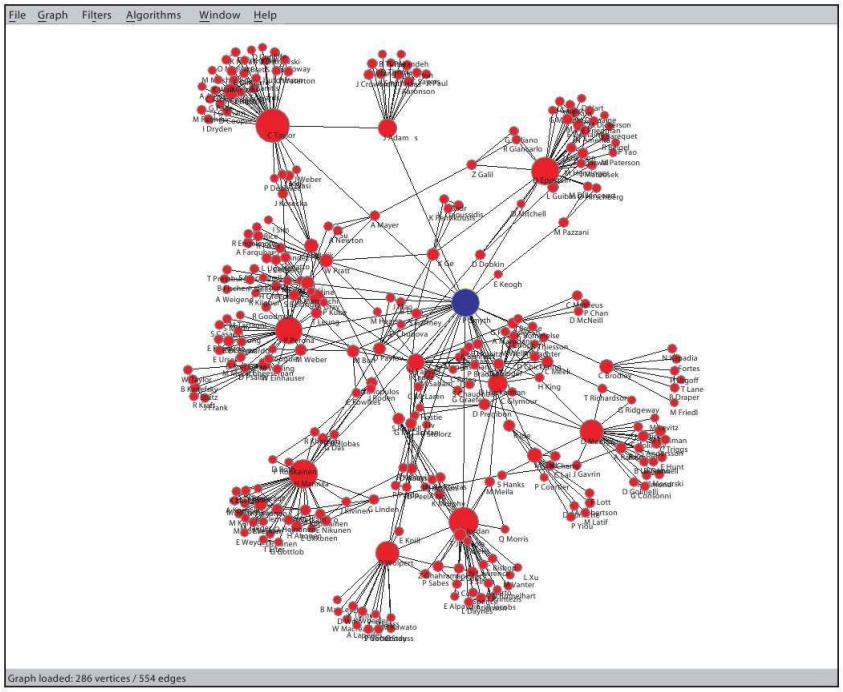


Figura 8: Jung: interface para visualização de grafos; estrutura mapeada em seu código.

3. O padrão Model – View – Controller (MVC)

Definição

MVC (*Model*, *View*, *Controller*) [21] é um padrão de desenvolvimento de software que busca a separação da parte lógica e visual do sistema. O ***Model*** (modelo de dados) contém toda a lógica funcional da aplicação; o ***View*** (objeto visual) possui a visualização do modelo sendo completamente disjunto do mesmo; o ***Controller*** (objeto de controle) é responsável por controlar a comunicação entre os objetos do *Model* e do *View*, atualizando ambos quando ocorre alguma alteração em algum deles. Um fato interessante deste padrão é que ele permite criar múltiplas visualizações de um mesmo modelo de dados: com um modelo, podemos criar um documento, um diagrama, um gráfico, ou qualquer tipo de visualização que possa ser feita sobre ele.

Há dois tipos de MVC: o primeiro (figura 9) não permite que os objetos pertencentes ao *View* e ao *Model* se relacionem, forçando que qualquer tipo de comunicação entre eles seja realizada via o obejto *Controller*. O benefício deste modelo é que isola completamente a lógica de negócio da visualização, porém, pode sobrecarregar a classe *Controller* com um número excessivo de mensagens; caso essa comunicação seja feita via internet, pode se tornar custoso.

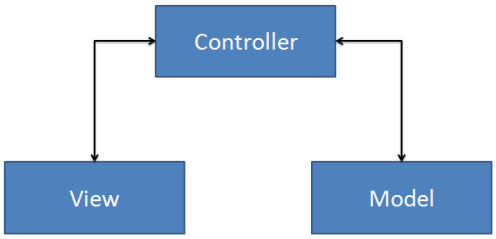


Figura 9: MVC isolado pelo controller

O outro tipo (figura 10) é mais flexível e permite a comunicação direta entre o *View* e o *Model*. Deve-se tomar o cuidado que as alterações realizadas em qualquer um dos dois passem pelo objeto *Controller*, de modo a atualizar ambas as partes. Este modelo facilita a programação e não sobrecarrega o *Controller*, visto que algumas

mensagens de controle podem ser ignoradas devido ao acesso direto entre os objetos do *View* e do *Model.*

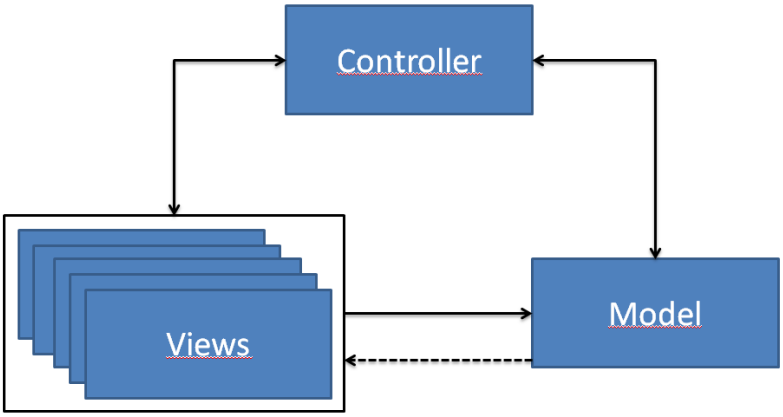


Figura 10: MVC onde o View e o Model se comunicam.

Cada objeto *View* e cada objeto *Model* devem ser registrados no objeto *Controller.* Quando uma alteração ocorre no *Model*, ele notifica o *Controller* que imediatamente registra a alteração no *View*; O *View* altera os dados gráficos e retorna uma mensagem para o *Controller* notificando que está atualizado; o mesmo ocorre quando o *View* sofre uma alteração.

Adaptação do modelo ao projeto

O projeto utiliza o segundo tipo de MVC, permitindo comunicação direta entre os objetos do *View* e do *Model*. Esta decisão de projeto foi tomada para simplificar a extensão futura do código ou a especialização para alguma necessidade do usuário, porém, deve-se tomar o cuidado em acessar os parâmetros do *Model* diretamente, de modo que não se pode realizar qualquer alteração na sua estrutura diretamente sem antes passar pelo objeto *Controller*, garantindo a consistência do sistema.

Três interfaces foram definidas no sistema: *AbstractController*, *AbstractModel* e *Viewable*: essas interfaces contém o cabeçalho dos métodos necessários para a implementação do MVC. Utilizamos as classes base do Java (*java.beans. PropertyChangeListener e java.beans. PropertyChangeEvent*) para implementação; isso garante que a aplicação não seja dependente de bibliotecas externas e, além disso, que

se tenha muita documentação on-line para pesquisa. A classe *AbstractController* possui uma vetor de objetos *Model* e *View´s* que devem ser registrados previamente; a classe *AbstractModel* possui os métodos necessários para avisar o objeto *Controller* sobre mudanças e passar os parâmetros que devem ser atualizados para o objeto *View*. A classe *Viewable* possui apenas um método, sendo responsável por avisar o *Controller* que alguma propriedade gráfica foi alterada, atualizando o *Model*.

Este padrão foi inserido após o início do projeto. Inicialmente não utilizávamos este padrão de desenvolvimento, pois não pensamos que poderia haver múltiplos modelos e visualizações; conforme surgiu a necessidade, buscamos na literatura um padrão de desenvolvimento que pudesse resolver tal problema. Contamos com a ajuda do professor Dr. Adenilso da Silva Simão para realizar uma reengenharia de software. Após alterar o diagrama de classes, houve maior produtividade e novas possibilidades de extensão.

Abaixo, decidimos expandir a discussão das classes do pacote *View* por serem mais complexas do ponto de vista funcional.

Quando dizemos ***pacote***, nos referimos ao agrupamento semântico de classes no Java. Um pacote contém classes que contém algum relacionamento, por menor que seja, o que permite melhor organização do código. As classes são inseridas dentro de um pacote e são referenciadas tomando por base o nome do pacote, seguido do nome da classe ou sub-pacote, como por exemplo*: Java.awt.dnd* (*awt* é um **sub-pacote** do **pacote** de classes *Java*; *dnd* é um **sub-pacote** do **pacote** *awt*, que por sua vez, contém todas as classes ou interfaces necessárias para manipulação de dados utilizando Drag-and-Drop).

3.1 View

Este pacote contém todas as classes responsáveis pela visualização do modelo. Devido a sua complexidade a dividimos em outros dois pacotes internos (*Equipamento* e *Graph*) que serão discutidos mais abaixo. Partiremos de uma análise das classes externas, descendo nos sub-pacotes conforme necessário.

A visualização esta dividida em quatro partes:

- *Panel*
- *Tree*
- *Log*
- *Canvas*

3.1.1 Panel

A classe ***Panel*** é responsável por representar cada elemento que pode ser inserido pelo usuário. A classe ***Tree*** contém a representação de uma árvore hierárquica de elementos definidos pelo usuário; cada elemento possui uma parte lógica e outra visual, conforme pode ser visto em 3.1.2. A classe ***Log*** representa uma área de texto que contém uma lista de eventos que foram realizados pelo usuário em uma sessão do sistema. A classe ***Canvas*** é um tipo de *Frame*: um *Frame* é um *container* de outros elementos gráficos, como por exemplo, uma janela do Windows ou Linux que contém uma área de visualização de pastas, arquivos e imagens, menu, etc. No *Canvas* outros objetos gráficos, como o *Tree* o *Panel* e *Log* são inseridos, compondo a aplicação; como todos os elementos estendem algum elementos gráfico do Java (*Canvas* o *JFrame*, o *Panel* o *JPanel*, etc.) eles podem ser inseridos diretamente sobre ele. Optamos por utilizar apenas os elementos gráficos contidos nos pacotes de classes *swing* e *awt,* pois contém ampla documentação on-line e ótima aceitação da comunidade que utiliza a plataforma Java para programação.

A classe *Panel* estende um container do “Java.swing” chamado ***JPanel***. O objeto *Panel* é responsável por armazenar os elementos gráficos que são manipuláveis pelo usuário. Cada elemento gráfico recebe como parâmetro um objeto ***Graphic*** do *JPanel*. A classe *Graphic* é a classe responsável por conter os métodos de desenho, como linhas, círculos, etc. Compartilhando o mesmoobjeto *Graphic* do *Panel* com todos os elementos gráficos manipuláveis, garantimos que eles sempre serão desenhados sobre a mesma área, preservando a consistência do sistema.

Os elementos são inseridos no objeto *Panel* através de um método chamado *Drag-and-Drop* (DND). O DND permite que elementos de áreas distintas sejam

compartilhados. Quando ocorre um evento de *Drag* (arraste), o elemento que está sofrendo o evento deve ser armazenado. Ocorrendo um evento de *Drop* (soltar), a área que irá receber o elemento antes armazenado deve estar preparada para converter o elemento em algo que pode manipular. No projeto, o *Drag* ocorre quando o mouse é mantido pressionado e movido sobre um elemento do *Tree* ou do *Panel*; pode-se soltar elementos que venham do objeto *Tree* no *Panel*, ou seja, o *Panel* é capaz de converter um nó da árvore em elemento gráfico; o contrário não é válido. Por padrão, elementos de um mesmo tipo possuem o *Drag and Drop*, de modo que se pode trocar elementos de ordem de uma árvore ou organizar os elementos do *Panel*. No Java há uma interface utilizada para programar o DND (*Java.awt.dnd.**); o conjunto de elementos que podem ser arrastados compõe um objeto ***Flavor***. O objeto *Flavor* é registrado na aplicação; por padrão, strings e alguns elementos do Java possuem uma pré-implementação. Como os elementos do projeto são complexos, deve-se registrá-los utilizando um objeto ***ModelTransferable***. O *ModelTransferable* armazena um *Flavor* quando ocorre um evento de *Drag*. No *Drop*, o *ModelTransferable* é inserido na área de *Drop* se ele pode transformar o objeto de origem no do destino, sendo destruído, ou, rejeitado se não pode ser convertido.

Quando arrastamos um elemento que veio do *Tree* sobre o *Panel*, ele é imediatamente convertido em elemento gráfico; se mantivermos o mouse pressionado e voltarmos o elemento para o objeto *Tree*, nada irá acontecer, porém, se soltarmos o mouse, o *Drop* será invocado e o elemento será inserido na estrutura topológica. Caso soltemos o elemento qualquer outro lugar fora do *Panel*, ele será automaticamente rejeitado. Podem-se arrastar elementos que já foram inseridos no *Panel*: neste caso, apenas a posição de desenho muda, não ocorrendo nada na topologia.

Além de conter os elementos gráficos, o *Panel* também é responsável por manipular os eventos de Mouse, como clique, duplo clique, etc. O objeto ***MouseListener*** é responsável por gerenciar os eventos do mouse. Pode-se adicionar diversos *MouseListeners* no *Panel*, dividindo as ações por classes que contenham uma lógica em comum (pan e zoom, seleção, clique e duplo clique, etc.). Quando ocorre um clique sobre o *Panel*, o objeto *MouseListener* registrado é invocado, disparando o

evento de clique; o mesmo acontece com os outros eventos, como duplo clique, botão do meio, scroll etc. Um clique sobre o *Panel* pode disparar vários eventos e processamento em várias classes, como seleção (clicar e soltar), movimento (clicar e arrastar), etc. Como a maioria dos eventos envolve uma mudança na topologia, o *Controller* sempre é invocado; quando ocorre a seleção de um objeto, por exemplo, deve-se armazenar o seu estado (selecionado ou não selecionado) para que as ações posteriores como arraste, deleção, copia, etc. possam ser realizados. O *Panel* não armazena esses elementos, ele apenas invoca o *Controller* passando as coordenadas de referência da tela. Como pode haver ações de *Zoom* e *Pan*, todo evento de mouse sofre uma transformação antes de ser passado ao objeto *Controller*.

As operações de *Zoom* e *Pan* sempre mantém o canto superior esquerdo da tela como referência. Assim como no openGL, uma matriz de rotação ou translação está associada a estas transformações; a única mudança é o ponto de referência que no openGL geralmente é o centro do sistema de coordenadas. O *Panel* armazena essa matriz e o seu estado, sempre aplicando a transformação sobre a posição do mouse.

O *Panel* permite a multi-seleção de objetos: se o botão esquerdo do mouse é mantido pressionado sobre uma área vazia do *Panel*, sendo arrastado, o *Panel* dispara um evento de multi-seleção de componentes. Todos os objetos que estão no quadrado formado pelo clique inicial e final do mouse serão selecionados; O *Controller* é invocado e os objetos selecionados são armazenados no *GI* (seção 4.2.3).

3.1.2 Tree

A classe Tree é uma *View* que estende a *JTree* do Java *(Java.swing*). Uma instância de uma JTree é objeto que armazena elementos (apenas nos nós folha) em forma de árvore, com um nó raiz e sub-nós que podem ou não conter sub-árvores. Os elementos são armazenados em forma compacta, de modo que a serem expandidos apenas quando há necessidade: quando ocorre um clique sobre um nó da árvore, ele é imediatamente expandido; como poder haver sub-árvores, esse processo é recursivo, na medida em que ela é explorada, terminando em um nó folha. Essa visualização pode

representar uma hierarquia, onde cada sub-árvore é uma organização lógica do sistema; cada sub-árvore contém elementos relacionados por algum critério.

Há duas árvores no sistema:

A primeira contém os elementos que podem ser manipulados pelo usuário. Quando o sistema é iniciado, o XML é carregado no sistema em forma de árvore, respeitando a organização lógica do arquivo (seção 4.1); o nó raiz contém o *root* do XML; a cada sub-nó percorrido, um novo nível na árvore é criado; quando os elementos são lidos do XML, eles são inseridos no último nível criado.

Assim como no *Panel* a interface do *Drag and Drop* também foi programada na classe *Tree*. Como os elementos são arrastados do *Tree* para o *Panel*, deve-se definir como os elementos são transferidos. Diferente do Panel que aceita o "*Drop*" desses elementos e os converte em elementos gráficos, o *Tree* apenas deve fornecer os elementos, deste modo, apenas uma parte da interface do DND foi criada (*DragSourceListener*). A interface *DragSourceListener* é uma interface criada apenas para tratar os eventos de "*Drag*" que podem ocorrrer: um elemento "*ModelTransferable*" com o *Flavor* suportado pelo sistema, previamente programado, é criado enquanto ocorre o evento de arraste. A área de "*Drop*" deve ser responsável por tratar o evento de soltura do mouse. Pode-se especificar este elemento gráfico permitindo que ocorra "*Drop*": basta criar a outra parte da interface do DND como foi descrito no *Panel*.

Toda vez que ocorre um clique na árvore, um evento chamado "*valueChanged*" é invocado: este evento indica que houve uma expansão na árvore ou um elemento folha foi selecionado. Pode-se obter o caminho pelo método "*getLastSelectedPathComponent*" e o último elemento selecionado pelos métodos "*getUserObject*" e "*isLeaf*".

A segunda classe *Tree* contém uma lista dos elementos que já foram inseridos no projeto: cada elemento arrastado da *Tree* de elementos para o *Panel* é inserido nesta árvore. Ao contrário da primeira, ela é responsável por armazenar os objetos e permitir que eles sejam facilmente encontrados no *Panel*: quando ocorre um clique sobre um elemento de um objeto contido no *Tree*, ele é imediatamente destacado no

Panel, como se tivesse sido selecionado. A interface definida aqui não é a mesma da primeira: existe um evento de DND, porém, é um evento padrão, que apenas define a interface, podendo arrastar elementos e soltá-los sobre ela mesma, porém, sem nenhuma reação. Novamente, esta decisão de projeto foi tomada prezando pela especialização do sistema pelo usuário: basta definir as ações da interface, dado que ela já esta criada. Do mesmo modo como a primeira árvore, os métodos "*valueChanged*", "*getLastSelectedPathComponent*", "*getUserObject*" e "*isLeaf*" são utilizados para poder selecionar os elementos do *Panel*.

Ambas as árvores necessitam ser registradas no objeto *Controller*, pois necessitam de acesso ao modelo e são tipos distintos de visualizações dos mesmos componentes. No primeiro caso o *Controller* é notificado que um evento de DND está ocorrendo; inicia-se um ciclo envolvendo a *View* (*Panel*) e o *Model*; ocorrendo um evento de *Drop* que seja aceito o *Controller* é notificado, enviando uma mensagem à outras *View*(s), então a *Tree* que contém a lista de objetos do *Panel* é notificada, atualizando os componentes já inseridos, notificando o *Controller* quando o elemento for inserido, terminando o clico. No segundo caso, um elemento da lista de objetos já criados no *Panel* é selecionado, enviando uma mensagem ao *Controller*; o *Controller* dispara um evento nas outras *View*(s), notificando que um elemento foi selecionado; O *Panel* acessa o GI e marca o elemento como selecionado, terminando mais uma vez o ciclo.

A especialização das árvores pelo usuário é simples, uma vez que basta seguir a criação do modelo e da estrutura de visualização para o elemento ser inserido nas árvores. Podem-se customizar algumas propriedades de visualização das árvores vindas da própria API do Java. Elas estendem a classe *JTree* do "*Java.Swing*"; basta seguir algum tutorial da internet para alterá-la como necessário.

3.1.3 Log

A área de Log registra as ações executadas pelo usuário: em alguns momentos é importante poder analisar quais foram as ações executadas para se chegar a certa

configuração do sistema; como essas ações tendem a ser muitas durante a execução de um programa, é extremamente importante manter um registro de tudo o que foi feito.

A classe Log estende um elemento do Java chama *JTextPane*, que quando instanciado é um objeto visual que armazena texto. A cada operação realizada pelo usuário, o objeto *Controller* é invocado, enviando uma mensagem ao objeto Log que concatena a descrição da ação com o texto que já está presente no painel de texto. Isso cria um histórico de ações realizadas pelo usuário enquanto esta utilizando o programa; como se pode trabalhar sobre um mesmo arquivo uma série de vezes, não é interessante recuperar este registro toda vez que o programa é iniciado, pois, muitas ações que poderiam ser recuperadas através do *Redo* e *Undo*, não estariam mais disponíveis, levando a um processamento extra e desnecessário. Deste modo, mantemos apenas as ações do usuário da última seção.

Foi necessário estender o elemento puro do Java devido ao fato de estarmos trabalhando o modelo MVC: o elemento *JTextPane* não está preparado para ser utilizado com este modelo, dado que ele a fluxo de mensagens é via *Controller*, ou seja, o controle foge ao escopo do fluxo de informações normalmente utilizado nas aplicações mais simples, sendo localizado e específico.

3.1.4 Canvas

A classe ***Canvas*** é um container de elementos gráficos: ele agrupa de forma ordenada os elementos que permitem interação com o usuário, como botões, painéis, árvores, barras de ferramentas, menus, etc.

Um layout define como os elementos são organizados e agrupados caso mais de um componente divida o mesmo layout: cada elemento gráfico possui um layout próprio; este layout define como os elementos que estão contidos dentro do objeto são apresentados, como, por exemplo, o texto de um botão, seu tamanho, etc. Além do layout interno de cada elemento gráfico, existe o layout da aplicação, onde cada elemento é inserido. Este layout global pertence ao Canvas que é um objeto que estende o *JFrame* (*Java.Swing*).

Um Frame (*JFrame*) é um objeto que permite armazenar qualquer elemento que estenda a API *Java.Swing*, representado uma janela do sistema. Como o Java é interpretado, não precisamos nos preocupar em como gerenciar o sistema de janelas, simplificando muito o desenvolvimento da aplicação. Pode-se definir algumas ações relacionadas ao manuseio das janelas, como por exemplo, qual ação ela irá executar ao ser fechada; as operações são definidas no próprio Frame, através de uma enumeração de possibilidades e, em geral, terminam o programa, porém, pode-se deixá-lo executando em segundo plano caso a aplicação necessite ser encerrada posteriormente.

Basicamente a classe *Canvas* possui 6 áreas principais: árvore de elementos, árvore de objetos inseridos no *Panel*, *Panel*, Log, barra de botões e menu. As quatro primeiras foram discutidas logo acima; as duas últimas contêm botões como o de zoom, pan, salvar, etc. e menus, como arquivo, editar, etc., respectivamente.

A maioria das operações realizadas sobre o objeto Frame passa pelo controlador, pois alteram o modelo diretamente, como deleção de componentes, simulação, execução de algoritmos sobre a matriz de adjacência, etc.

Operações como copiar e colar, por exemplo, são operações intermediarias, pois não alteram o modelo diretamente, dependendo de alguns comandos do usuário: quando uma cópia de objetos do *Panel* é feita, um estado intermediário é criado, armazenando os objetos selecionados em uma área transitória, contida no *Canvas*; assim que um comando *colar* é identificado, os objetos já armazenados no *Canvas* são enviados ao *Model* via *Controller* para serem inseridos; o *Model* envia uma mensagem ao *Controller* dizendo que os objetos foram inseridos, esvaziando o buffer armazenado. Uma nova mensagem é enviado ao *Controller* que atualiza todas as *View*(s).

As operações que alteram somente a visualização do sistema, sem envolver o *Model*, como Zoom e Pan não passam pelo *Controler*: como não é necessário envolver o *Model* em manipulações gráficas, não faz sentido mandar mensagem ao controlador para que atualize todas as visualizações: as árvores não devem ser alteradas se a cor ou a escala dos elementos do *Panel* são alteradas.

A edição do *Canvas* pelo usuário é relativamente simples; apesar de parecer complexo devido ao grande número de elementos gráficos que manipula, o *Canvas*, por estender um *JFrame*, se torna comum sob o ponto de vista de um programador que já tenha tido algum contato com o desenvolvimento utilizando gráficos em Java. Qualquer aplicação Java que exiba gráficos ou que tenha alguma interação com usuário, que não seja via terminal, parte do desenvolvimento de um JFrame. Construímos o *Canvas* partindo do mais simples ao mais complexo; o diferencial é que previamente ao desenvolvimento, estudamos a aplicação e os problemas que iríamos enfrentar, desenvolvendo um diagrama de classes que guiou o desenvolvimento. Com o diagrama de classes e um pequeno conhecimento em Java, mais especificamente utilizando componentes Swing, qualquer programador é capaz de compreender e estender o código, adequando-o às suas necessidades. As barras de menu, assim como botões, podem ser adicionados editando o objetos que os representam dentro do *Canvas*.

Apesar do NetBeans (seção 1) fornecer um conjunto de ferramentas para o designe de formulários, optamos por construir os formulários via código. Esta decisão de projeto foi tomada devido ao fato de utilizarmos elementos que estendiam os objetos base do Java.Swing: se utilizássemos as ferramentas automáticas, ficaríamos restritos a utilizar apenas a base do Swing, sendo necessário realizar “cast” entre objetos. Como este fato poderia acarretar perda de processamento, devido à sobrecarga de métodos, métodos virtuais, etc. criamos todos os elementos via código, sem auxilio de nenhuma ferramenta. Devido a este fato, inserimos algumas operações base nos menus, como ***Arquivo*** e na barra de botões, como ***Sair*** e ***Novo*** de modo a servir de exemplo ao usuário.

3.1.5 View – Equipamento

Como dito anteriormente, este pacote é um sub-pacote do View. Cada elemento pertencente a este pacote possui uma representação gráfica do modelo, por exemplo: se houver um objeto instanciado que estende a classe ***Model-Equipamento*** (seção 4.2.5) chamado ***FonteComputador***, a classe que contém os métodos de desenho deste elemento estaria neste pacote e por definição de projeto se chamaria ***GFonteComputador.*** Todo elemento presente neste pacote segue a nomenclatura G**x**,

onde **x** representa o elemento criado no ***Model- Equipamento – Persistence*** (seção 4.2.4); deste modo, ao encontrarmos um objeto em qualquer parte do código que comece com G, saberemos que ele representa a parte gráfica de um objeto que contém um modelo de dados.

Qualquer elemento desta classe deve estender a classe GEquipamento: esta classe contém os métodos e as propriedades gráficas que todo elemento que pode ser manipulado deve conter. Há três métodos principais que podem ser sobrescritos por cada componente:

- Draw
- IsInside
- SetTransform

O método *Draw* recebe como parâmetro um contexto gráfico (***Graphics2D***), onde o elemento será desenhado. Como o elemento responsável por conter os elementos gráficos no projeto é o Panel (seção 3.1.1), ele é compartilhado entre todas as entidades desse pacote. Note que se houver outra interface que estenda o *Graphics2D* sobrescrevendo os métodos de visualização, para que os elementos sejam impressos ao invés de desenhados na tela do computador, teríamos uma versão impressa do projeto, dada a maleabilidade da aplicação. Este método permite que tanto figuras como elementos vetoriais sejam desenhados; como não armazenamos informações geométricas do desenho, ele deve ser criado inteiramente dentro deste método. Esta decisão de projeto foi tomada devido ao fato de: se o objeto for muito complexo e possuir muitos detalhes, criamos uma imagem para representar o elemento; se for simples, com poucos comandos definimos um gráfico vetorial ou criamos uma imagem, evitando que mais classes fossem criadas no projeto, diminuindo a complexidade do código.

Por padrão, todo elemento é desenhado como sendo um quadrado com as dimensões do objeto; isso, além de servir de exemplo ao usuário, permite que erros, como a falta de representação gráfica de um componente, sejam claros quando instanciados no *Panel*. Por mais complexo que o elemento seja sempre há uma forma

de selecioná-lo; como objetos podem conter gráficos ou meios de seleção complexos, criamos o método ***IsInside*** que pode ser sobrecarregado para cada elemento.

O método *IsInside* recebe como parâmetro um ponto vindo do clique do mouse sobre o Panel. Por padrão, o objeto é selecionado se ponto recebido estiver sobre o quadrado formado pelo canto superior esquerdo e o canto inferior direito, baseado em seu tamanho, que é um atributo da classe GEquipamento. Para uma lógica de seleção mais complexa, basta sobrecarregar o método e processar o ponto recebido por algum critério interno ao elemento.

Todo elemento que esteja ou não selecionado pode sofrer transformações, como por exemplo, uma rotação de 90 graus para a esquerda ou zoom global, respectivamente. O método SetTransform indica qual será a transformação utilizada antes do elemento ser exibido. Por padrão, a transformação inicial é a identidade, que não altera os dados da visualização do objeto. A cada interação com o usuário, essa matriz é aumenta, movendo, rotacionando ou realizando escala sobre o elemento gráfico.

3.1.6 View – Graph

Assim como o pacote acima, este pacote também contém a representação gráfica dos elementos, não como aqueles que representam o *ModelEquipamento* como descrito anteriormente, mas sim do Grafo, como, por exemplo, uma aresta.

Há dois elementos que representam as conexões mapeadas no Model (matriz de adjacência) (seção 2.1): GConexao e GPorta. Ambas estendem a interface Graphic definida apenas nesse pacote; esta interface foi criada de modo a permitir que qualquer tipo de elemento que venha a ser representado sobre o grafo, pudesse seguir um padrão de desenvolvimento; ela possui dois métodos:

- Draw
- IsInside

Como não queríamos misturar os conceitos, dividimos os elementos visuais naqueles que representam um modelo e aqueles que estão relacionados diretamente

com o grafo. Além disso, os elementos que representam um Modelo de certa forma possuem uma mesma estrutura gráfica: como cada componente possui uma dimensão inicial (altura e largura), pode-se criar um elemento gráfico que seja um quadrado com tais dimensões, sendo alterado pelo usuário assim como queira. Os elementos relacionados ao grafo, não possuem estruturas semelhantes, de modo que devem ser criadas no pelo usuário assim que são adicionadas ao projeto; devido a este fato, optamos por criar uma interface.

O método ***Draw*** funciona exatamente como o descrito acima, sendo que não há uma estrutura que armazena as informações geométricas, ou seja, o desenho deve ser criado dentro do método. O método ***IsInside*** depende diretamente do tipo de objeto que esta sendo manipulado e, contém uma lógica mais complexa para o elemento GConexao, como descrito abaixo.

O elemento GPorta representa uma porta (seção 4.2.6.1) de um vértice; a lógica de desenho e seleção funciona semelhante ao descrito sobre os elementos em (View – Equipamento (seção 3.1.5), dado que apenas suas dimensões mudam.

O elemento GConexao, é representado como um fio que liga dois equipamentos. Como podemos “quebrar” esse fio, criando vários segmentos, de modo a ficar visualmente apropriado, o desenho é formado por um conjunto de segmentos de reta que se intersectam, partindo de um elemento à outro. A seleção procura o provável segmento da conexão que foi selecionado, determinando o coeficiente angular da reta, criando um retângulo com esta inclinação, verificando se o ponto passado como parâmetro (clique do mouse sobre o Panel) esta contido dentro dele; a espessura do retângulo e comprimento podem ser ajustados no código, permitindo um ajuste fino da precisão, adaptando-se ao tipo da aplicação.

4.Desenvolvimento do Projeto

O projeto está divido logicamente em pacotes: cada pacote é responsável por realizar um conjunto de operações semelhantes; conceitualmente poderiam ser distribuídos sobre o mesmo pacote, porém, além de dificultar a interpretação das operações, poderia causar confusão, dado o número de classes elevado. Os pacotes foram divididos da seguinte maneira:

- IO
- MVC (seção 3)
    - Model
        - Equipamento
            - Persistence
        - Graph
    - View
    - Controller

Alguns pacotes possuem uma sub-hierarquia (Model e View); Conforme desenvolvemos o digrama **UML** *(Unified Modeling Language*) sentimos a necessidade de especializar alguns pacotes mais que outros, devido a sua complexidade de representação e estrutura de dados, como será discutido mais a frente.

A UML é uma linguagem que foi projetada para ser facilmente entendida pelo ser humano (linguagem de terceira geração) auxiliando a visualizar as classes que compõe um sistema e a as suas inter-relações, através de diagramas padronizados, além de expressar semântica entre os objetos que se relacionam.

O usuário, seguindo o diagrama UML e navegando entre os pacotes, pode herdar das classes base os métodos e atributos necessários para compor sua aplicação, de modo a criar sua própria visualização dos dados, estrutura de dados, etc.

Como o projeto foi desenvolvido sendo o mais genérico possível, na medida em que desenvolvíamos os pacotes, analisávamos o quão custoso seria para o usuário sua extensão.

Abaixo discutiremos com maiores detalhes cada um destes pacotes.

4.1 Input Output

O pacote IO é responsável por mapear os elementos salvos em disco para o projeto ou, do projeto para o disco, através de um **XML** (*Extensible Markup Language*) ; o XML foi desenvolvido pela **W3C** (*World Wide Web Consortium*) em meados de 1990 (http://www.w3.org/TR/REC-xml/), combinando **SGML**(*Standard Generalized Markup Language)* e **HTML**(*HyperText Markup Language)* de modo a permitir que dados e suas propriedades pudessem ser lidos por software e compartilhados pela internet. Um XML armazena uma estrutura hierárquica de nós, sub-nós e dados, sendo que cada um pode conter características próprias, descrevendo-os (metadados).

Há dois modos de leitura do XML:

- DOM
- SAX

DOM cria uma estrutura em forma de árvore na memória contendo todo o documento em disco; SAX acessa os dados do XML seqüencialmente. O problema do DOM é o gasto de memória visto que todo elemento é armazenado, porém permite que os elementos sejam acessados randomicamente; Devido ao acesso seqüencial do SAX uma busca por dados não pode ser realizada, porém há pouco gasto de memória envolvida nas operações[1]. Como o arquivo de configuração do projeto é pequeno (possui poucos elementos e propriedades) e o arquivo de dados não tende a exceder em média a 500 elementos, optamos por utilizar o DOM. Para interpretar os dados presentes no XML, necessita-se que o documento esteja estruturado segundo um conjunto de regras pré-definidas: um arquivo **DTD** (*Document Type Definition*) define quais são as regras para um arquivo XML poder ser interpretado por algum leitor, segundo as regras da aplicação; distribuindo-se ambos o DTD e o XML pode-se validar

[1] (outras informações sobre estes dois modelos podem ser obtidas em *http://www.w3schools.com/).*

um documento, criando uma estrutura que pode ser compartilhada pela rede ou por aplicações que utilizem o mesmo padrão.

Há duas classes específicas neste pacote: *ListaXml* e *LoadData.* A primeira é responsável por ler o arquivo XML e criar um vetor com os nós e suas propriedades. O XML do arquivo de dados define quais são os tipos de objetos que podem ser inseridos pelo usuário; Objetos do mesmo tipo, porém com propriedades distintas podem ser agrupados dentro do mesmo nó. Cada nó deste é representado no projeto como sendo uma sub-árvore dentro de uma árvore principal, representando a hierarquia dos dados. A segunda classe é responsável por processar esse vetor de dados e criar esta estrutura de árvore, como pode ser visto na figura abaixo:

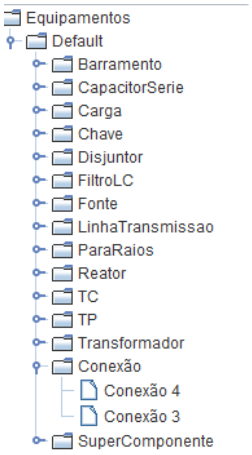


Figura 11: Arvore criada após processamento;
Note que o nó ***Conexão***, possui dois sub-nós.

O XML é construído utilizando a biblioteca XStream (http://xstream.codehaus.org/); esta biblioteca além de ser livre, possui uma interface bem simples de ser utilizada, possuindo documentação e exemplos on-line para referência. Como ela permite a utilização de um driver DOM sendo otimizada para processar arquivos com este modelo, optamos por utilizá-la na leitura de qualquer XML que possa ser interpretado pelo projeto.

O projeto trabalha com dois arquivos XML: o primeiro descreve quais são os objetos que podemos criar graficamente (Tabela 1) e outro contém a estrutura topológica que foi criada (arquivo de dados, com todos os vértices e ligações) (Tabela 2). Além disso, pode-se utilizar o DTD para validar o documento de dados, caso o usuário queira criá-lo antes de iniciar o programa.

O arquivo de objetos possui a seguinte estrutura:

```
<listaEquipamentos>
        <Equimento prop1: "" prop2: "" .../>
            <name> Nome objeto </name>
                        <Equipamento prop1: "" prop2: "" .../>
            <name> Nome objeto </name>
        .....
        .....
</listaEquipamentos>
```

Listagem 1: Listagem do arquivo XML de elementos.

O elemento raiz (Root), chamado *listaEquipamentos,* contém uma seqüência de nós internos denominados *Equipamento;* cada equipamento inserido no documento possui uma série de propriedades que podem ser definidas pelo usuário: cada propriedade corresponde a, por exemplo, um atributo da classe Equipamento; após o XML ser lido percorre-se cada nó; um nó deve conter obrigatoriamente um campo chamado *Name* que será seu identificador na árvore de objetos manipuláveis pelo usuário. Cada elemento pertencente à árvore pode ser arrastado para o painel, de modo a ser representado graficamente, permitindo interação com os outros elementos já inseridos. A hierarquia dos elementos no XML é respeitada na árvore de elementos do sistema (seção 4.1.2), ou seja, elementos contidos em um mesmo nó do XML serão representados no mesmo nível lógico na árvore do sistema; essa decisão de projeto foi tomada de modo a permitir que elementos com algum relacionamento pudessem ser agrupados.

O arquivo de dados (salvo/lido pelo usuário) contém uma estrutura mais complexa, pois deve conter cada propriedade do objeto e todas as alterações

realizadas pelo usuário. Além das propriedades gráficas, os atributos do modelo devem ser armazenados, pois podem ter sofrido alteração. Como campos obrigatórios, um arquivo de dados deve conter o modelo a que pertence, a lista de elementos inseridos, as conexões e as propriedades gráficas do ambiente, como, por exemplo, cor de fundo do painel, nível de zoom, etc. Na tabela abaixo, um exemplo de arquivo de dados; note que o usuário pode especificar como ocorre a leitura e quais são os dados que são necessários ser instanciados na criação dos elementos.

```
<casoHydra titulo="Exemplo de caso.">
  <descricao>
    Exemplo de caso.
    A descrição pode ser um texto com várias linhas para documentação geral.
  </descricao>
  <root> <!-- Equipamento root -->
    <topology>
      <!-- Todos os equipamentos são derivados de Vertex -->
      <fonte id="G01"><graphic x="0.0" y="0.0" corLinha="#FFFFFF" corFundo="#FF0000" /></fonte>
      <transformador id="aaxx0011" modelo="ABB.XYZ"><graphic x="0.0" y="1.0" corLinha="#FFFFFF" corFundo="#FF0000" /></transformador>
      <disjuntor id="zzX03" modelo="Siemens.ABC" posicao="aberta"><graphic x="0.0" y="2.0" corLinha="#FFFFFF" corFundo="#000000" /></disjuntor>
      <subsystem id="SUB01">
        <graphic x="0.0" y="3.0" corLinha="#00FF00" corFundo="#000000" />
        <topology>
          <!-- mesma forma... -->
          <port id="EXT1" portExt="0"><graphic x="0.0" y="0.0" /></port> <!-- utiliza coordenadas relativas do subsistema -->
        </topology>
      </subsystem>
      <carga id="C01"><graphic x="0.0" y="5.0" corLinha="#FFFFFF" corFundo="#FF0000" /></carga>
      <edge id="01" vertexDe="G01" portDe="0" vertexPara="aaxx0011" portPara="0"><graphic corLinha="#0000FF" estiloLinha="continua" /></edge>
      <edge id="02" vertexDe="aaxx0011" portDe="1" vertexPara="C01" portPara="0"><graphic corLinha="#0000FF" estiloLinha="tracejada" /></edge>
    </topology>
  </root>
</casoHydra>
```

Listagem 2: O arquivo de dados contém três elementos (*fonte*, *transformador* e *disjuntor*); as conexões podem ser vistas nas ultimas linhas (*edge*).

Do mesmo modo como o arquivo é lido, sua escrita ocorre através do XStream. Cada atributo ou classe que deve ser mapeada no XML possui uma "*annotation*"; *annotation* é um recurso do Java (a partir da versão 1.5) permitindo que metadados sejam inseridos no código, de modo a serem interpretados por um compilador ou pré-compilador posteriormente. No caso do XStream, as *annotations* são utilizadas para dizer o que mapear em disco ou o que recuperar do disco. Uma vez selecionado o que se quer salvar, basta utilizar o XStream pedindo para que processe as *annotations* presentes no código; para isso é preciso criar uma vetor de classes onde existem os *annotations* (as classes em Java podem ser recuperadas através do "*.class*", como por exemplo, *minhaClasse.class*), passando com referência para o método "*processAnnotations(Class[] annotations)*".

O usuário pode estender as classes base descritas acima, adicionando suas configurações, além de também adicionar novas “anotações” no código, atendendo as necessidades da sua aplicação. Como o intuito do projeto é tornar a aplicação o mais genérica possível, pensamos em na maneira mais simples de atender o usuário. Assim, não é necessário o usuário trabalhar diretamente com esta biblioteca: apenas mudando as *annotations* pode-se criar uma nova aplicação e novos arquivos de configuração e de dados.

4.2 Model

O *Model* é responsável por toda a lógica de negócio da aplicação. Ele contém os atributos e métodos responsáveis por manipular a topologia, salvar e abrir um caso, criar *Redo* e *Undo*, etc. Como ele esta inserido no MVC, ele estende o *AbstractModel* (seção 3), podendo ser registrado no *Controller*; por decisão de projeto, adotamos permanecer com apenas um único modelo em todo o sistema, podendo existir diversas topologias, como será esclarecido mais a frente.

O modelo (*Model*) foi criado para ser inserido no padrão do MVC, concentrando diversas operações sobre as classes:

- Topology
- Redo Undo
- GI

Essas três classes representam o núcleo da lógica do sistema, pois permitem que a topologia seja manipulada, que se possa navegar no tempo e que possamos interagir com o usuário, respectivamente. Logicamente, por ser uma classe complexa, ela possui diversos métodos e atributos de controle, utilizados, porém, para controlar as três classes principais; devido a este fato, nos concentraremos em descrevê-las a fundo, deixando sob responsabilidade de o usuário ler a documentação do projeto para obter detalhes menos relevantes.

4.2.1 Topology

A classe *Topology* representa a topologia do sistema. Há duas formas básicas de se representar um grafo (seção 2.1): lista de adjacência e matriz de adjacência. Por decisão de projeto, optamos por utilizar listas de adjacência: além de economizar memória, facilita a programação de alguns algoritmos clássicos (seção 2.1).

Optamos por uma variação de representação das listas de adjacências: Armazenamos cada vértice da estrutura em um *HashTable*; um *HashTable* é uma estrutura que utiliza uma função *Hash* para identificar um elemento em uma estrutura, como por exemplo em um vetor[4] retornando um índice onde ele foi armazenado ou deve ser procurado. Como cada vértice possui uma descrição única no sistema (seção 4.2.6), isso aumenta o desempenho em, por exemplo, uma busca por um determinado vértice.

As conexões são armazenadas em um vetor de Edges. Como cada Edge possui duas portas que determinam quais são seus pais, fica simples percorrer a lista de elementos e possuir as conexões com outros elementos.

Muitos métodos já estão processando a topologia, como por exemplo, teste de conexão, lista de caminhos de um vértice a outro, por quais portas estão conectados dois vértices, etc. O usuário pode estender essa estrutura, herdando a classe Topology ou, inserindo seus métodos na classe já criada. Por uma questão de consistência e segurança é aconselhável herdar a classe Topology, criando as estruturas auxiliares.

4.2.2 SuperComponente

Todo Vertex possui uma topologia interna: a fim de poder representar componentes com elementos internos (SuperComponente) um ponteiro para uma topologia é armazenado em cada elemento. No caso do SuperComponente essa topologia é instanciada.

Por decisão de projeto e pelo próprio padrão do MVC, mantemos apenas um modelo, que por sua vez possui uma topologia que armazena os elementos inseridos de forma global, ou seja, representa uma topologia onde não há níveis. Apesar de o SuperComponente armazenar uma topologia, o Model o enxerga como um elemento

único. No caso de projetos onde se avalia a confiabilidade de um sistema, os elementos que possuem topologia interna devem ser processados antes da avaliação do sistema como um todo. Note que o SuperComponente por ser um Vertex também possui uma topologia interna, ou seja, pode-se aninhar múltiplos níveis; isso garante que a representação de qualquer estrutura por mais complexa que seja possa ser representada sem problemas, porém, a complexidade dos algoritmos que manipulam essa estrutura em níveis deve ser capaz de aprofundar o quanto for necessário. Em geral, estruturas que dependem de uma análise em profundidade de elementos para o cálculo utilizam algoritmos recursivos, subindo de nível quando calculam a estrutura mais profunda; deve-se tomar cuidado com a complexidade desses algoritmos e das operações que utilizam, pois o tempo de processamento ou o gasto de memória podem se tornar inaceitáveis. Como a maioria dos problemas que envolvem grafos são da classe NP ou NP-Completo (SAT e Clique, por exemplo) deve-se tomar cuidado extra quando manipulados este tipo de estrutura.

4.2.3 Redo Undo

Esta classe armazena os estados da topologia permitindo explorar as ações que ocorrem no decorrer do desenvolvimento do projeto em uma seção. A topologia é armazena em um vetor de no máximo 100 posições: isso permite que o usuário volte até 100 vezes no histórico de ações.

Cada entrada do vetor possui uma topologia completa; quando efetuado o comando de desfazer (*undo - CTRL+Z*) ou refazer (*redo - CTRL+Y*) um contador percorre as posições anteriores ou posteriores refazendo a topologia previamente armazenada. Cada instância recuperada envolve trocar a topologia corrente pela obtida; isso significa limpar as visões, inserindo os dados recuperados, de modo a representar fielmente a topologia corrente. Como há uma cópia profunda da topologia a cada manipulação do usuário, um algoritmo de lista circular sobre o vetor é utilizado a cada passo, descartando as estruturas que excedem o limite de 100 posições.

A instância da classe RedoUndo é criada no Model. A cada chamada do Controller que altera a topologia, uma nova entrada é inserida no histórico de ações do usuário.

Como a classe salva toda a topologia, basta o usuário inserir as entradas corretamente no Model; o usuário pode estender a classe como desejar, porém, não é preciso, pois o processamento é simples, manipulando qualquer classe que derive do Model.

4.2.4 Graphics Interface

GI (*Graphics Interface*) é uma classe responsável por manipular os elementos gráficos que representam a topologia, realizando a interação com o usuário. Ela não altera o modelo diretamente, sendo que estas operações são delegadas ao *Controller*, como descrito em (seção 4). Aqui esta uma decisão de projeto descrita no MVC, onde uma *View* acessa diretamente o *Model*; este acesso não passa pelo *Controller*, pois nenhum dado topológico é modificado: as operações realizadas no GI modificam apenas as propriedades gráficas dos elementos; a lista de vértices e arestas da topologia é processada quando ocorre um evento de interação com o usuário, de modo a selecionar vértices, arrastá-los, quebrar arestas, rotacionar elementos, etc.

O *Panel* ao identificar um evento de mouse ou teclado invoca métodos específicos do GI, de acordo com a ação do usuário. Isso envolve uma lógica complexa, pois uma mesma ação pode significar diversos comandos: um clique de mouse pode significar a seleção de um componente, uma porta ou uma aresta caso nenhum destes estejam selecionados; se houver algum elemento selecionado, um clique pode significar uma nova conexão caso ocorra sobre a porta de outro componente ou, anulação de uma seleção anterior, por exemplo.

Uma estrutura de dados armazena o resultado sobre as operações que são realizadas pelo usuário, de forma a podermos recuperá-los quando ocorrem ações que alteram o *Model*, como por exemplo, deleção de componentes multi-selecionados, cópia, recortar e colar, etc. Em geral, qualquer ação exceto a seleção de múltiplos componentes armazena um elemento no GI, seja ele *Vertex*, *Edge* ou *Port*, de modo a passá-lo como referencia ao *Model*; no caso da seleção de múltiplos componentes, cria-se um vetor que é armazenado no GI. Caso ocorra uma cópia de elementos, uma estrutura de dados auxiliar é criada, permitindo que os elementos sejam reproduzidos

fielmente à cópia quando forem colados: como os elementos são conectados por seus *Port(s),* através de *Edges*, é necessário especificar quais são as portas que estão conectadas; uma vez identificadas, basta adicionar os vértices e ligá-los pelas portas corretas. Caso a cópia não seja realizada deste modo, pode ocorrer a troca da ordem das portas, visto que a conexão entre componentes pode ser realizada por qualquer porta.

O GI foi criado para podermos isolar completamente as operações do *View* e do *Model*. Antes ao *MVC*, todo o controle do modelo e da parte de interação com o usuário estava sobre uma única estrutura, de modo a não ser possível trabalharmos em, por exemplo, um terminal, sendo que a toda a parte de visualização de dados deveria estar instanciada. Este fato limitou o desenvolvimento do projeto; como não podíamos ficar restritos a tal modelo, dividimos completamente o que fazia parte do modelo e da visualização e interação com o usuário. Todo o controle gráfico, além de poder ser modificado pelo desenvolvedor, pode ser isolado da aplicação, permitindo que as estruturas do modelo sejam executadas sem haver instancia gráfica.

Este novo modelo permite que qualquer modo de manipulação de dados seja criado, apenas herdando esta classe, alterando seus métodos.

4.2.5 Model- Equipamento – Persistence

Este pacote contém as classes que mapeiam os equipamentos que podem ser instanciados pelo usuário.

Todo objeto que pode ser manipulado pelo usuário possui um modelo de dados que o descreve; neste contexto, ***equipamento*** pode significar uma série de conceitos: como o projeto define uma interface genérica, por decisão de projeto, definimos como equipamento todo o objeto que pode ser instanciado pelo usuário, representando um vértice na topologia.

O termo *Persistence* significa que os dados inseridos neste pacote definem as propriedades dos elementos que serão armazenados na topologia, além de posteriormente poderem ser armazenados ou recuperados do disco.

Qualquer elemento deste pacote estende a classe ModeloEquipamento, que por sua vez estende a classe Model. A classe Model possui propriedades básicas como nome do equipamento e a assinatura de um método chamado “fábrica” que retorna a instancia de um equipamento (seção 4.2.5); como descrito logo abaixo, o equipamento retornado estende a classe Vertex e contém um modelo que o representa (seção 4.2.5); o método fábrica deve ser sobrecarregado naquelas classes que estendem o Model, criando instância de equipamentos que podem ser inseridos na topologia. A classe ModeloEquipamento, como dito acima, estende a classe Model e possui outras propriedades que podem ser definidas pelo programador; deste modo mantemos a base sólida em Model, permitindo a extensão em outro nível.

Todo Modelo que representa um equipamento que pode ser manipulado pelo usuário deve estender a classe ModeloEquipamento criando a definição do método “fábrica” herdada de Model: este método, presente em todos os Modelos, contém uma instancia do modelo que representa o objeto, e do próprio equipamento; por exemplo, o equipamento FonteComputador deve conter um modelo, que por padrão de projeto deve sempre começar com a palavra modelo, ou seja, ModeloFonteComputador; uma vez que estende de ModeloEquipamento, deve-se implementar o método “fábrica” que irá retornar um Equipamento (FonteComputador) com o modelo de dados ModeloFonteComputador.

Esta representação de dados cria uma hierarquia sobre os componentes: uma vez respeitada esta hierarquia, pode-se representar qualquer equipamento, criando o modelo de dados em Model-Equipamento-Persistence e o componente em Model-Equipamento.

4.2.6 Model – Equipamento

Este pacote contém os elementos que podem ser manipulados pelo usuário. Todo equipamento estende a classe Equipamento, que por sua vez estende Vertex;

como esses elementos são inseridos na topologia, eles devem estender a classe Vertex (Model-Graph), sendo que isso ocorre através desta classe.

A classe equipamento possui algumas propriedades básicas que são utilizadas para criar o desenho, como cor de fundo, cor da linha de borda, etc., além de conter o modelo que representa o objeto. Como o método “fábrica” (seção 4.2.4 e 4.2.5) retorna um Equipamento, deve-se armazenar o Modelo de dados que representa o componente. Todo equipamento possui um elemento gráfico (seção 3.1.5): ele é utilizado para representar o equipamento graficamente quando ocorre DND (Drag and Drop) pela interação do usuário no programa.

Basicamente, um Equipamento deve conter um modelo de dados e uma classe de desenho; outros métodos podem ser inseridos, porém, se fizerem relação ao modelo de dados, devem ser inseridos diretamente em Model; propriedades que fazem sentido ambos no modelo e na parte gráfica, como por exemplo, se um Disjuntor está aberto ou fechado, podem ser armazenadas em Equipamento. Como o equipamento é armazenado na topologia, pode-se utilizar estes dados para auxiliarem em algum processamento específico, como por exemplo, no caso do disjuntor, para achar os caminhos que podem ser energizados.

Abaixo, a figura 12 ilustra o Equipamento, que contém ambos o modelo e a visualização do componente.

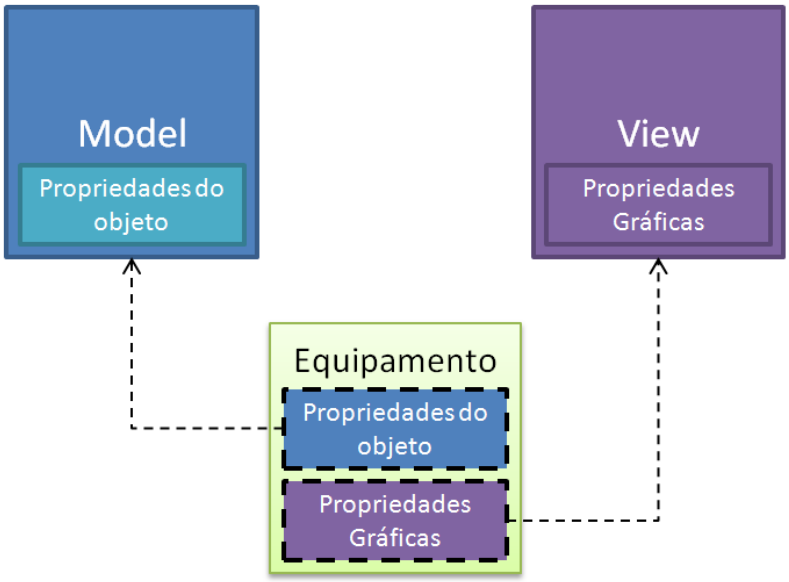


Figura 12: Representação de um Equipamento.

4.2.7 Model- Graph

Este pacote contém as classes responsáveis por manipular o grafo, como vértice, edges e etc., ou seja, elementos que compõe a topologia.

A figura abaixo (figura 13) ilustra os elementos (*Vertex*, *Edge* e *Port*) e seu relacionamento; note que um conjunto de vértices, edges e portas constituem um grafo completo. O elemento *PortVertex* é descrito em detalhes mais abaixo, pois contém uma lógica própria e é utilizado em um componente que agrupa outros (*SuperComponente*).

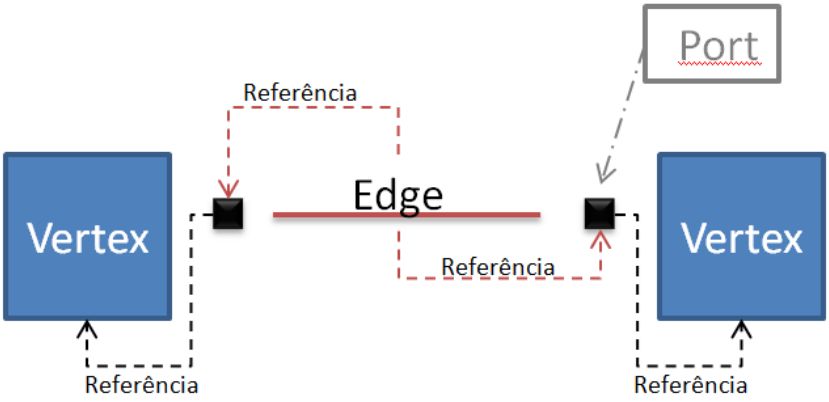


Figura 13: Relacionamento entre os elementos do Grafo.

4.2.7.1 Port

Um *Port* é um elemento que contém um ponteiro para um *Edge* e para um *Vertex*. Um *Port* é criado apenas quando um novo vértice é adicionado na estrutura. Como um *Port* contém um ponteiro para um *Edge* que liga dois vértices, só faz sentido criar um *Port* quando um vértice é instanciado.

O construtor de *Port* recebe dois parâmetros: Um *Vertex* e uma String que é utilizada como sua identificação. Não há sentido em construir um *Port* com *Vertex* nulo, como dito no parágrafo acima.

O Id do *Port* pode ser o mesmo do *Vertex* ou pode ser uma descrição única: como a Topologia não utiliza diretamente os id's dos *Port(s)*, se não faz sentido identificá-los posteriormente, pode-se iniciá-los com o mesmo id do *Vertex*; se a aplicação necessitar manipular as portas em algum processamento específico deve-se criar um Id único, que pode ser composto pelo nome do *Vertex* e um contador interno manipulado por ele.

Por padrão, o Id de um *Port* é criado com o Id e o número atual de portas do vértice; o ponteiro para *Edge* é construído como nulo.

4.2.7.2 Edge

Um *Edge* representa uma conexão entre dois elementos inseridos na topologia. Um *Edge* possui dois *Port(s)* que identificam os dois lados da ligação. Os ponteiros para os vértices também são armazenados de modo a aumentar o desempenho, não sendo necessário percorrer o caminho *Edge->Port->Vertex* a cada vez que processamos um *Edge*. Como uma porta contém um ponteiro ao seu vértice pai, acessar essa informação é trivial.

Como o número de conexões é definido pelo número de *Port(s)*, um *Vertex* pode ter no Maximo ***n*** conexões, onde ***n*** é o número de *Port(s).*

4.2.7.3 Vertex

Um vértice (*Vertex*) é um nó do grafo. Como os elementos possuem representação gráfica e um modelo que pode armazenar diversas informações, optamos por criar uma classe que o represente, de tal modo que o usuário possa estender o conceito de um vértice representado em uma lista ou matriz de adjacência, expandindo-o com informações que fazem sentido em sua aplicação.

Todo vértice contém em seu interior uma topologia (*Topology*). Esta decisão de projeto nos permite armazenar objetos que são compostos (*SuperComponente*) e possuem uma hierarquia, por exemplo: uma Fonte de Computador pode ser representada apenas como um componente do projeto, porém, interiormente ela pode conter parafusos, fios, carcaça, etc. Podemos descer nos níveis afim de realizar algum cálculo, como por exemplo a confiabilidade do elemento falhar dado pela combinação das falhas de seus constituintes, ou representar múltiplos objetos que só fazem sentidos em um contexto específico. Além da topologia interna, um vértice possui uma identificação única (id). O id é um texto determinado pelo nome do objeto se sua instância for única ou, é composto por um número seqüencial gerado pelo *Model* seguido do nome padrão, como, por exemplo: FonteComputador, FonteComputador1, etc.

Além do Id e do *Topology*, outro atributo importante é um vetor (*ArrayList*) de Port. Apesar de um vértice poder conter ***n*** ligações com outros, em algumas aplicações é interessante fixar o número de conexões que um vértice pode realizar, como por

exemplo, em um problema elétrico, um motor comum deve ter apenas duas ligações, representando os pólos positivo e negativo.

Como os vértices não contem um vetor de *Edges*, esse número aleatório de conexões não é permitido por padrão. Essas conexões são feitas através de um *Port*: se o componente estiver conectado com outros 10 elementos, deverá haver no mínimo 10 ligações, logo deve haver no mínimo 10 *Ports* no vetor. Isso permite que o número de ligações que um vértice pode fazer seja limitado pelo usuário. Logicamente é possível permitir que qualquer número de ligações seja feito entre os elementos, bastando adicionar portas no elemento, quanto for necessário.

4.2.7.4 PortVertex

PortVertex é um *Vertex* especial, utilizado na construção de um *SuperComponente*. Quando agrupamos vários elementos em um único componente é necessário identificar as conexões que podem ser realizadas com o mundo externo. Um *PortVertex* é um elemento que faz a interface entre estes mundos.

Cada *Port* que esta desconectado ou que contém uma ligação entre um elemento diferente daqueles que serão agrupados devem ser mapeados sobre o *SuperComponente*, permitindo que seja realizada uma nova conexão ou que, as que já existiam, sejam representadas.

Um *PortVertex* possui um *Port* e um link de *Port* (para o elemento externo). Quando um *Port* do elemento que será agrupado está desconectado, um *PortVertex* é criado e seu *Port* é conectado ao *Port* antes sem conexão; o link ao *Port* externo é definido como nulo. Para um *Port* conectado, a conexão deve ser armazenada e desfeita, enquanto o *PortVertex* é inserido na topologia e conectado ao *Port* do elemento interno, agora sem conexão; o link externo é definido como o elemento externo à conexão armazenada.

A figura abaixo (figura 14), exibe os dois casos da criação do *PortVertex*. Em A, há cinco elementos, sendo que V3, V4 e V5 serão agrupados, como pode ser visto em B. Como V3 e V4 contêm uma ligação com o mundo externo, as conexões C1 e C2 serão

destruídas, construindo os links LK1 e LK2 e as conexões C3 e C4. No caso do elemento V5 as conexões C4 e C5 serão construídas; como não há link com o mundo externo nada mais será criado. Em C, pode-se ver a representação gráfica com os elementos em seu interior, em D, o *SuperComponente* (*SC1*), sendo uma caixa preta.

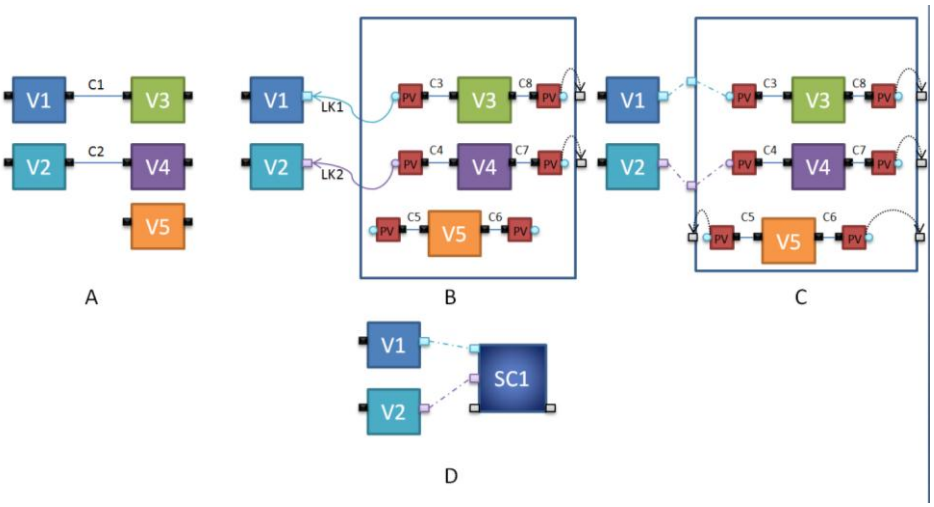


Figura 14: *PortVertex* e sua relação com o *SuperComponente*.

5 Interface Gráfica

A interface gráfica foi desenvolvida pensando na extensão do código e na sua adaptação pelo desenvolvedor, de modo a satisfazer os requisitos da aplicação. Basicamente esta dividia em seis áreas visuais: Painel (1), Lista de Equipamentos (2), Lista de Equipamentos inseridos (3), Log (4), barra de botões (5) e barra de menus (6) (figura 15).

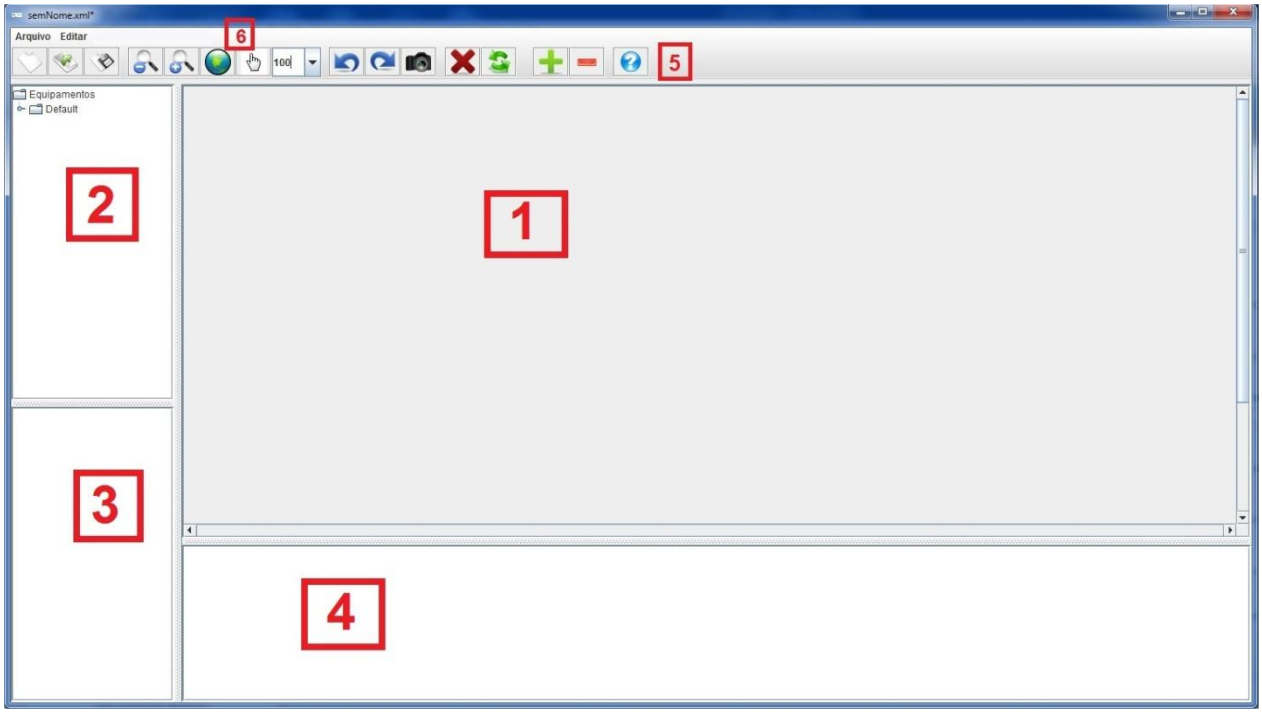


Figura 15: Divisão da interface gráfica.

A Área (1) contém a representação gráfica dos elementos inseridos na topologia, permitindo manipulá-los (movimento, rotação, deleção, etc.). Os elementos são arrastados da área de equipamentos (2) e soltos em (1), sendo inseridos na topologia. Automaticamente ao inserir um elemento em (2), as áreas (3) e (4) são atualizadas, de modo a conter uma árvore de elementos já inseridos e um log de atividades realizadas, respectivamente. As barras de botões e menus, alteram a visualização como um todo, seja executando algum algoritmo sobre a topologia ou alterando as propriedades visuais de elementos individualmente ou de todo o Painel.

Além dos menus básicos encontrados na maioria dos programas, como “Arquivo” e “Editar”, disponibilizamos algumas ferramentas em forma de botões (5), de modo que todas as interfaces derivadas desta possam aproveitar tais recursos. Os menus Arquivo e Editar (1) possuem as operações *Novo, Abrir, Salvar, Salvar como, Sair* e *Desfazer, Refazer, Opções.,* respectivamente. Essas operações são básicas em qualquer programa e podem ser aproveitadas em qualquer aplicação.

Como sempre salvamos e carregamos uma topologia de um XML, as operações *Abrir* e *Salvar* ficam triviais, dado que estes métodos já estão criados na estrutura. *Desfazer* e *Refazer* são operações que carregam uma topologia previamente salva na estrutura do **Undo** e **Redo**. O menu *Opções* permite que uma lista de equipamentos seja inserida no sistema ou permite configurar o painel principal (1) estabelecendo altura e largura mínimas. Pode-se salvar uma imagem completa do Painel, no formato *.jpg* ou *.png*, caso o usuário deseje compartilhar a visualização ou mesmo fazer um relatório de suas atividades.

Qualquer alteração que o usuário queria executar, como adicionar ou botões, pode ser feita em Canvas (seção 4.1.5). Todos os menus são criados utilizando a API swing do javax (*http://docs.oracle.com/javase/tutorial/uiswing/*). Como todos os elementos gráficos derivam de uma classe base do Java, pode-se trocar os componentes visuais que se mantenha a interface descrita em View (seção 4.1); além disso, qualquer alteração em menus ou barra de botões será gerenciada automaticamente pelo Java, visto que os elementos apenas são extensões delas.

Como utilizamos o MVC, qualquer alteração topológica em alguma das visualizações, será refletida no Model que por sua vez atualizará todas as outras View’s. Note que como cada View possui um modo distinto de exibição dos elementos, as mudanças visuais que podem ocorrer localmente em uma View não são compartilhadas como as outras, pois isto é uma característica particular de cada uma; por exemplo, uma rotação de um elemento no painel principal não irá alterar o elemento na árvore de elementos inseridos, visto que este exibe apenas um texto que é o identificador do elemento na estrutura topológica.

6 Adaptação do código para um sistema de estudo de subestações, o projeto HYDRA.

6.1 O projeto e seus integrantes

O projeto Hydra[22] é uma parceria entre o Cepel (Centro de Pesquisas de Energia Elétrica) a UERJ (Universidade Estadual do Rio de Janeiro) e o ICMC-USP (Instituto de Ciências Matemáticas e de Computação) que visa à construção de um sistema para análise de confiabilidade de subestações. Contou com a participação dos professores doutores Antonio Castelo Filho (ICMC) e Norberto Mangiavacchi (UERJ), os pesquisadores Carlos Kleber C. Arruda (Cepel) e Pablo de Abreu Lisboa (Cepel), os alunos Luiz Carlos Lucca, Hugo Checo Silva e Raama Costa que na época estavam cursando respectivamente: mestrado em Ciência da Computação, ICMC-USP, mestrado em Engenharia Mecânica, UERJ e graduação em Engenharia Mecânica, UERJ. Os professores gerenciaram as equipes na cidade de São Carlos e Rio de Janeiro respectivamente.

Os pesquisadores Pablo e Carlos foram responsáveis por mapear as propriedades físicas da subestação, assim como os modelos que representam os equipamentos. Os alunos Hugo e Raama, ficaram responsáveis por estudar o caso e propor soluções para alguns problemas que não eram triviais de resolver, assim como trabalhar no desenvolvimento da leitura e escrita do XML, do gerenciamento das ações do usuário (redo/undo) e criação de alguns relatórios; o aluno Luiz ficou responsável por estender as classes do projeto GGraph, portando o código para as necessidades do projeto. Todos participaram das atividades do grupo, de modo que muitas idéias foram incorporadas no projeto durante o desenvolvimento, ou seja, apesar de cada um desenvolver atividades particulares, houve grande entrosamento da equipe, permitindo que novas funcionalidades fossem inseridas no projeto, de modo a torná-lo completo e usual.

Uma subestação é formada por um conjunto de equipamentos interligados. Uma fonte alimenta o sistema e uma carga consome a energia, que também pode ter o

sentido de distribuição para uma cidade, por exemplo. O problema consiste em analisar a confiabilidade da subestação, avaliando o risco de contingências (problemas elétricos) que podem se alastrar pela rede (contingência simples, dupla, tripla, etc..) ou afetar apenas um equipamento (contingencia simples).

Alguns problemas que surgem da natureza do projeto são:

- Dado um conjunto de nós específicos (cargas e fontes) pode-se, por exemplo saber quais serão todos os caminhos que a energia pode percorrer;
- Dado todos os caminhos, quais são os percursos mais críticos, ou seja, aqueles em que há maior taxa de contingências;
- Podem-se reorganizar os elementos de modo a melhorar a confiabilidade do sistema, reduzindo número de falhas que podem ocorrer;
- Existem equipamentos isolados que não contem ligação com nenhum outro equipamento (este é um erro comum que pode ser encontrado quando editamos grafos muito grandes.)
- Há redundância na rede, de modo a garantir o suprimento de energia quando ocorrer manutenção de um equipamento?

Note que mapeando os equipamentos como vértices e as ligações como arestas de um grado, pode-se obter facilmente as respostas para algumas destas perguntas. Alguns problemas necessitam de um processamento extra, porém, podem-se combinar diversos algoritmos citados na literatura[4] para obter uma resposta aceitável, seja por heurísticas, estatística ou força bruta.

Por este motivo, decidimos portar o código original para o Hydra, dado que ele é um problema da natureza de grafos.

6.2 Portabilidade do código para o sistema Hydra

Todo código original foi portado para o sistema Hydra, que pode ser utilizado como referência para outros problemas de mesma natureza. Abaixo segue a descrição dos pacotes IO, Model, MVC e VIEW, além da descrição da interface gráfica.

6.2.1 HydraIO

Estende a classe base IO (seção 4.1). Assim como no IO, há dois tipos de XML que são criados na aplicação. O primeiro mapeia os elementos que serão carregados na árvore de equipamentos, preenchendo as características básicas do modelo que os representa. Os elementos mapeados inicialmente devem seguir a estrutura:

- Raiz *<listaEquipamentos>;*
    - em seguida vem um novo nó que descreve o equipamento, seguido das propriedades do Modelo, como por exemplo *<Barramento localizacao="EXTERNO" anoFabricacao="0" fabricante="" modelo="Teste" frequencia="0.0" potenciaNominal="0.0";*
    - segue o nome de exibição na árvore *<name>Barramento externo</name>* e depois, fecha-se o nó criado *</Barramento>*. Se mais de um nó for criado com o mesmo identificador, no caso *Barramento*, teremos na árvore de equipamento a identificação *Barramento* seguido dos nomes.
- Fecha a raiz </listaEquipamentos>

A tabela e a figura abaixo ilustram o caso onde há dois Barramentos, uma carga, etc. A tabela 3 ilustra o arquivo que descreve os equipamentos; a figura 16 mostra a disposição na árvore de equipamentos.

```
<listaEquipamentos>
  <Barramento localizacao="EXTERNO" anoFabricacao="0" fabricante="" modelo="Teste" frequencia="0.0" potenciaNominal="0.0">
    <name>Barramento externo</name>
  </Barramento>
  <Barramento localizacao="INTERNO" anoFabricacao="0" fabricante="" modelo="" frequencia="0.0" potenciaNominal="0.0">
    <name>Barramento blindado SF6</name>
  </Barramento>
  <CapacitorSerie localizacao="EXTERNO" anoFabricacao="1983" fabricante="HITACHI" modelo="X1" frequencia="60.0" potenciaNominal="200.0">
    <name>CS 500 kV</name>
  </CapacitorSerie>
  <Carga localizacao="INTERNO" anoFabricacao="1983" fabricante="HITACHI" modelo="X1" frequencia="60.0" potenciaNominal="200.0">
    <name>Carga</name>
  </Carga>
```

Listagem (**3**): XML da lista de equipamentos

Figura 16: Dois barramentos dentro do nó *Barramento: Barramento Externo e Barramento blindado SF6*

O outro, que salva o projeto em disco, mapeia os dados de cada equipamento, ou seja, do Model de cada Equipamento. Além dos objetos que foram instanciados serem salvos, é preciso salvar as características gráficas de cada elemento como rotação, cor, posição, etc., além da configuração geral do painel, como nível de zoom e as conexões entre os elementos.

A tabela abaixo ilustra alguns trechos de um arquivo salvo em disco, com dois barramentos conectados.

```
<vertex class="cepel.hydra.model.equipamento.Barramento">
                <...>
                  <id>Barramento externo_1</id>
                  <vertex class="cepel.hydra.model.equipamento.Barramento"
reference="../../.."/>
                  <...>
                  <equipamentoPai
class="cepel.hydra.model.equipamento.Barramento" reference="../../.."/>
                <…>
                 </cepel.hydra.model.Porta>
                </listOfports>
                <id>Barramento externo</id>
                <topology>
                  <contReg>0</contReg>
                  <vertex/>
                  <edge/>
                </topology>
                <model
class="cepel.hydra.model.equipamento.persistence.ModeloBarramento">
                  <name>Barramento externo</name>
                  <...>
              </vertex>
```

Listagem 4: Estrutura de um projeto salvo em disco; ilustra as propriedades de um barramento de nome Barramento externo 1 que esta conectado com outro barramento, denominado Barramento externo (não representado). As tags **<...>** representam informações que foram retiradas, afim de diminuir a listagem.

6.2.2 HydraModel

Os pacotes em Model não foram alterados, apenas copiamos e alteramos as classes de modo a acrescentar o prefixo Hydra em todos as classes do pacote, de modo a tornar, por exemplo, GI como HydraGI, Topology como HydraTopology, etc.

Vale destacar dois pacotes do Model: o Hydra-Model-Equipamento e o Hydra-Model-Equipamento-Persistence.

O Hydra-Model-Equipamento contém todos os equipamentos que fazem parte do sistema. Assim como descrito em Model (seção 4.2) todo equipamento em Hydra-Model-Equipamento estende a classe Equipamento deste mesmo pacote. Como todo equipamento possui um modelo, e o modelo é transferido em um DND, é necessário criar um equipamento antes de criar o modelo que o descreve. Os equipamentos mapeados no sistema são:

- Barramento
- CapacitorSerie
- Carga
- Chave
- Chaveador
- CompensadorEstatico
- Disjuntor
- FiltroLC
- Fonte
- Generico3
- Generico4
- LinhaTransmissao
- ParaRaios
- Reator
- TC
- TP

- Terminal
- Transformador

Além das propriedades herdades de equipamento, alguns deles apresentam características próprias, como o Transformador, que contém métodos que alteram a tensão de operação. Os elementos Conexão3 e Conexão4 são componentes de ligação, ou seja, podem ligar 3 ou 4 equipamentos respectivamente; como também podem apresentar falha, foram mapeados como equipamentos.

O Hydra-Model-Equipamento-Persistence mapeia os equipamentos em modelos, que são transferidos e manipulados pelo usuário. Para cada equipamento existe um Modelo, de modo a termos pares: (Barramento, ModeloBarramento), (Carga, ModeloCarga), etc. Seguindo este padrão se torna simples saber em qual pacote estamos trabalhando além de simplificar a navegação entre as classes do projeto.

6.2.3 HydraMVC

O pacote MVC continua o mesmo. Apenas copiamos e mudamos os nomes das classes. Devido ao fato de ser genérico, não necessitamos efetuar nenhuma mudança no código.

Apenas inserimos alguns métodos no Controller, assim como atributos, de modo a mapear os métodos do Model que podem ser invocados através do controlador: os atributos são strings que identificam os métodos criados no Model. Como os objetos efetuam mudanças via controlador, foi necessário criar métodos que recebessem uma nova instância de algum objeto ou, alguma propriedade que deveria ser alterada.

Estendemos a classe *AbstractController* (*HydraController*), especializando ele para esta aplicação. Basicamente, registramos os métodos contidos no *Model*. Como há uma grande interação com o usuário na parte gráfica (*View*), criamos alguns métodos no controlador para facilitar que dados fossem inseridos no *Model*; grande parte desses métodos é responsável por manipular os elementos da topologia. O diagrama de classes utilizado no *Hydra* pode ser visto no Anexo 9.

6.2.4 HydraView

Houve poucas mudanças no pacote View devido ao fato de o Panel o Log e as árvores estarem prontas para manipular qualquer objeto que fosse derivado de Model; como se seguiu a hierarquia de desenvolvimento proposta na seção 3.1, as maiores mudanças ocorreram no pacote Hydra-View-Equipamento, onde as instâncias gráficas de cada equipamento foram criadas.

Assim, os elementos mapeados em HydraModel, foram mapeados neste pacote, seguinda a nomenclatura G*Equipamento*, onde *Equipamento* foi substituído pelo nome do respectivo modelo; deste modo, criou-se os pares (Transformador, GTransformador), (Carga, GCarga), etc. Em alguns equipamentos, como o Barramento, foram inseridos métodos que controlam, por exemplo, o crescimento vertical: equipamentos específicos como o Barramento, devem permitir que seu tamanho vertical seja alterado, de modo a representar uma barra de ferro onde os elementos podem ser conectados. Note que qualquer propriedade gráfica que pode ser manipulada deve ser inserida na classe gráfica; deste modo, elementos com propriedades particulares podem ser mapeados sem maiores problemas, bastando apenas criar um ou mais métodos que alterem as propriedades dentro da classe deste pacote.

Abaixo, segue uma lista de figuras que ilustra cada equipamento que pode ser manipulado pelo usuário. Os gráficos foram criados reescrevendo o método draw, herdado da classe GEquipamento.

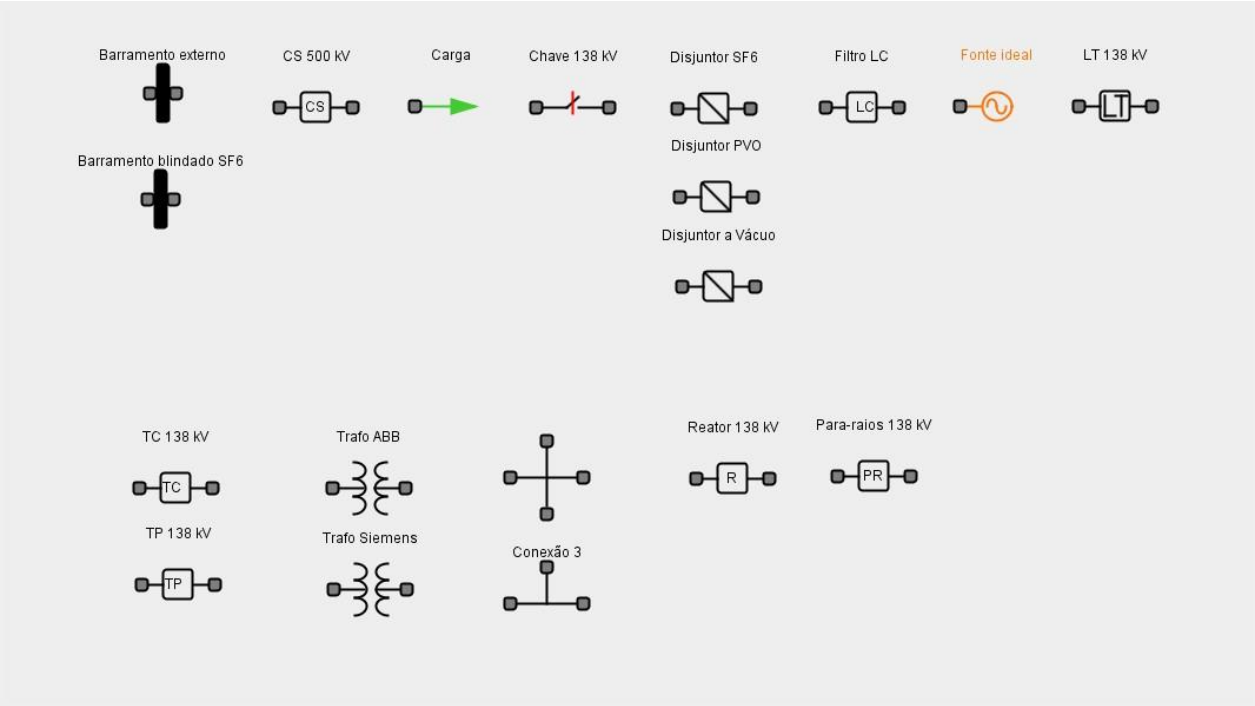


Figura 17: Equipamentos criados para o sistema Hydra. Note que pode haver várias instancias de um mesmo componente, bastando criar uma classe que o represente.

7 Interface Gráfica do Hydra

A interface gráfica do Hydra herdou todas as características do projeto original. Apenas acrescentamos ferramentas que faziam sentido no contexto de engenharia elétrica e confiabilidade.

Figura 18: Menu de Equipamento.

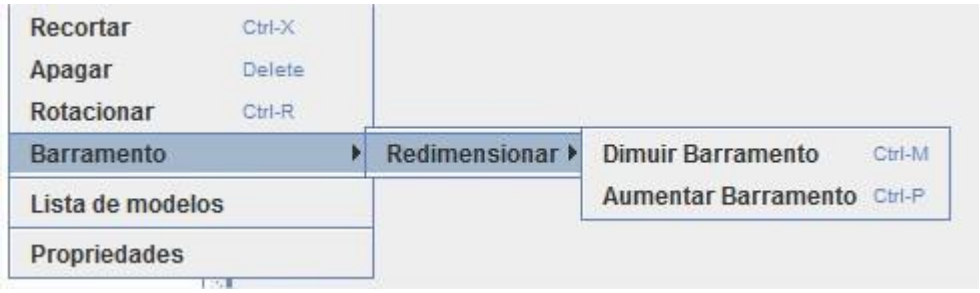


Figura 19: Menu Barramento adicionado, além de sub-menus, como redimensionar, diminuir barramento e aumentar barramento.

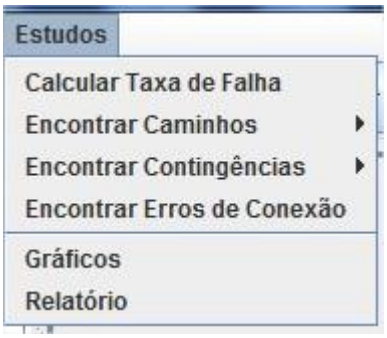


Figura 20: Menu estudos adicionado na barra de menu principal

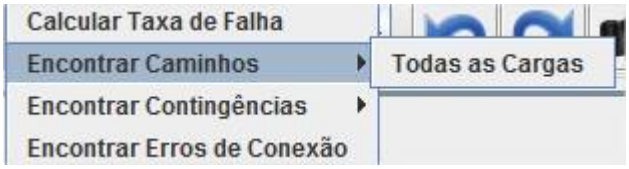


Figura 21: Sub-menu encontrar caminhos dentro do menu estudos.

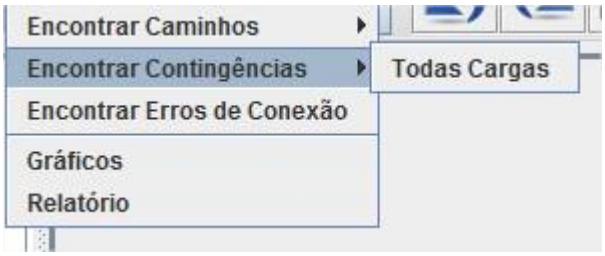


Figura 22: Sub-menu encontrar contingências dentro do menu estudos.

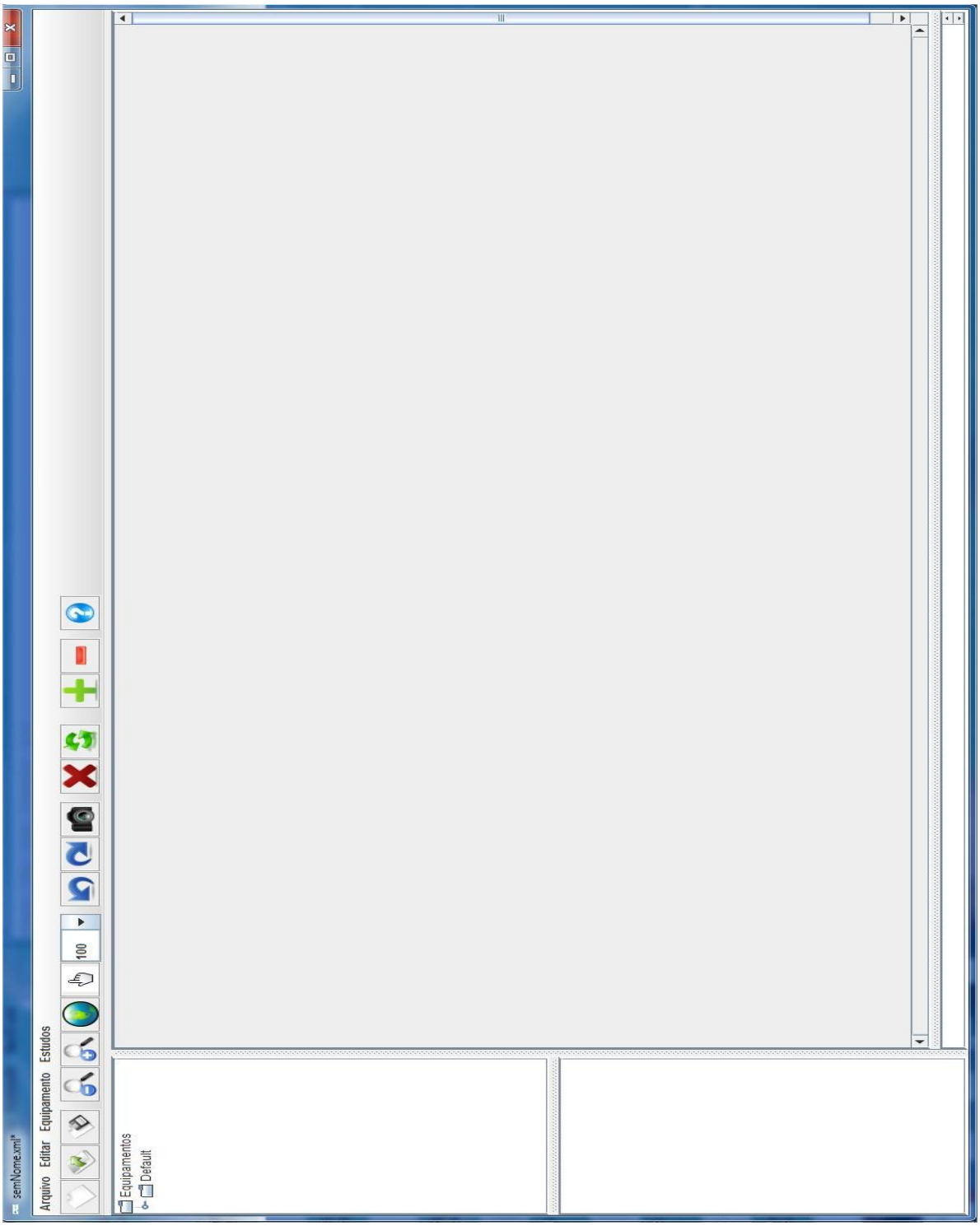


Figura 23: Interface Inicial do Sistema Hydra.

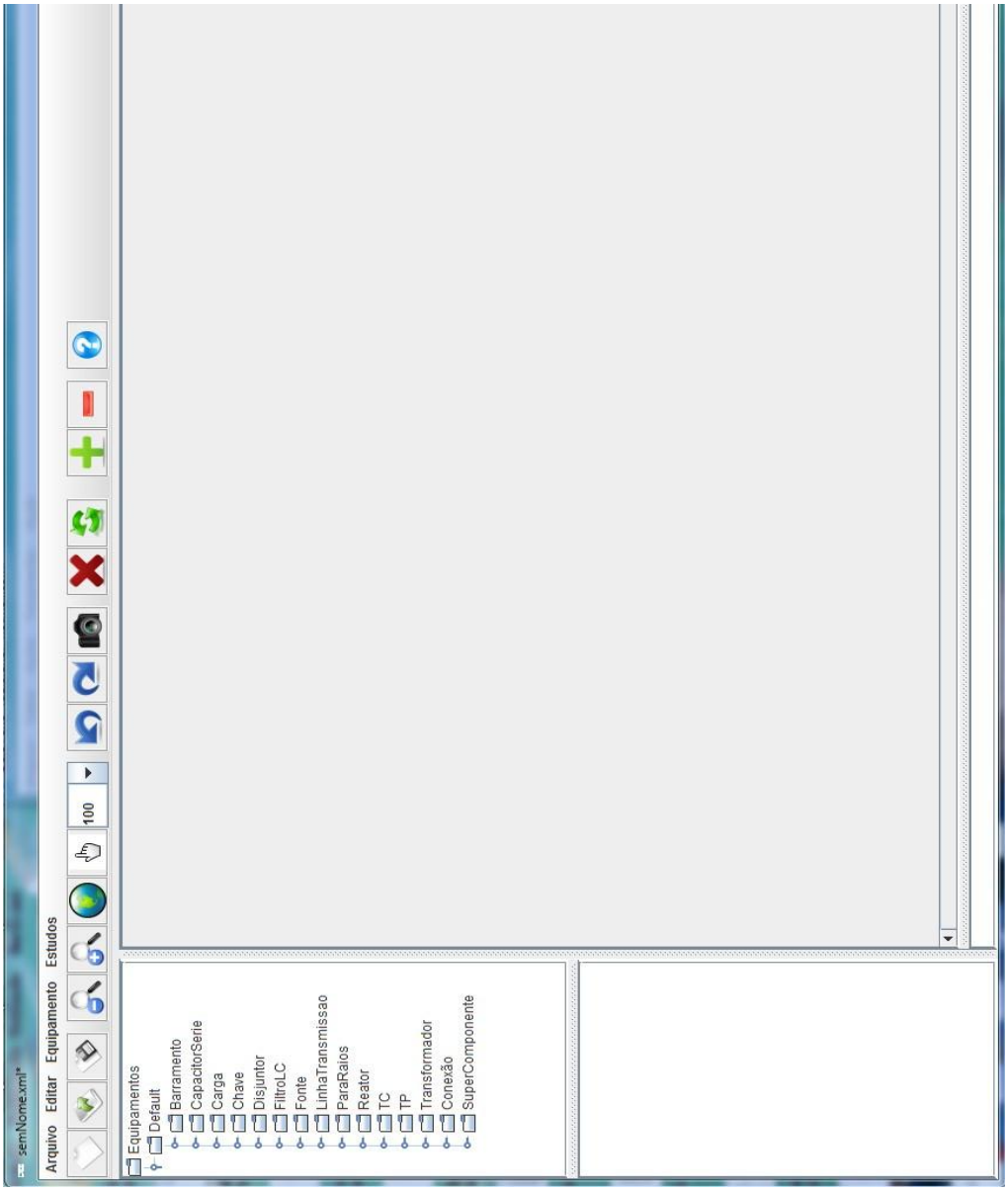


Figura 24: Lista de Equipamentos na árvore a esquerda/cima. Os equipamentos são carregados de um XML.

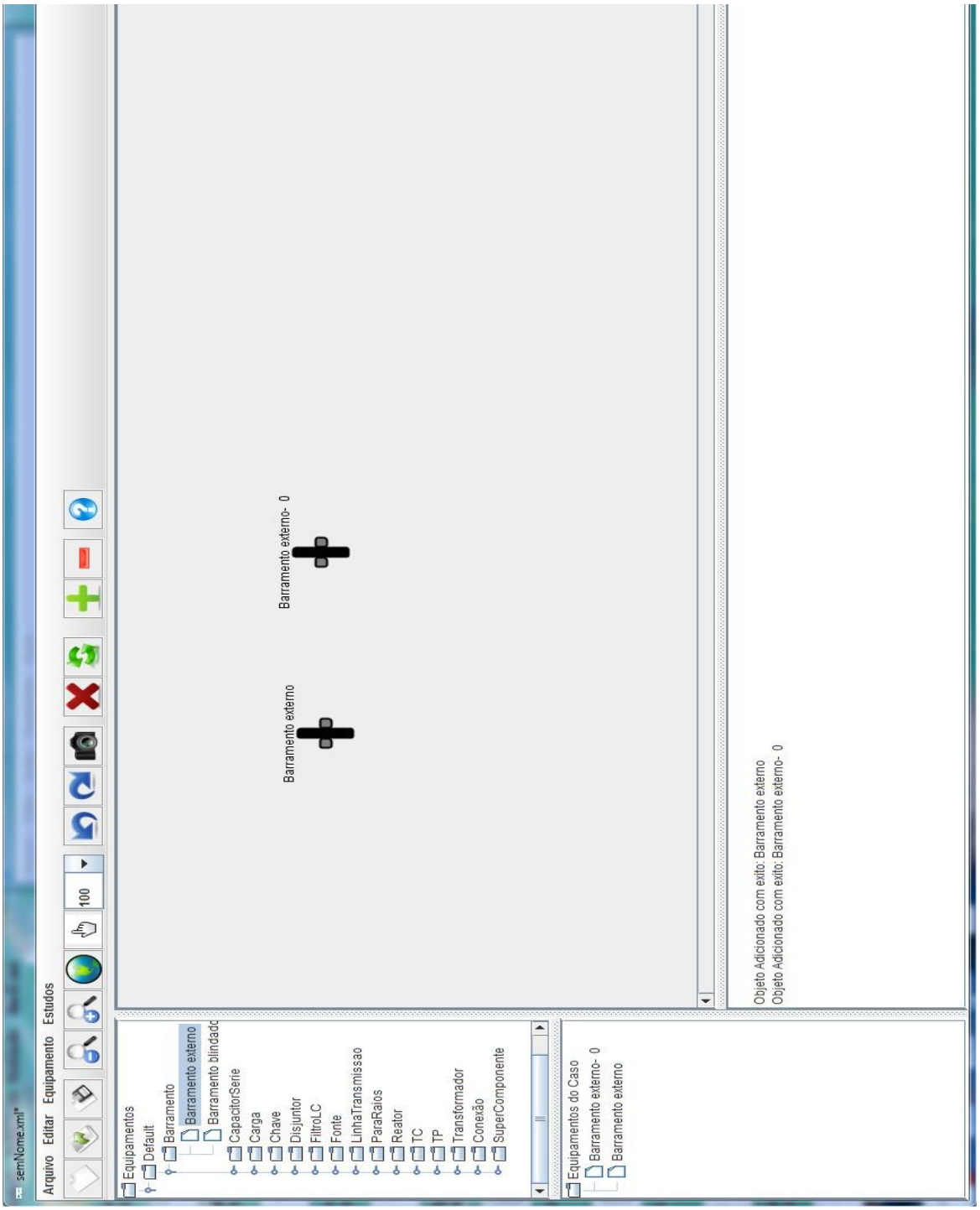


Figura 25: Dois elementos inseridos no Painel central. A inserção ocorre por DND; a árvore esquerda/abaixo contém os elementos inseridos.

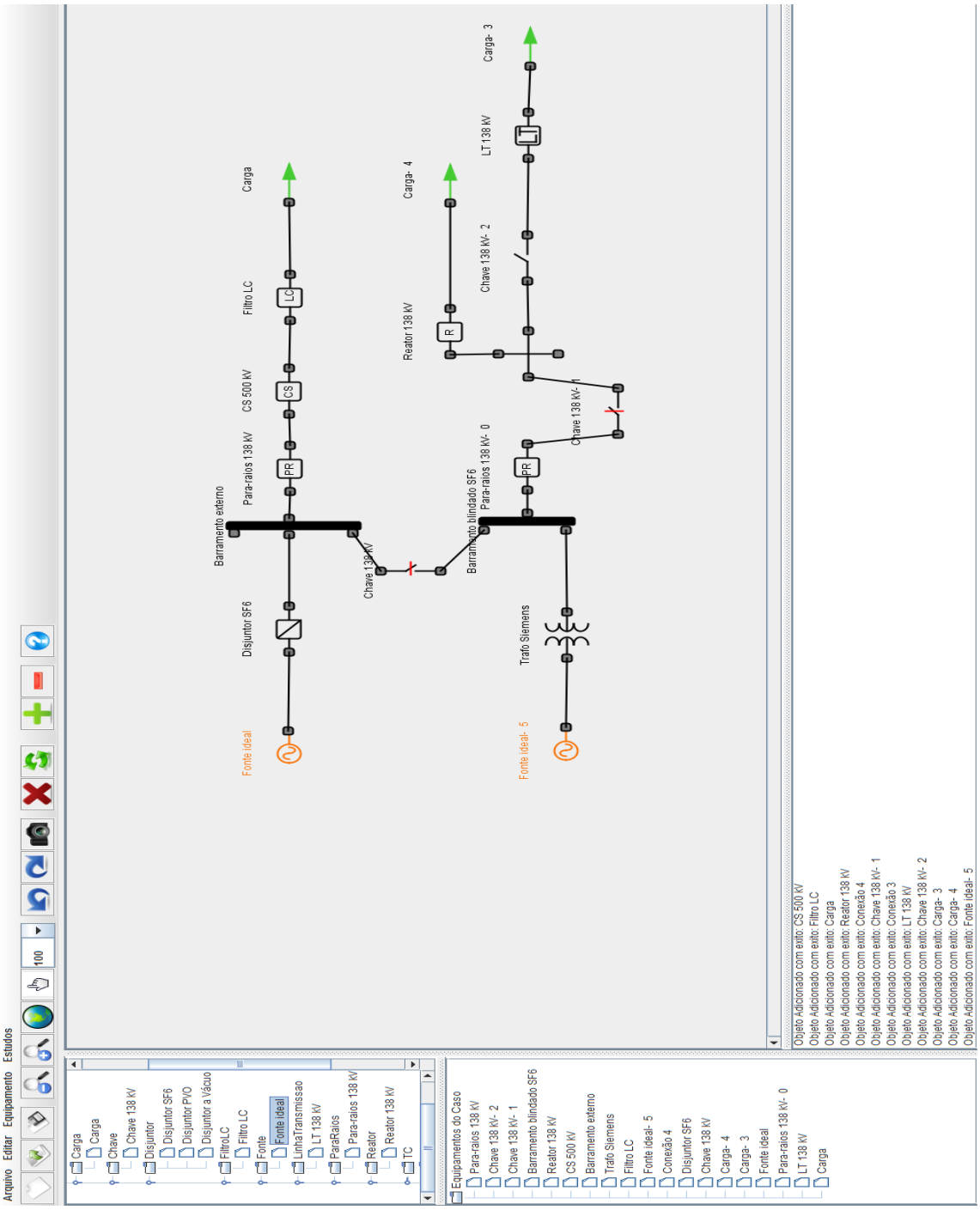


Figura 26: Sistema Hydra em ação. Vários equipamentos instanciados. A área de Log, abaixo/centro contém uma lista de ações realizadas pelo usuário.

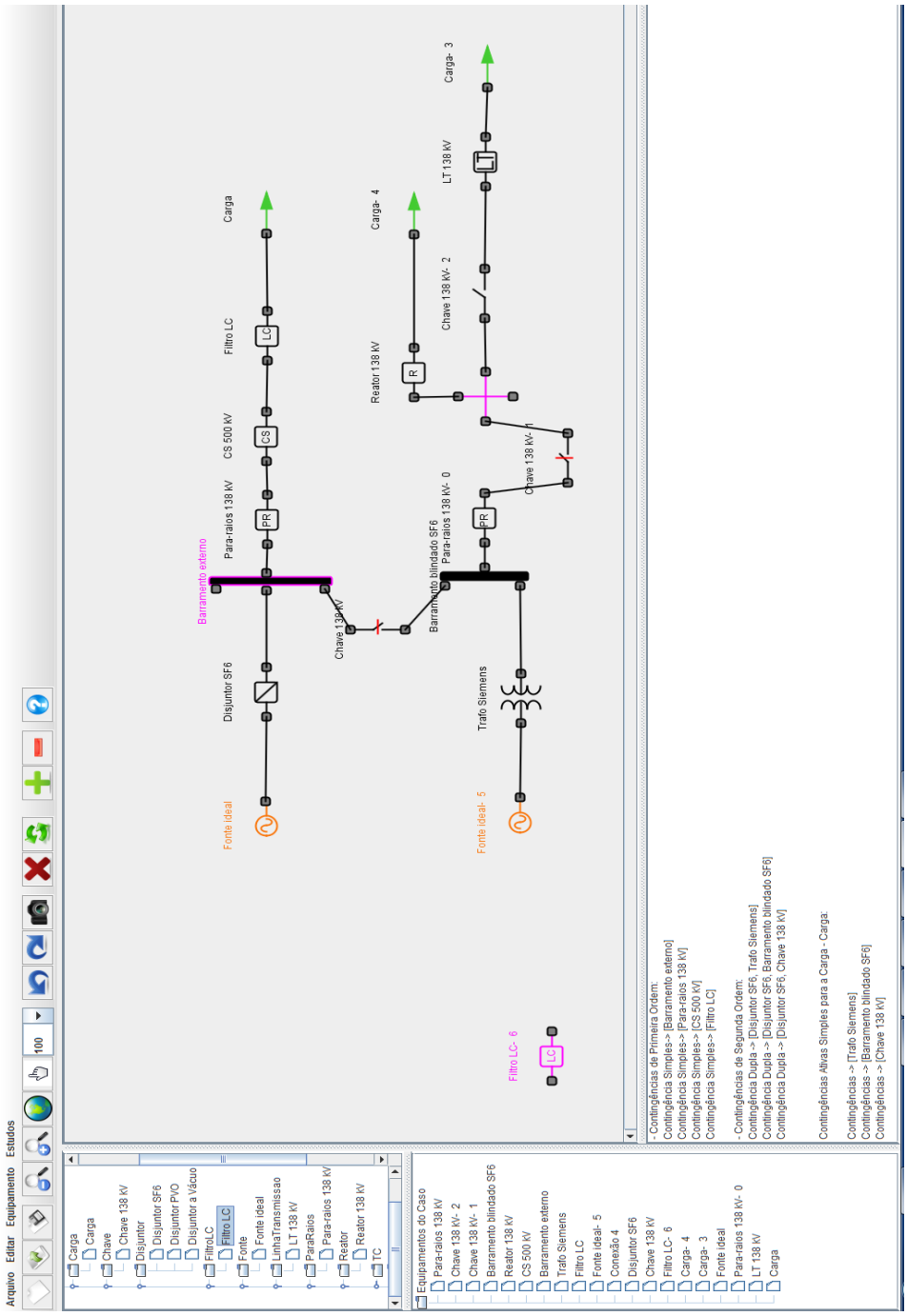
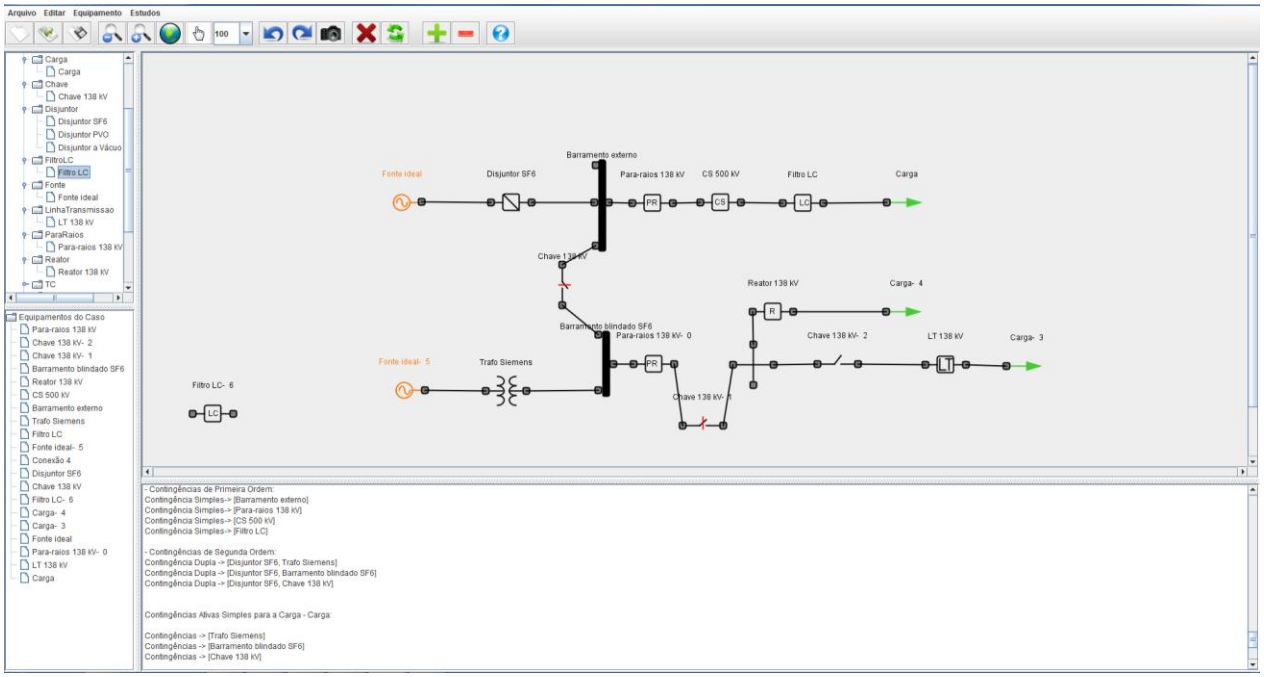


Figura 27: Uma das ferramentas inseridas no sistema: detecção de equipamentos desconectados (em roxo). Um Barramento e uma conexão foram deixadas sem uma conexão de propósito.



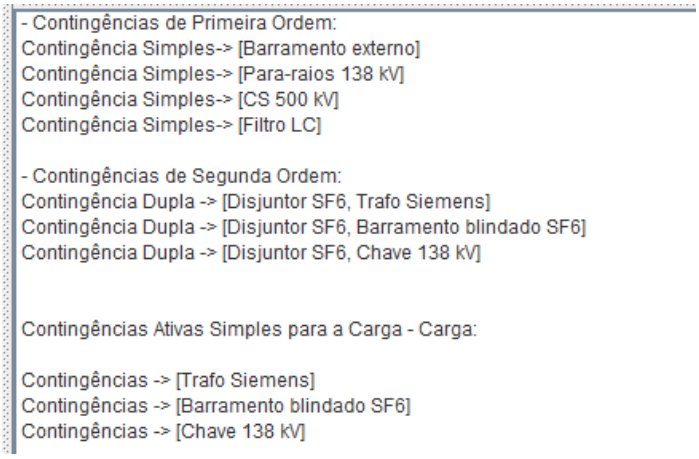


Figura 28: Outra ferramenta inserida no sistema: exibição dos caminhos onde podem ocorrer contingências; por padrão, apenas contingências simples e duplas são calculadas. Abaixo um zoom da área de Log.

Conclusão

Inicialmente o objetivo era construir uma interface genérica para o trabalho com grafos, permitindo que ambas as estruturas e visualização fossem contempladas no trabalho. Porém, com o trabalho desenvolvido, um passo além do objetivo inicial foi alcançado.

Por ser uma plataforma genérica, acreditava-se que poderia perder desempenho em algum momento, dado a complexidade de muitos algoritmos envolvidos e o tipo de estrutura utilizada para representar os grafos; porém, alcançou-se um estágio de desenvolvimento, onde separa-se por completo todas as estruturas que representam o problema, tanto matemáticas quanto gráficas, permitindo que ela seja genérica ao ponto de representar todo problema que possam envolver grafos e obter o melhor desempenho possível, dado a flexibilidade de extensão do código.

O propósito do trabalho foi atingido com êxito e ainda podemos aplicar nossa plataforma a um problema real: o Hydra. Permitir a idealização de um projeto complexo, tanto no sentido do problema, quanto no gerenciamento do desenvolvimento, dado que as equipes não estavam fisicamente num mesmo lugar, mostrou que o projeto teve êxito além do esperado.

9. Referências

[1] *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[2] *Data Structures and Algorithms*. Addison-Wesley, 1983.

[3] *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[4] *Projeto de Algoritmos*. Thomson, 2002.

[5] *Human-Computer Interaction (3rd Edition)*. Prentice Hall, 2003.

[6] *Sistema de Banco de Dados - Fundamentos e Aplicações - 4 Edição*. Pearson Education, 2005.

[7] *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, 2006.

[8] *Hardware, o Guia Definitivo:*. GDH Press e Sul Editores, 2007.

[9] *The Boost C++ Libraries: On-Line Book*. HighScore, 2008.

[10] *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[11] Kenneth Apple and Wolfgang Haken. Every planar map is four colorable. *Illinois Journal of Mathematics 21: 439–567*, 1977.

[12] Melissa DeLeon. A study of sufficient conditions for hamiltonian cycles. D*epartment of Mathematics and Computer Science, Seton Hall University*.

[13] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11), 1991.

[14] William Rowan Hamilton. Account of the icosian calculus. *Proceedings of the Royal Irish Academy, 6*, 1858.

[15] Ivan Herman, Ieee Cs Society, Guy Melanc¸on, and M. Scott Marshall. Graph visualization and navigation in information visualization: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 2000.

[16] L. Linsen, T. V. Long, P. Rosenthal, and S. Rosswog. Surface extraction from multi-field particle volume data using multi-dimensional cluster visualization. *IEEE Transactions on Visualization e Computer Graphics*, 14(6), 2008.

[17] Fernando Paulovich. *Mapeamento de dados multi-dimensionais - integrando mineração e visualização*. PhD thesis, Universidade de São Paulo, 2009.

[18] Fernando Vieira Paulovich, Maria Cristina F. Oliveira, and Rosane Minghim. The projection explorer: a flexible tool for projection-based multidimensional visualization. *In: Proceedings of XX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2007),Belo Horizonte. IEEE Computer Society Press*, 2007.

[19] M. Steyvers. Multidimensional scaling. *In. Encyclopedia of Cognitive Science*, 2002.

[20] Andre Kazuo Takahata. Heurísticas para programação inteira com trajetórias de busca factíveis e infactíveis. Master's thesis, Unicamp, 2009.

[21] Leff, Avraham; James T. Rayfield . Web-Application Development Using the Model/View/Controller Design Pattern. IEEE Enterprise Distributed Object Computing Conference. pp. 118-127.

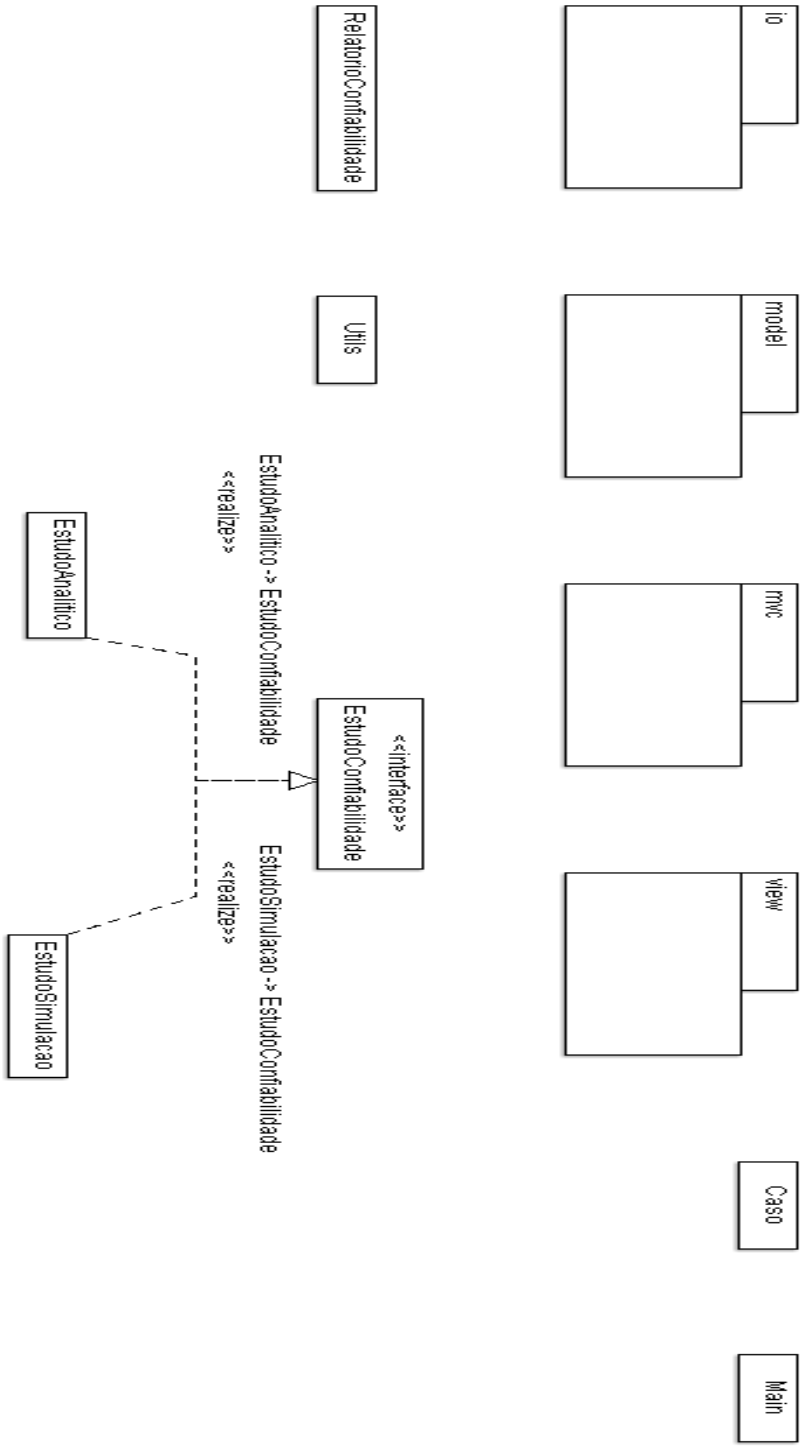
[22] Carlos K. C. Arruda, Helio P. Amorim Junior, Luis A. M. C. Domingues, Ricardo C. Fonte, Pablo A. Lisboa, Sergio L. Zaghetto, Lidio F. A. Nascimento, Lilian F. Queiroz. Analysis of Substation Availability. 2010 IEEE/PES Transmission and Distribution Conference and Exposition: Latin America

Anexos

Os anexos fazem referência ao pacote de classes do sistema Hydra.

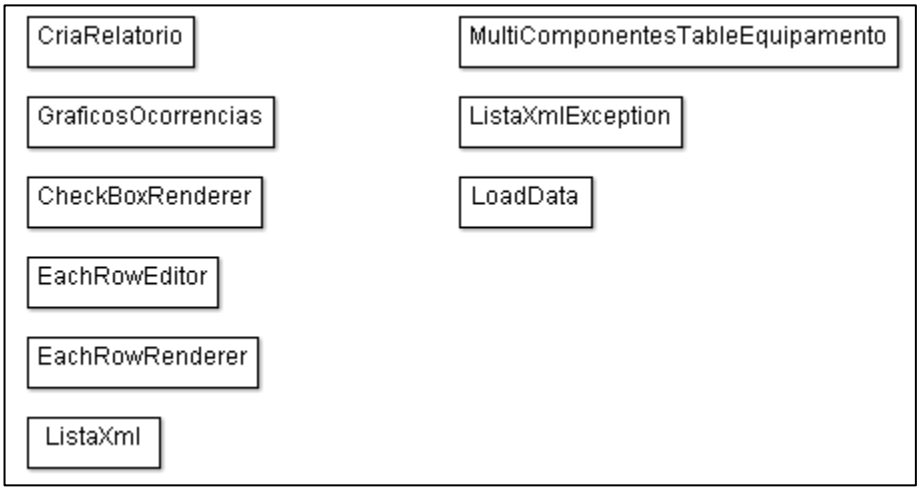
Anexo A

Pacotes e estrutura do sistema Hydra.



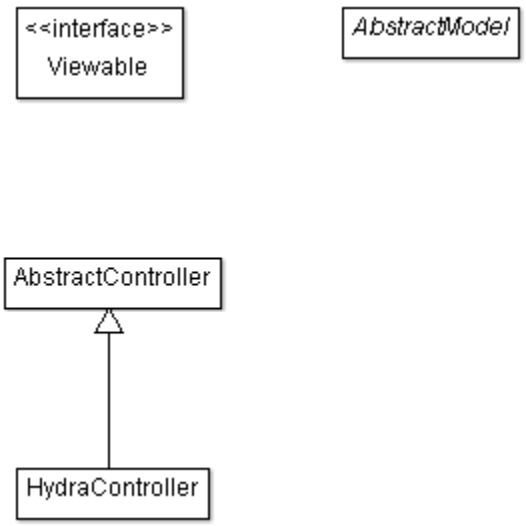
Anexo B

Pacote IO do sistema Hydra.



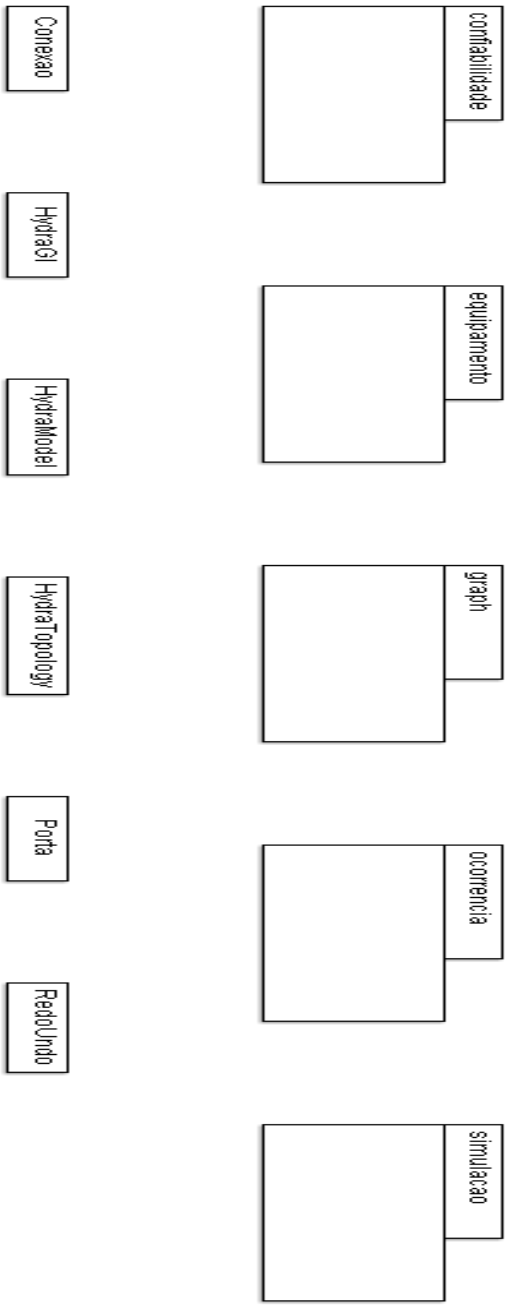
Anexo C

Pacote MVC do sistema Hydra.



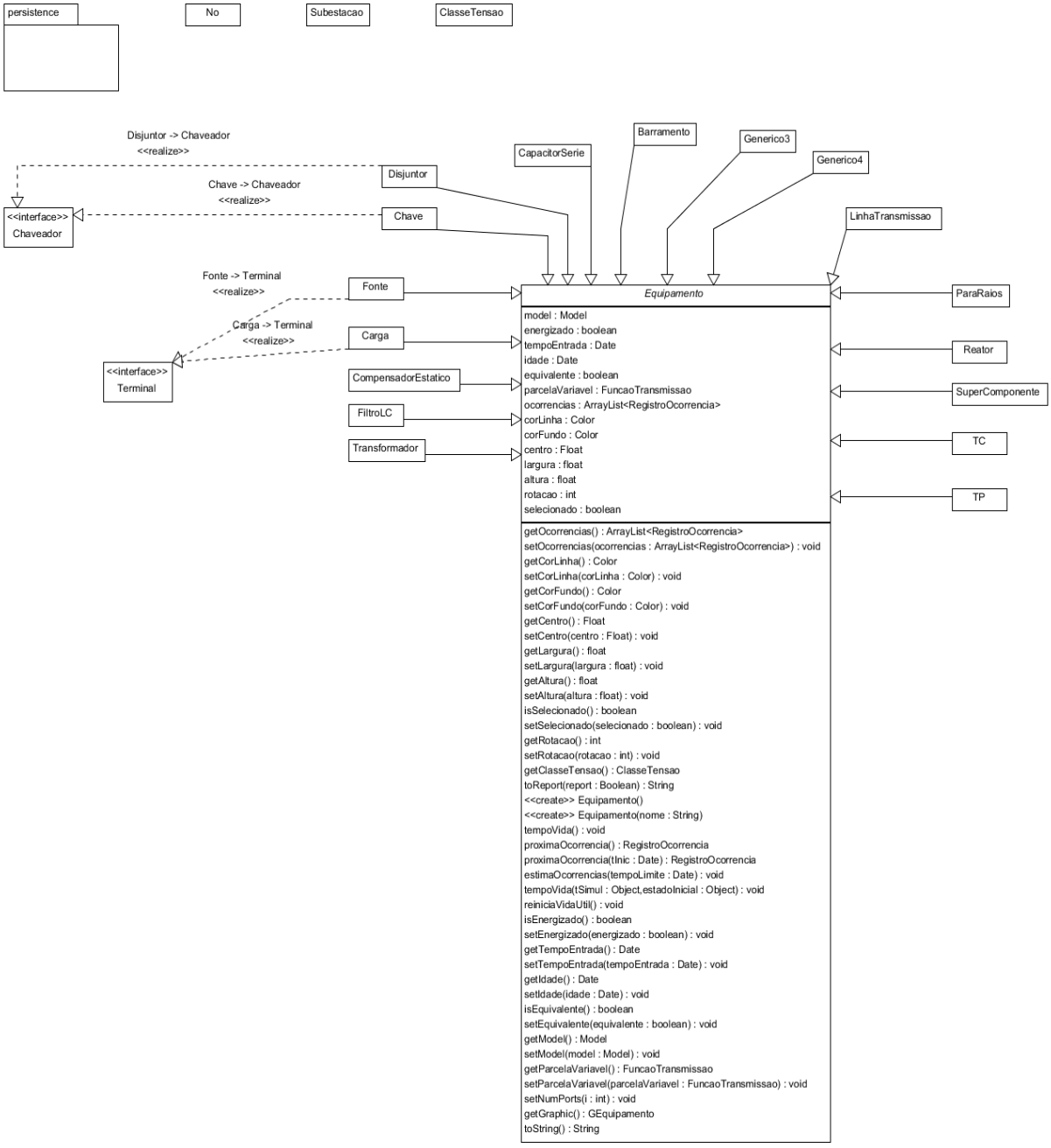
Anexo D

Lista de todos os pacotes em Model do sistema Hydra



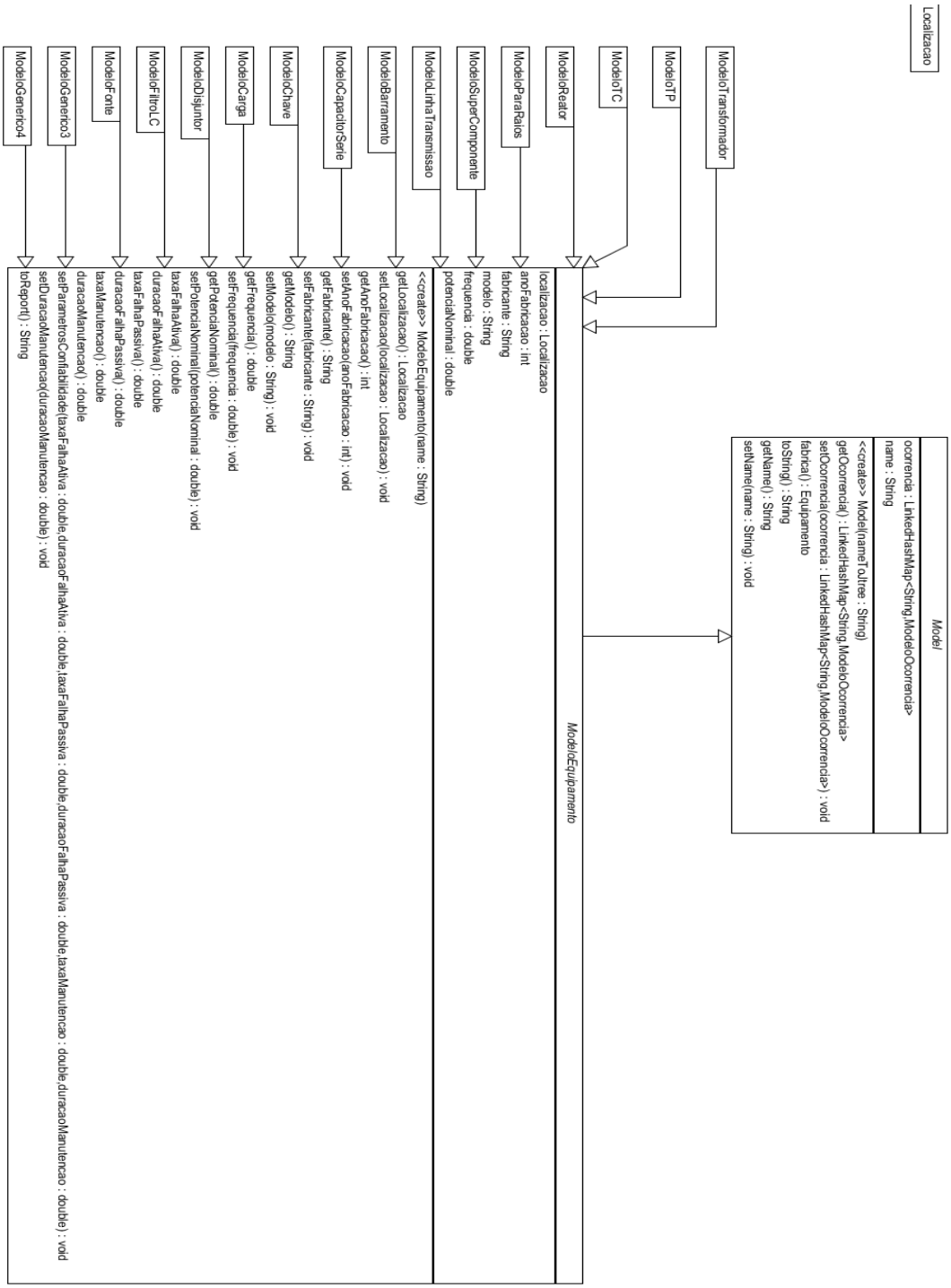
Anexo D1

Pacote Hydra Model Equipamento



Anexo D2

Pacote Hydra Model Equipamento Persistence



Anexo D3

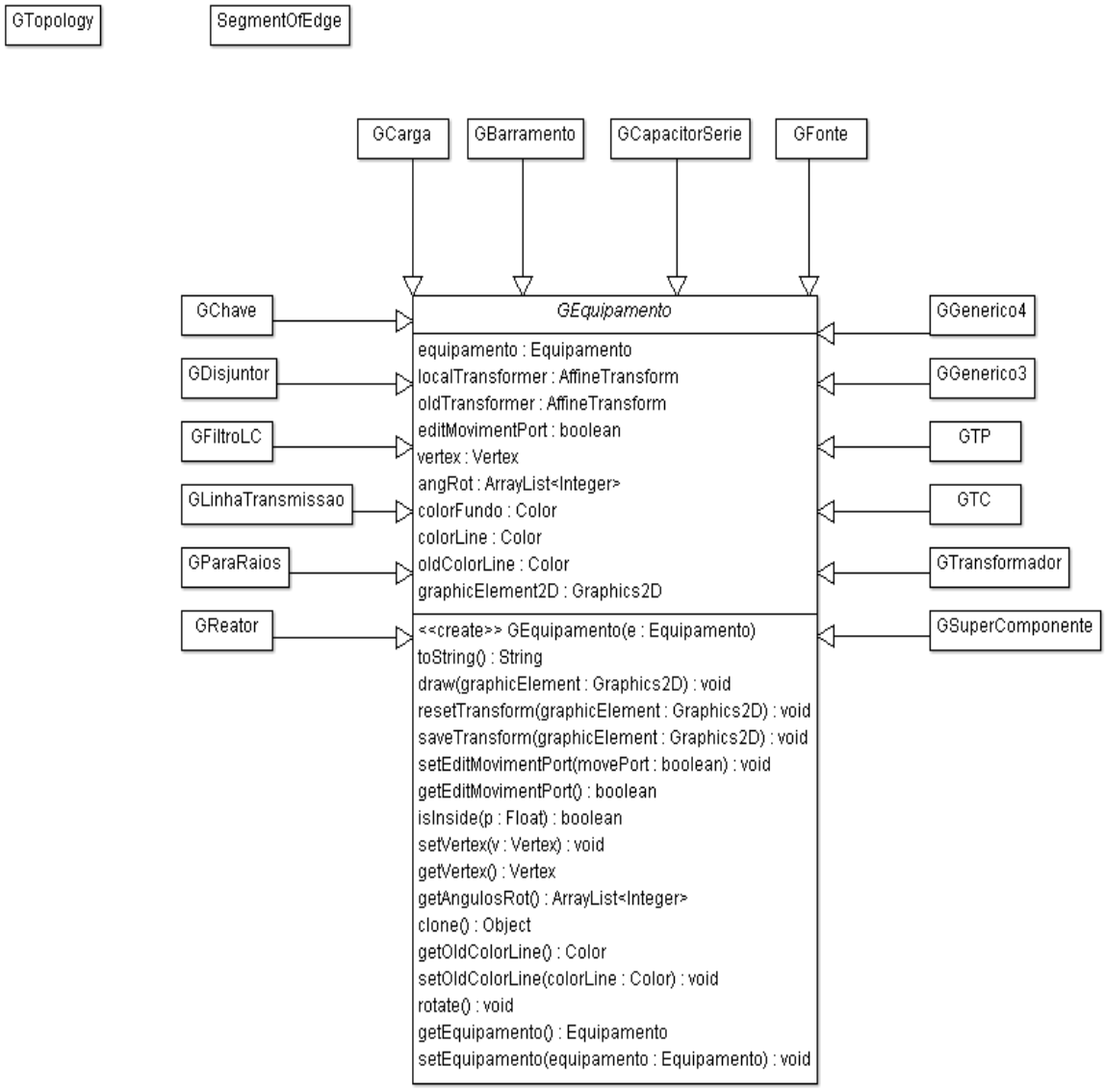
Pacote Hydra Model Graph

Anexo E

Lista de pacotes em Hydra View.

Anexo E1

Pacote Hydra View Equipamento



Anexo E2

Pacote Hydra View Graph