4
O Framework Gráfico

A ferramenta *Dependencies Viewer* precisava de uma *API* de desenho gráfico para poder gerar os seus diagramas. Esta API tem que ser obrigatoriamente orientada a metamodelos para conseguir trabalhar junto com a ferramenta e fornecer a flexibilidade buscada. Outra característica essencial é a predisposição para atender as demandas necessárias de um produto voltado para engenharia reversa, como a capacidade de interação humana no gráfico gerado. Isto eliminou a possibilidade de utilização de diversos *frameworks* gráficos, como o Graphviz.

Como visto na seção 2.3, foi feita uma análise no mercado para descobrir quais os produtos que poderiam atender a esta demanda. São diversos os *frameworks* para criação e exibição de diagramas, porém nenhum atendeu plenamente aos requisitos desejados.

Para criar um visualizador de diagramas, são várias as possibilidades, entre elas *Grappa*, *JHotDraw* e *JGraph*. Todos esses produtos são *APIs* que facilitam o desenvolvimento de editores. Na prática eles são uma evolução a camada gráfica de desenho da linguagem Java. O custo de aprendizado é alto.

O EMF (Eclipse Foundation, 2008) é o *framework* que chega mais perto do que se espera obter. Ele também é orientado a metadados, e a partir disso, consegue-se geração de editores e código. São dois, porém, os seus problemas: a curva de aprendizado e as limitações de uso. Para utilizá-lo é obrigatório o uso dentro do Eclipse, seja ele como ambiente de desenvolvimento, ou *standalone*. Isto é uma barreira a se considerar, dado que em Java, a grande maioria dos projetos usam a interface gráfica Swing.

Devido a todas essas limitações, foi decidido pelo desenvolvimento de um *framework* gráfico próprio, que funciona pelo conceito de metamodelo, permitindo ao usuário programar em um nível mais alto de abstração. Esse nível elevado de abstração é necessário para capacitar o usuário a pensar no seu domínio, e não em detalhes de implementação, como desenhar um círculo com um texto no meio.

O produto gráfico desenvolvido nesta dissertação pode ser considerado uma API, e não um aplicativo completo, pois não possui uma interface principal, precisa ser inserido dentro de outro programa.

4.1. O metamodelo

Surgiu então, inspirado no Talisman [Staa, 2002], a idéia de desenvolver um *framework* próprio.

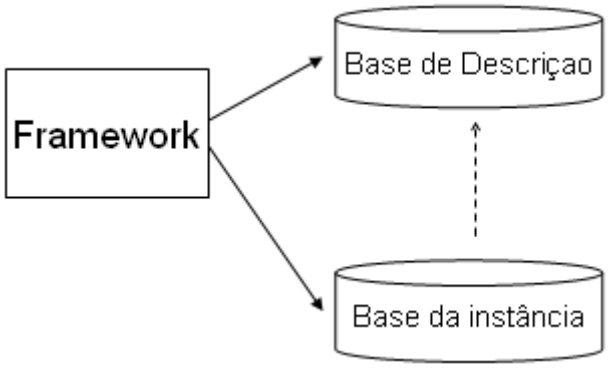


Figura 24: Esquema básico de funcionamento

A base de instância contém todos os objetos que constituem as informações que devem ser mostradas na forma de diagrama. São dados dos objetos ligados ao modelo de negócio do software que utiliza o framework. Os dados aqui estão relacionados com a base de descrição.

Na base de descrição são guardados os objetos relacionados com a apresentação e as regras de negócio que tratam disso. São armazenadas as definições que restringem e especializam a base de instância.

Num diagrama de fluxo de dados (DFD), por exemplo, um processo pode se ligar a outro processo, ou a uma entidade externa. Essa, porém, não pode se ligar a outra entidade externa. Na base de descrição ficam armazenadas estas regras e restrições. Exemplificando com um aplicativo de *e-commerce*. O DFD poderia conter um processo “Comprar” que se relaciona com outro processo “Entregar”. Na base de instância se armazenam as informações desses dois processos, como seus nomes e identificadores, além de suas relações e a definição dele como sendo do tipo “Processo. Na base de descrição, está configurado como será apresentado visualmente os dados dos objetos que são do tipo “Processo”. Caso o usuário tentasse criar um relacionamento entre duas entidades externas neste diagrama, o *framework* iria criticar e não deixaria. Isso

acontece porque as informações da base de instância estão ligadas e restringidas pela regras de definições da base de descrição.

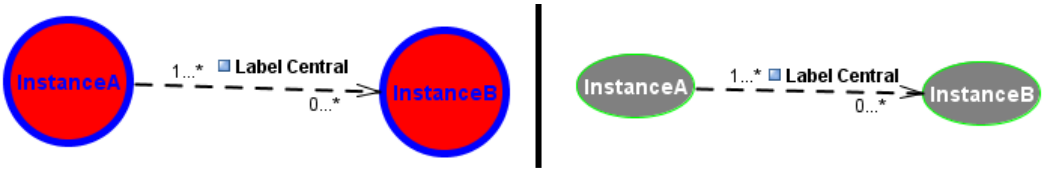


Figura 25: Mesma base de instância e diferentes bases de descrição

Na figura acima, é apresentado um exemplo visual de como a base de descrição interfere na representação do diagrama. O exemplo serve também para acentuar as diferenças das bases. O lado esquerdo e o lado direito na imagem apresentam diferentes diagramas que compartilham a mesma base de instância. Isto significa que possuem as mesmas informações de modelos. Isto pode ser observado pelos dados lidos nos diagramas. Apesar disso, apresentam diferentes representações. O da esquerda é circular com borda grossa em azul e fundo vermelho. O da direita se mostra oval, com borda fina em vermelho e fundo cinza. Estas diferentes representações são causadas pelo fato de possuírem uma base de descrição diferente.

4.1.1.
Modelagem da base de descrição

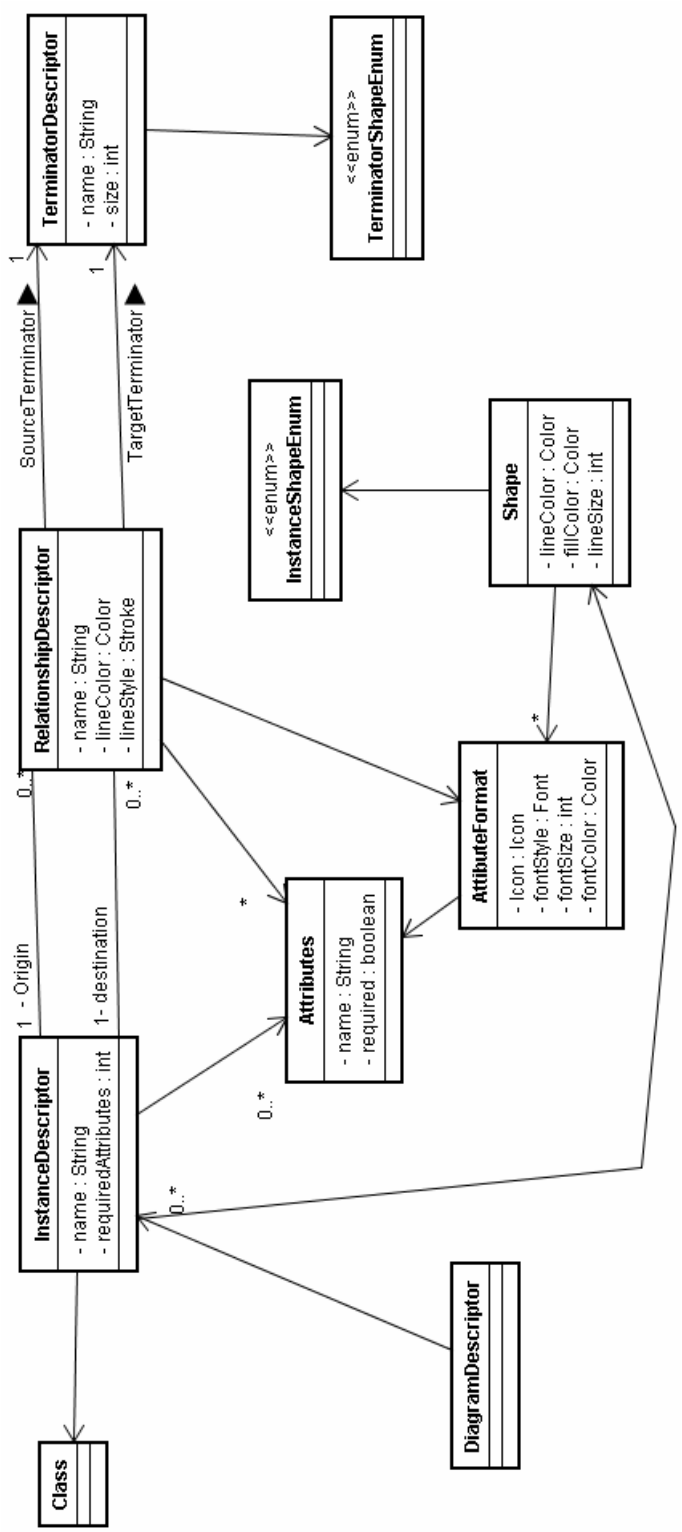


Figura 26: Diagrama de classes da base de descrição[13]

Na figura acima, podemos observar um diagrama de classes do meta-modelo utilizado. É ele que orienta a apresentação do diagrama na tela.

O *InstanceDecriptor* é responsável por descrever uma classe de instâncias. Imaginando um diagrama como um grafo, uma instância analogamente poderia ser considerada um vértice. Num diagrama de classes, por exemplo, um *InstanceDescriptor* descreveria como seria a apresentação visual da classe, quais ícones seriam exibidos ao lado do nome do atributo, entre outros. Ao existirem duas classes distintas num diagrama, na verdade são duas instâncias que referenciam um mesmo *InstanceDescriptor*.

Similarmente, o *RelationshipDescriptor* descreve uma classe de relacionamentos, tanto as restrições comportamentais como a apresentação visual. Estas duas características se encontram juntas para simplificar o uso da *API* e unificar em um único ponto toda a característica de um relacionamento. No exemplo do diagrama de classes, existiria, possivelmente, um descritor para os relacionamentos do tipo Associação. Uma particularidade deste objeto é que ele recebe sempre dois objetos do tipo *InstanceDescriptor*, representando a instância de origem e a instância de destino. Isso define o comportamento de quais instâncias ele pode se relacionar.

O *DiagramDescriptor* descreve as classes de instâncias que farão parte de um determinado tipo de diagrama. Isto serve para limitar e definir claramente o perfil de um diagrama. No exemplo do diagrama de classes, só existiria o descritor de instância “Classe” e os descritores de relacionamentos “Associação”, “Dependência” e “Herança”.

Tanto os descritores de instância, quanto os de relacionamentos possuem *Attributes*, que são atributos dos descritores, aos quais as instâncias ou relacionamentos concretos dão valor individualmente. No diagrama de classes, o título, nome dos métodos, entre outros, seriam *Attributes*.

Um *InstanceDescriptor* se relaciona com a classe *Shape* para definir o formato da apresentação de uma instância. Ele deve relacionar-se com exatamente um *Shape*. Um *shape* é a classe responsável por agrupar as informações necessárias para renderizar uma instância. Entre os seus atributos estão a cor da borda, cor de fundo, tamanho da borda e um

[13] Adaptado de: Staa, A.v.; Software and knowledge base specification for Talisman; preliminary report.

InstanceShapeEnum. Esta última é uma classe que encapsula, pelo padrão de delegação, o renderizador concreto, ou seja, a classe que na prática é responsável por desenhar a instância na tela.

Um *Attribute* pertencente a um *InstanceDescriptor*, para um determinado *Shape*, possui informações especificas para a sua exibição. Isto permite que um *Attribute* possa ser renderizado de modo diferente de outro, caso seja esta a vontade. Além disto, permite também que um mesmo *Attribute* tenha exibições diferentes em Shapes diferentes.

Entendendo que um relacionamento é exibido como uma linha entre duas instâncias, um *TerminatorDescriptor* descreve a forma de exibição do início e do final desta linha. Por isso que um *RelationshipDescriptor* possui dois relacionamentos com o *TerminatorDescriptor* no diagrama de classe apresentado. E análogo ao *Shape*, ele possui um relacionamento com a classe *TerminatorShapeEnum*, que pelo padrão de delegação, é o responsável pelo desenho na tela propriamente dita.

A classe *InstanceDescriptor* se relaciona com a classe *Class* de Java. Isto se dá para possibilitar a ligação com o modelo de negócios do usuário. É possível descrever qual objeto do modelo aquela instância representa, além de descrever o comportamento gráfico.

4.1.2. Modelagem da base de instância

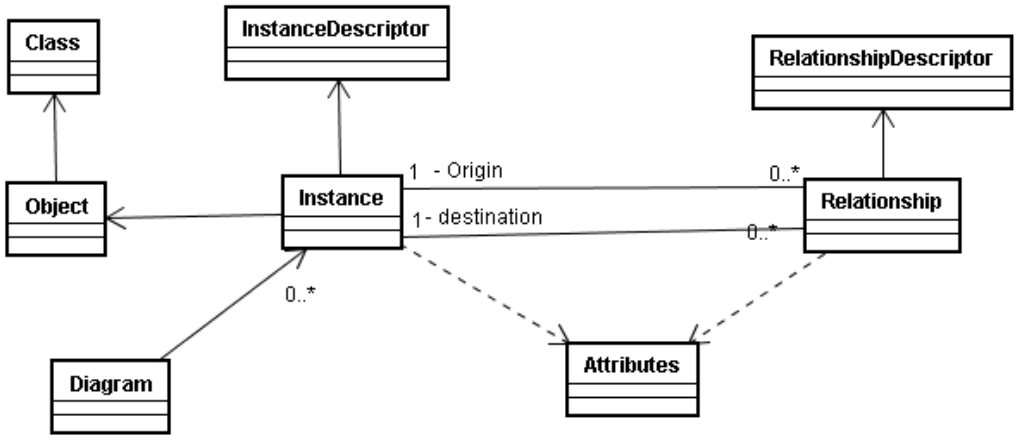


Figura 27: Diagrama de classe da base de instância

Na seção 4.1.1 foi apresentada a organização da base de descrição. Nesta seção será discutida a criação de uma base de instância.

Muito mais simples que a base de descrição, ele é composto apenas de três classes: *Diagram*, *Instance* e *Relationship*.

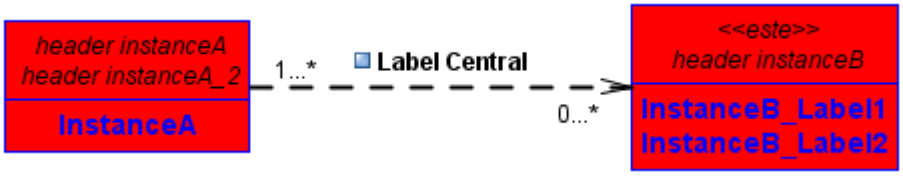
Um *Diagram* se relaciona com diversos *Instance* e é definido por um *DiagramDescriptor*. A conseqüência prática disso é que ocorrerá um erro ao tentar adicionar uma instância de um objeto de um tipo que não está no descritor do diagrama. Isso limita o diagrama a ter apenas o comportamento esperado.

Uma classe *Instance* possui dois relacionamentos com a classe *Relationship.* Um para definir o fato de ser a origem, e o outro para dizer que é o destino do relacionamento. Recebe no seu construtor uma classe *InstanceDescriptor*, que define, conforme dito anteriormente, o comportamento e informações para exibir uma instância. Com o mesmo objetivo, a classe *Relationship* recebe uma instância de *RelationshipDescriptor*. Caso a instância de origem ou de destino seja de um tipo diferente da configurada no descritor do relacionamento, o relacionamento não é criado e a visualização é abortada, com o lançamento de uma exceção.

Uma instância se relaciona com um *Object* Java. Isto define qual objeto concreto do modelo do usuário é representado por determinada instância do diagrama. Vale ressaltar que ela respeita a regra pela qual o objeto em questão deve ser obrigatoriamente do tipo configurado previamente no descritor da instância.

4.2.
API de Uso

Para usar o framework, a aplicação precisa definir primeiro a base de descrição, a partir deste ponto criar as instâncias concretas.



Na figura acima, pode-se observar duas instâncias com um relacionamento entre elas. Abaixo segue o código comentado de como foi gerado o diagrama acima.

```
public static void main(String[] args) {

    /*Cria-se o DiagramDescriptor*/
    DiagramDescriptor diagramDescriptor = new DiagramDescriptor("DiagramaTeste");

    /*Cria-se o Shape*/
    Shape shape = new Shape(new BoxShapeEnum());

    /*Cria-se o único InstanceDescriptor existente, que utiliza o Shape criado anteriormente*/
    InstanceDescriptor instanceDescriptor = new InstanceDescriptor("Label",shape);

    /*O atributo Label é criado. Este atributo corresponde a parte de baixo da instância*/
    Attribute label = new Attribute("label");

    /*É configurado a apresentação deste atributo na tela.*/
    AttributeFormat labelFormat = new AttributeFormat(label);
    labelFormat.setFont(new Font("Arial", Font.BOLD, 14));
    labelFormat.setFontColor(Color.BLUE);

    /*Adiciona-se ao shape a configuração do atributo*/
    shape.addAttributeFormat(labelFormat);

    /*Configura como o Shape deve ser apresentado*/
    shape.setLineColor(Color.BLUE);
    shape.setLineSize(1);
    shape.setFillColor(Color.RED);

    /*Cria-se o atributo Header, e logo depois configura a sua apresentaçao*/
    Attribute header = new Attribute("header");
    AttributeFormat headerFormat = new AttributeFormat(header);
    headerFormat.setFont(new Font("Arial", Font.ITALIC,  12));

    /*Adiciona ao shape a configuraçao visual do atributo header*/
    shape.addAttributeFormat(headerFormat);

    /*É adicionado na instância os atributos criados*/
    instanceDescriptor.addAttribute(label);
    instanceDescriptor.addAttribute(header);

    /*Adiciona-se no diagrama a sua única instância*/
    diagramDescriptor.addInstanceDescriptor(instanceDescriptor);

    /*Cria-se o relacionamento. Nao é necessário adicionar a instância o relacionamento criado, pois
     * ele mesmo na sua construção se encarrega disto*/
    RelationshipDescriptor relationshipDescriptor = new RelationshipDescriptor(
            "Association", instanceDescriptor, instanceDescriptor);

    /*Configura a apresentaçao visual do relacionamento */
    relationshipDescriptor.setLineStroke(LineStrokeFactory.createTracedStroke(2));

    /*Configura sua apresentaçao visual do seu atributo Label*/
    labelFormat = new AttributeFormat(relationshipDescriptor.findAttribute("label"));
            labelFormat.setFont(new Font("Arial", Font.BOLD, 12));
    labelFormat.setIcon(Utilities.loadImage("variablePublic.gif"));
    relationshipDescriptor.addAttributeFormat(labelFormat);

    /*Cria o terminador de seta para este relacionamento, e depois adiciona ele no descritor do relacionamento*/
    TerminatorDescriptor terminatorDescriptor = new TerminatorDescriptor(new ArrowAnchorShapeEnum());
    relationshipDescriptor.setTargetTerminator(terminatorDescriptor);
    terminatorDescriptor.setSize(15);

    /*Usando o facade, carrega o metamodelo no framework*/
    Facade.getInstance().loadDiagramDescriptor(diagramDescriptor);
```

Código 7: Criação da base de descrição

Um detalhe a ser mencionado é que não é possível adicionar atributos ao descritor de relacionamento, porque os únicos atributos possíveis são o texto da origem, o texto do destino e o texto central. Esta é uma restrição desta implementação, porém o modelo permite evoluções para que seja possível uma expansão mais adiante.

Abaixo segue código referente à criação das instâncias reais.

```
        /* Cria-se a instância A */
        Instance instanceA = new Instance(Facade.getInstance()
                .findInstanceDescriptor(diagramDescriptor, "Label"));

        /* Atribui a instância A o valor dos seus atributos */
        instanceA.addAttributeValue("label", "InstanceA");
        instanceA.addAttributeValue("header", " header instanceA");
        instanceA.addAttributeValue("header", " header instanceA_2");

        /*Cria-se a instancia B */
        Instance instanceB = new Instance(Facade.getInstance()
                .findInstanceDescriptor(diagramDescriptor, "Label"));

        /*Atribui a instância B o valor dos atributos */
        instanceB.addAttributeValue("label", "InstanceB");
        instanceB.addAttributeValue("label", "InstanceB_Label2");
        instanceB.addAttributeValue("header", " <<este>>");
        instanceB.addAttributeValue("header", " header instanceB");

        /*Cria o relacionamento*/
        Relationship relationshipA = new Relationship(relationshipDescriptor,instanceA, instanceB);

        /*Adiciona o valor dos atributos no relacionamento*/
        relationshipA.addAttributeValue("label", "Label Central");
        relationshipA.addAttributeValue("sourceLabel", "1...*");
        relationshipA.addAttributeValue("targetLabel", "0...*");

        /* Cria o diagrama, adicionando posteriormente as instâncias criadas e o relacionamento.*/
        Diagram diagram = new Diagram(diagramDescriptor);
        diagram.addInstance(instanceA);
        diagram.addInstance(instanceB);
        diagram.addRelationship(relationshipA);


        /* Cria-se o frame para exibir o diagrama */
        int width=800,height=600;
        JFrame frame = new JFrame();

        /*A partir do Facade, pega-se o componente com o diagrama já desenhado*/
        JComponent component = Facade.getInstance().getDiagramView(diagram);

        /*Cria o frame principal da janela*/
        frame.add(component, BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        java.awt.Dimension screenSize = java.awt.Toolkit.getDefaultToolkit()
                .getScreenSize();
        frame.pack();
        frame.setBounds((screenSize.width-width)/2, (screenSize.height-height)/2, width, height);
        frame.setVisible(true);

        Facade.getInstance().invokeLayout(diagram);
    }
```

Código 8: Criação da base de Instância

Outros tipos de *Shape* disponíveis no momento são: *CircleShapeEnum*, para desenhar um círculo, *LabelShapeEnum*, para exibir apenas um texto sem nenhuma borda e *OvalShapeEnum*¸ para exibir um formato oval.

O *BoxShapeEnum*, utilizado no exemplo, pode ser parametrizado para ser renderizado na horizontal, ao invés do *default* exibido na imagem, que é na vertical. Além disso, pode-se configurar para exibir ou não a divisão entre os atributos. Isto serve para conseguir montar um quadrado normal ou uma borda com o estilo de um diagrama de classe com o mesmo desenhador. Ele também recebe um parâmetro para poder ter ou não o canto arredondado.

O usuário do *framework* pode desenvolver outro renderizador de formatos livremente, bastando, para isso, criar uma classe em Java que implementa a

interface *InstanceShapeEnum* e passá-la como parâmetro no construtor do objeto *Shape*.

4.3. Renderização

A seguir, são apresentados alguns diagramas de seqüência que mostram como é implementada a renderização.

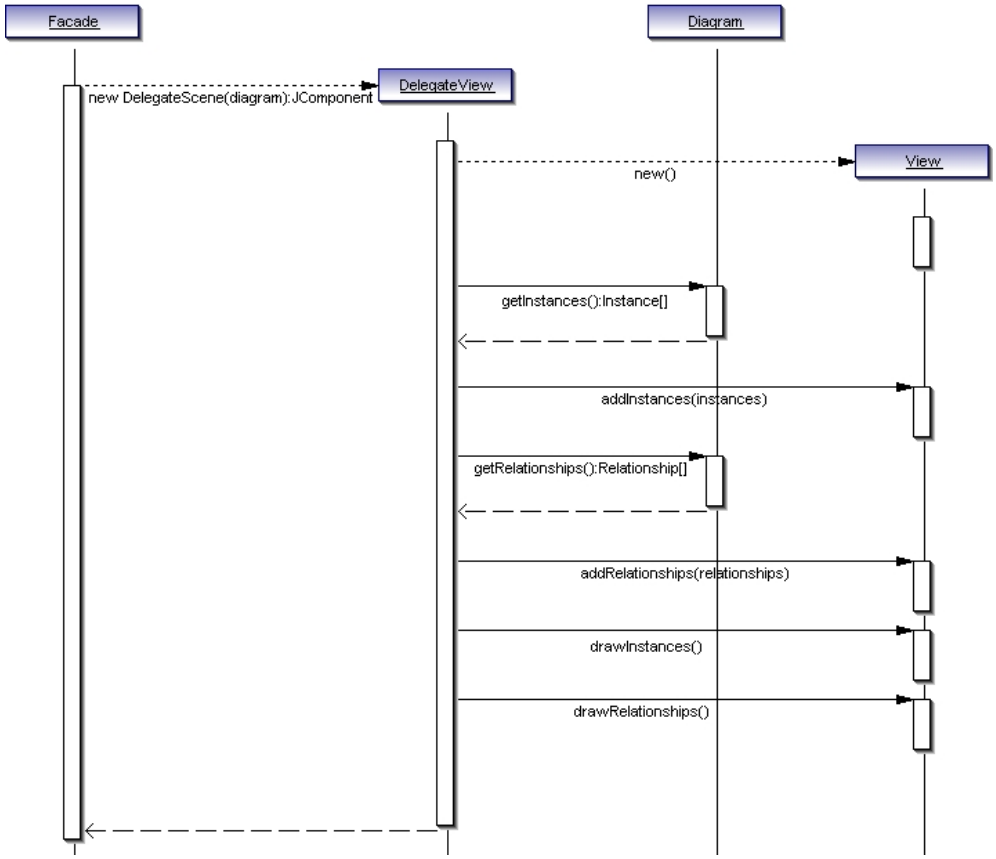


Figura 28: Método *getView(Diagram diagram),* responsável por criar o diagrama

Na figura acima é detalhado o mecanismo de criação do diagrama. Tudo tem início pela classe *Facade*. Esta classe é, como o próprio nome sugere, uma interface de comunicação com o mundo externo implementado através do padrão de projeto *Facade*. Além disso, ela também implementa o padrão *Singleton*.

Logo no início do método, um objeto da classe *DelegateView* é criado. Esse, através da delegação, encapsula a verdadeira *View,* que é simplesmente um JPanel do Java, com a inteligência específica para a renderização desejada. Ele é retornado com a chamada do método *getView()* da classe *Facade* e pode ser usado normalmente pela aplicação como um *JComponent* qualquer. A

DelegateView foi criada para isolar a aplicação da API gráfica utilizada, pois a *View* está completamente dependente dela sendo, inclusive, uma classe que herda de outra da *VisualLibrary*. Entre as vantagens de utilizar esta técnica, está o fato de facilitar a troca da API gráfica, caso tenha necessidade disso em algum momento.

Após ser criado e receber um diagrama, o *DelegateView* gera uma *View* e nela adiciona os relacionamentos e instâncias do diagrama recebido. Posteriormente, chama métodos responsáveis por desenhar as instâncias e os relacionamentos. Estes métodos são detalhados abaixo:

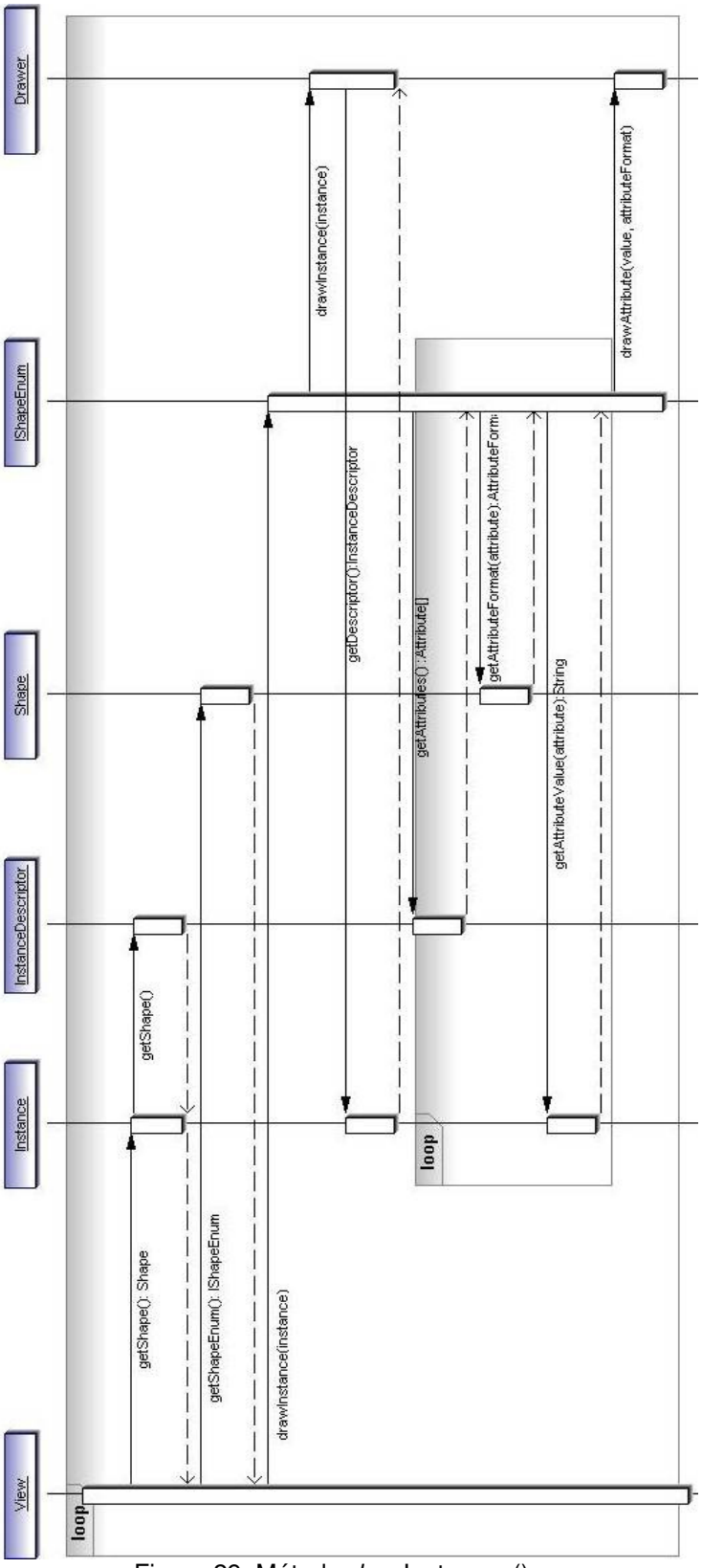


Figura 29: Método *draw*Instances()

Na Figura 29, é explicado o método responsável por desenhar as instâncias na tela. O loop maior sinaliza que o conjunto de métodos é chamado para cada Instância existente na classe *View*.

Para cada objeto *Instance* existente, busca-se o seu *Shape* associado, que está guardado no *InstanceDescriptor* da instância. A partir do *Shape*, adquire o *IShapeEnum* que, como explicado anteriormente, serve para encapsular a classe *Drawer,* responsável de fato pelo desenho. O método *drawInstance()*, recebendo a instância a ser desenhada, é chamado, e obtém, para desenhar, o descritor da instância em questão. Isso ocorre pelo fato deste descritor ser o responsável por guardar as informações de interface, como cor de fundo, cor da borda, entre outros. Diferentes objetos *IShapeEnum* desenham bordas e formatos distintos.

Depois, para renderizar os atributos, é necessária outra interação, na qual para cada atributo existente no descritor da instância, é obtido do *Shape* o seu formatador e posteriormente o seu valor concreto, guardado na instância propriamente dita. Com isso em mãos, basta chamar o método responsável por desenhar um determinado atributo. O desenho de um atributo engloba exibir ou não determinado ícone antes do texto, exibir a fonte correta, com tamanho correto, entre outros.

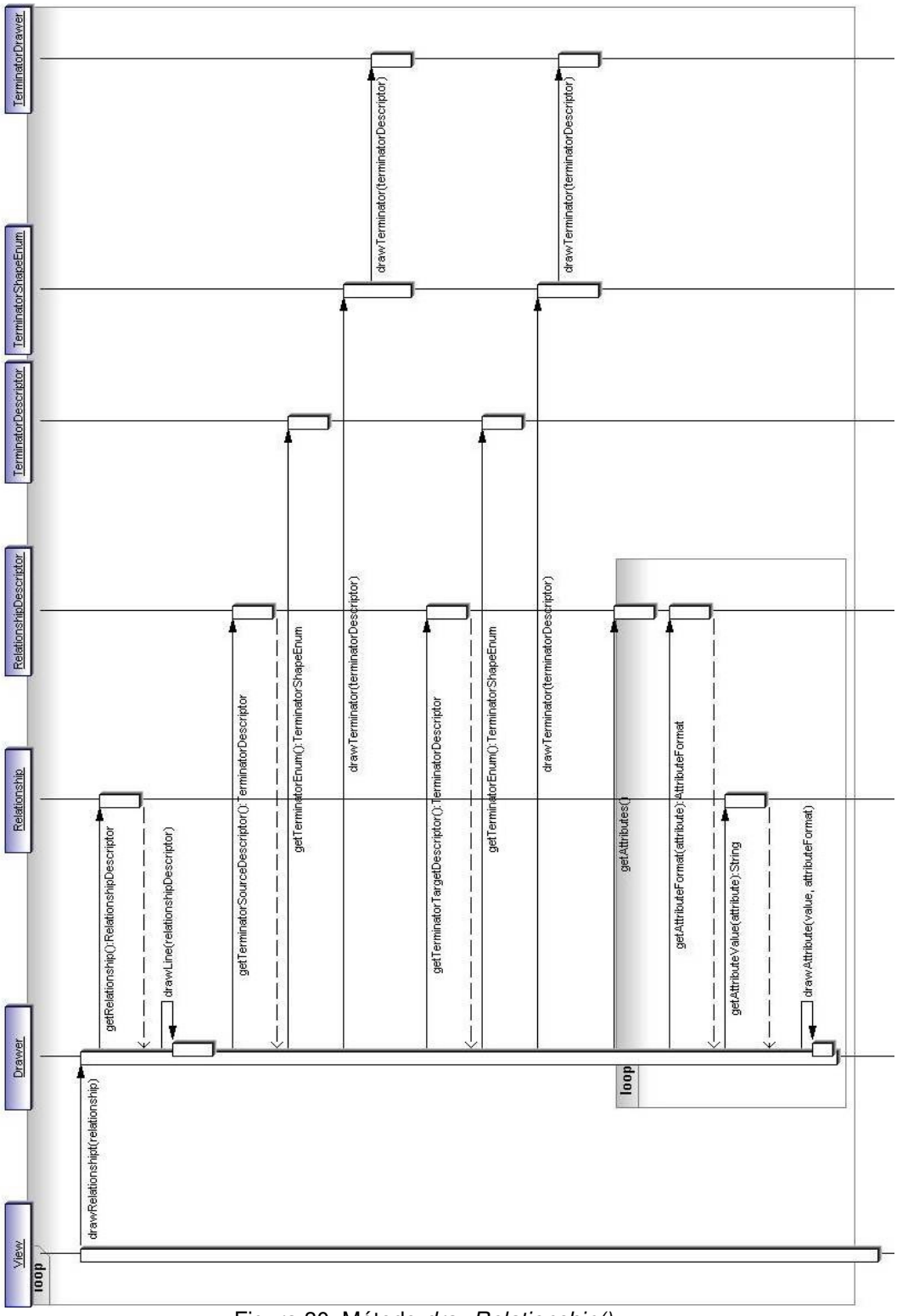


Figura 30: Método *drawRelationship()*

Similar ao método para desenhar instâncias, cada relacionamento é exibido separadamente a partir de uma iteração no início. Como desenhar um relacionamento é muito mais simples do que desenhar uma instância, não foi necessário criar um mecanismo de delegação para o *Drawer*, pois toda a informação obrigatória estava armazenada no descritor do relacionamento.

Então a linha do relacionamento, ligando uma instância à outra, é desenhada, respeitando, obviamente, a cor configurada no descritor, assim como espessura, hachura, entre outros.

Após desenhar a linha principal, os terminadores são desenhados. Ele possui uma arquitetura parecida com o *IShapeEnum*. Existe um *ITerminatorEnum* que possui um *TerminatorDrawer*. Por ser mais simples, não necessitava obrigatoriamente dessa estrutura. Porém, mostrou-se necessária, pois pode haver um mesmo desenhador de terminador para diferentes tipos, como o triângulo cheio e o vazio. São dois tipos distintos com um mesmo desenhador.

Os atributos são exibidos obtendo, para cada atributo dentro de uma iteração, a partir do descritor, o seu formatador e, do relacionamento, o valor concreto. Só então o atributo é desenhado.

4.4. Modelagem de Interação

Como dito anteriormente, o framework tem que estar pronto para atender as necessidades de um software desenvolvido para engenharia reversa. Um requisito fundamental é permitir que haja interação do usuário com o diagrama exibido. Para isso, o *framework* precisa possuir algum meio de comportar esta funcionalidade.

Uma das premissas principais do framework é ser de simples utilização. Para isso, o modelo de interação deveria ser algo natural ao desenvolvedor. A *Visual Library* (*API* de apoio utilizada) possui uma interface para programação rica. Isso se dá pela quantidade de funções que ela predispõe. Para obter a coordenada de um simples clique do mouse, por exemplo, é necessário fazer uma conta que leva em conta a escala de visualização do diagrama.

Visando facilitar esse processo, foi criada a classe abstrata *ActionAdapter*. Esta classe provê métodos que visam aumentar a usabilidade da *API*. Além disso, os nomes dos métodos foram mudados para se equiparar com a API gráfica *swing, padrão* do Java, já que os originais eram diferentes.

Para interagir com o gráfico, então, basta o desenvolvedor especializar a classe *ActionAdapter*, estender o método de interação desejado, como *mousePressed()* ou *keyPressed()*, e adicionar a nova classe no objeto da classe *Instance* (seção 4.1.2) que deve reagir a ação do usuário.

As funcionalidades de visualização de detalhes (seção 3.5.3) e navegação (seção 3.5.2) fazem uso desse modelo.

4.5. O Processo de desenvolvimento e Instalação

A ferramenta foi desenvolvida nos mesmos moldes que o *Dependencies Viewer* (Seção 3.4.1). Foi usado *test driven development* em todos os pacotes em que fosse possível ser aplicado. Apenas as classes especificadamente desenvolvidas para desenhar não tiveram cobertura de testes, pois precisariam de outras técnicas de testes, não abordada nesse processo.

Para utilizar o framework dentro de uma aplicação desenvolvida em Java, é necessário inserir o Jar *meta-drawer-0.0.1.jar*[14] no *classpath* da aplicação.

[14] http://code.google.com/p/meta-designer/