3
A Ferramenta *Dependencies Viewer*

Como visto anteriormente, para programas escritos na linguagem Java, existe uma infinidade de ferramentas capazes de fazer a engenharia reversa. Os diagramas gerados, em geral, são baseados na notação UML. Para a exibição do comportamento e chamadas da aplicação, é normalmente utilizado o diagrama de seqüência.

Uma carência foi registrada ao procurar produtos similares para outras linguagens. Não existe software pago ou grátis que faça isso de forma adequada para um programa escrito em PL/SQL, por exemplo.

O objetivo da ferramenta *Dependencies Viewer* é fornecer informações similares a encontrada no diagrama de seqüência para qualquer tipo de linguagem, ou pelo menos que facilite este objetivo. Dentro os principais requisitos, estão:

- Ser orientada a metamodelos. É preciso que a ferramenta seja flexível para servir como visualizador de dependências de inúmeras linguagens. Isto inclui ter sua interface e seu modelo altamente configuráveis via arquivos de metadados.
- Possuir funções de geração de modelos parciais. Normalmente um diagrama de dependência é grande o suficiente para poluir demais a tela com informações desnecessárias ao usuário.
- Função de exportação do modelo gerado. Para servir como uma ferramenta de documentação é preciso exportar o diagrama gerado para algum formato de saída conhecido, como uma imagem PNG.
- Rapidez no desenvolvimento. Para ser uma ferramenta diferenciada, é importante ter como característica marcante a agilidade na criação de uma nova instância da ferramenta.

3.1.
O Metamodelo

Na prática, a tarefa de produzir um diagrama comportamental para softwares escritos em qualquer linguagem é muito difícil, senão impossível, visto

que é impraticável a implementação de um analisador sintático genérico o suficiente para conseguir interpretar todas as especificidades existentes num conjunto finito, porém grande, de linguagens.

Para solucionar esse problema a ferramenta *Depedencies Viewer* segue o modelo elaborado mostrado abaixo:

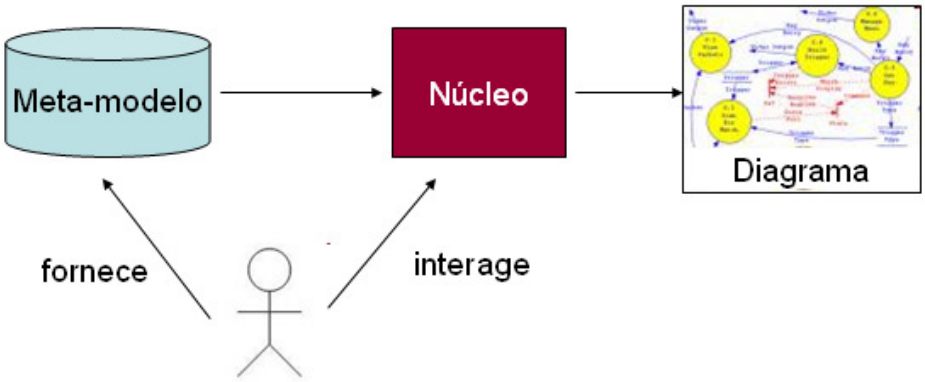


Figura 5: Esquema de funcionamento da ferramenta

Como pode ser visto, a ferramenta tem como ponto de extensão um meta-modelo que representa a árvore de chamadas e as informações necessárias para exibir o diagrama. Para flexibilizar esse ponto, o usuário da ferramenta deve complementá-la com a implementação de um analisador específico para a linguagem escolhida. Esta extensão não é algo complexo, visto não ser necessária a produção de um *parser* para toda a gramática da linguagem em questão, mas somente para alguns pontos chaves, como chamadas de métodos, que podem ser feitas através de um simples programa usando expressões regulares. No estudo de caso discutido no capítulo 5 é mostrado um exemplo desse processo.

Ao ser completada com as informações provenientes do *parser* implementado pelo usuário, a ferramenta está pronta para ser usada. Ela fornece uma interface completa para sua utilização e o usuário interage com ela configurando como deve ser o diagrama final produzido pela ferramenta. Estas funcionalidades serão detalhadas melhor na seção 3.5.

Os metadados fornecidos pelo usuário podem ser divididos em duas categorias: dados de modelo e dados de configuração:

Os dados de modelo informam a composição da árvore de chamadas, ou seja, indicam a referência entre os elementos e comentários que serão exibidos na ferramenta. Um exemplo disso, é que no estudo de caso, existiam

comentários no estilo *Javadoc*[7] no início de cada *procedure* que eram de fundamental importância para o entendimento do funcionamento do sistema. Além disso, era necessário informar os parâmetros de entrada e saída, pois auxiliavam a compreensão da *procedure* pelos desenvolvedores.

Os dados de configuração servem para ajudar a ferramenta no modo como ela deve se apresentar ao usuário. Caso não queira usar o diagrama padrão, é possível alterar a forma como é exibido na tela o diagrama final. Isto provê uma flexibilidade grande à ferramenta. Além disso, é possível modificar a existência, ou não, de alguns menus e funcionalidades. Essas configurações podem ser suficientes para produzir duas ferramentas diferentes entre si, tendo como única diferença o arquivo de configuração.

Os arquivos de metadados são na verdade o coração da ferramenta. Eles são lidos por ela através do formato de arquivos textuais escritos numa linguagem de representação especificadamente feita para isso. A linguagem será apresentada nas seções a seguir.

3.2. A Linguagem de Representação

Para o funcionamento da ferramenta existem dois tipos de dados que precisam ser configurados: os metadados de configuração e os metadados de modelo. Embora estas informações pudessem ser passadas para a ferramenta através de uma API de programação, foi decidido que, para tanto, seriam utilizados arquivos de configuração. O motivo dessa escolha está no fato destes facilitarem a integração com outras ferramentas que não estão necessariamente escritas em Java. Estas ferramentas poderiam ter como saída direta os arquivos de configuração prontos, o que simplificaria o seu uso. Um exemplo disto pode ser encontrado no capítulo 5, que relata um estudo experimental.

3.2.1. Metadados de Configuração

Abaixo segue uma gramática abstrata definindo a linguagem de configuração. Terminais estão em negrito e não terminais estão em itálico. Caracteres literais são apresentados entre aspas simples, Parênteses, “**(** e **)**”,

[7] Javadoc é uma ferramenta desenvolvida pela Sun Microsystems para gerar documentação de API no formato HTML. Em: http://java.sun.com /j2se/javadoc/

indicam um grupo necessário. Colchetes, "**[** e **]**", indicam grupos opcionais. Barra vertical, "|" , separa alternativas.

```
config: configuration '{' stmt_list'}'
stmt_list: stmt [ ';' ] [ stmt_list ]
stmt: (application | edge | menu | ID_NODE) attr_list
attr_list: '[' a_list ']'
a_list: ID_ATTR '=' (value | a_comp_list) [ ','][ a_list ]
a_comp_list: '{'a_comp_item '}'
a_comp_item: value [ ','][ a_comp_item ]
```

Código 1: Gramática da linguagem de configuração

As palavras **application**, **edge** e **menu** são palavras reservadas e identificam um elemento de configuração, juntamente com o **node**, identificado por um *ID_NODE*.

Um *ID_NODE* define o identificador de um *node*. Nas próximas seções será discutido o que é um *node*. Obrigatoriamente um *ID_NODE* deve estar entre aspas duplas e é formato pelos caracteres alfanuméricos, o que no caso de ASCII corresponde a **[A-Za-z0-9]**. Não pode haver espaço.

Um *ID_ATTR* tem uma formação praticamente idêntica ao *ID_NODE*, diferenciando-se pelo fato de não ser escrito entre aspas duplas.

A regra de definição de um *value* é a mesma de um *ID_NODE*, adicionando-se o fato de poder conter os caracteres espaço (' '), undescore ('_') e hífen ('**-**'), que não existem em um *ID_NODE*.

Abaixo segue um exemplo de um arquivo escrito nessa linguagem. Este arquivo foi confeccionado para a ferramenta da avaliação experimental.

```
configuration {
	application[
		title="Dependency Viewer";
		files={"table-table.txt", "procedure-table.txt", "inputs-outputs.txt"}
	];
	"tabela"[
		id=Nome,
		label="Tabela",
		shape="table"
	];
	"procedure"[
		id={"package","name"},
		label="Procedure",
		shape="process",
		detail={input, output, comentarios}
	];
	edge[
		source="table",
		target="target",
		lineStyle="TRACED",
		lineSize="1"
	]
	menu[
		title="Escolha a procedure",
		node="procedure",
		columns={"pakage","Pacote"},
		columns={"name","Nome"},
		elements={"table","Ver tabelas"}
		invisible={"table"}
	]
	menu[
		title="Escolha uma tabela",
		node="table",
		columns={"name","Nome"},
		visible={"procedure"}
	]
}
```

Código 2:Arquivo *dependenciesViewer.config* da avaliação experimental

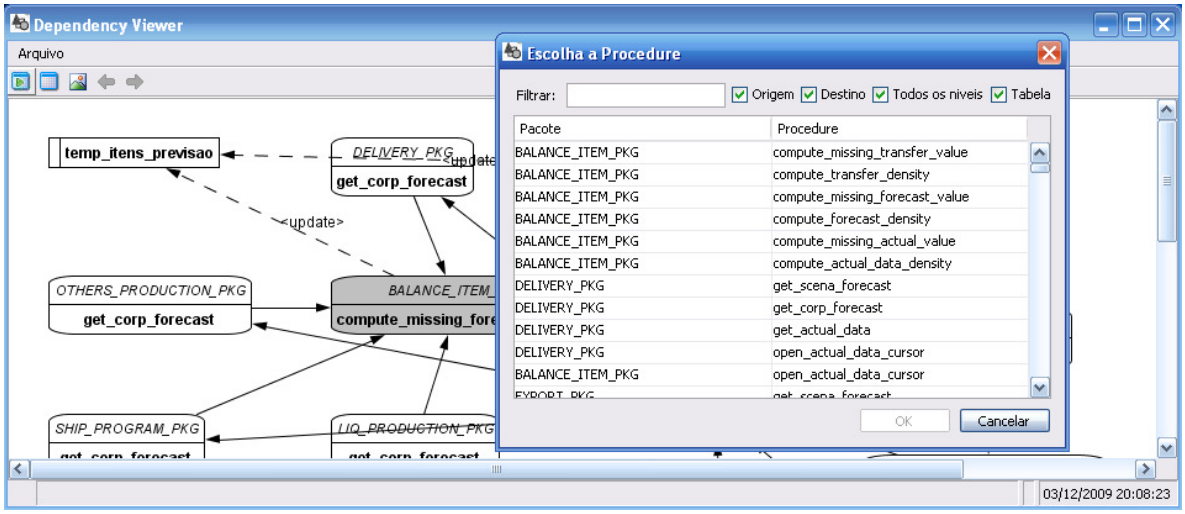


Figura 6: Aplicação resultante do arquivo de configuração anterior

```
configuration {

        application[
                title="Orkut";
                files={"pessoas.txt", "pesoas_detalhes.txt"}
        ]
        "pessoa"[
                id="name";
                label="Pessoa";
                shape="com.coutosistemas.shapes.enums.OvalShapeEnum";
                detail={"Url" }
        ]
        edge[
                source="pessoa";
                target="pessoa";
                lineStyle="TRACED";
                lineSize = "3"
        ]

        menu[
                title="Escolha a pessoa";
                node="pessoa";
                columns={"name","Nome"};
                icon={"pessoa.jpg"}
        ]

}
```

Código 3: Arquivo de configuração de um visualizador de dependências do orkut

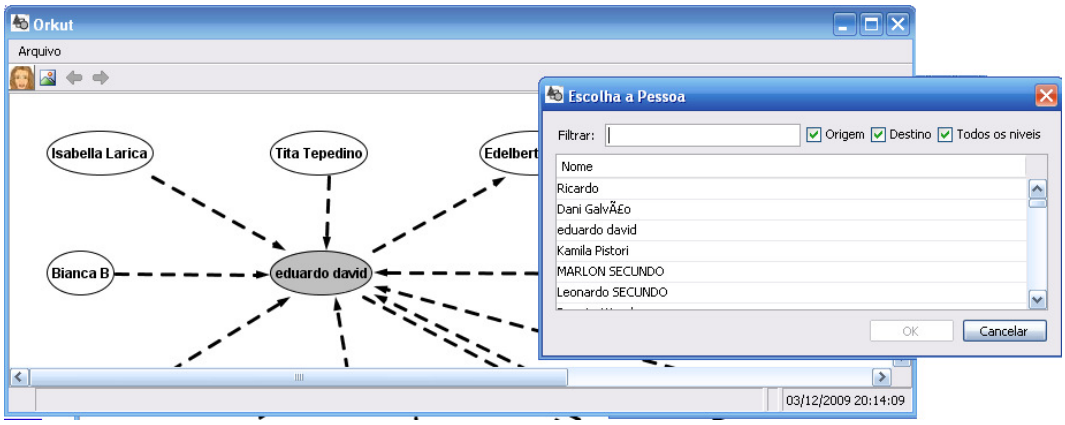


Figura 7: Aplicação resultante do arquivo de configuração acima.

Cada elemento (*Node*, *Menu*, *Edge* ou *Application*) possui um objetivo específico e uma listagem própria de atributos e valores possíveis e serão discutidos a seguir.

3.2.1.1. Node

Um nó, ou node, representa um tipo de objeto do modelo de negócio no qual a aplicação está inserida. Por exemplo, em um **diagrama de classe**, um nó poderia ser a representação de uma **classe**. Na linguagem proposta, é possível identificar um *Node* quando o nome do elemento não for uma das palavras reservadas e/ou estiver escrito entre aspas duplas. No exemplo dado, **"classe"**.

Na linguagem é possível configurar atributos de visualização do nó, regras de formação e quais são as informações de detalhes. Abaixo segue uma tabela com três colunas: nome do atributo, composição e descrição. A coluna composição indica se o atributo é multivalorado ou não. Em caso positivo, na coluna de descrição há o detalhamento do significado destes vários valores.

Nome	Composição	Descrição
id	Sim	Indica a lei de formação do identificador do objeto. No exemplo do diagrama de classes, uma classe não é indexada apenas pelo seu nome, pois podem existir duas classes homônimas, porém em pacotes diferentes. O que as diferenciam e as tornam únicas é a composição do nome do pacote com o nome da classe. Tem de haver, porém, uma contraposição no arquivo de modelo, que deve seguir, sempre que referenciar um objeto deste tipo, a lei de formação configurada aqui. Exemplo: Arquivo de configuração: id={“package”, “nome”) Arquivo de modelo: “br.puc-rio.les”.”NomeClasse”
Label	Não	Indica qual a cadeia de caracteres que representa o nome deste tipo na tela.
Shape	Não	Indica como será a apresentação visual deste elemento. Pode ser umas das formas padrão da ferramenta (*table, process*) ou uma classe que implementa *IInstanceShapeEnum* (Seção 4.1.1)
Detail	Não	Indica quais são as informações de detalhes do elemento. No exemplo do diagrama de classes, poderia ser um *“Javadoc”*. Nos arquivos de modelo, quando for dada a entrada de um detalhe deste objeto, deverá estar especificada como sendo uma informação de *Javadoc*.

Tabela 1: Atributos do elemento *Node*.

3.2.1.2. Edge

Este elemento serve para configurar como será a representação visual da relação de dois nós. A configuração default é uma linha reta cheia direcionada.

Nome	Composição	Descrição
Source	Não	Elemento de origem
Target	Não	Elemento de destino
LineStyle	Não	Indica o estilo da linha que representa o relacionamento. Os valores possíveis são: *TRACED:* Para linhas pontilhadas *DASHED*: Linhas que possuem um ponto seguido de um traço pequeno reto. *CONTINUOUS*: Linha cheia. É o valor *default*, caso não seja informado nada, ou se trata de um valor inválido
LineSize	Não	A grossura da linha
SourceTerminator	Não	Representa como será apresentado o início da reta. Deve ser um *ITerminatorEnum* (Seção 4.3)
TargetTerminator	Não	Representa como será apresentado o fim da reta. Deve ser um *ITerminatorEnum* (Seção 4.3)

Tabela 2: Atributos do elemento *Edge*.

3.2.1.3. Menu

Um elemento de menu especifica uma opção de geração deu um submodelo a partir de um tipo de n*ode*. Mais adiante esta funcionalidade será explicada de modo mais claro (Seção 3.5.1). O importante agora é saber que todos os relacionamentos são traduzidos internamente para um grande grafo, onde cada nó do grafo pertence a um tipo específico, configurado pelo elemento n*ode*. Esta funcionalidade, então, abre uma busca por um objeto a partir de um tipo específico, para que seja apresentado na tela como subgrafo do grafo maior.

Nome	Composição	Descrição
Title	Não	Título da janela principal da funcionalidade.
Node	Não	Especifica qual o tipo de objeto que será buscado nesta funcionalidade. Tem que estar configurado um elemento *Node*, onde o ID_NODE deste elemento é igual ao valor desta propriedade.
Columns	Sim	É um par de valores que especifica as colunas que serão exibidas. Obrigatoriamente são identificadores do nó em questão, sendo o primeiro valor o identificador e o segundo a *Label* de visualização. Por exemplo, no caso do diagrama de classes, haverá uma entrada no arquivo com columns={"package", "Pacote"} e outra com columns={"name", "Nome da Classe"}. Este atributo poderá ser repetido quantas vezes forem os identificadores do nó, sendo apenas uma vez para cada identificador. A ordem de escrita no arquivo é a ordem de ordenação das colunas na tabela de busca.
Elements	Sim	Um objeto de um tipo pode referenciar um objeto de outro tipo. Num DFD, por exemplo, um processo pode ter uma ligação com uma entidade externa. Este atributo serve para que na interface apareça um *checkbox* que desliga e liga, no diagrama final apresentado, a opção de visualizar estes outros tipos relacionados com o *Node* configurado. Os valores deste atributo são os *ID_NODE* dos outros tipos de nós.
Invisible	Sim	*ID_NODE* dos outros tipos de *Node* que vem desligado por padrão. Com o atributo acima, é possível reverter essa configuração, habilitando o *checkbox* do nó invisível, que neste caso, vem desabilitado.
Visible	Sim	Atributo análogo ao de cima, diferenciando-se pelo fato de vir ligado, e não desligado, por padrão.

Tabela 3: Atributos do elemento *Menu*.

3.2.1.4. Application

Este elemento serve para configurar atributos da aplicação como um todo.

Nome	Composição	Descrição
Title	Não	Título da janela principal do programa
Files	Sim	Nome dos arquivos de modelo. Serão lidos apenas os arquivos que tiverem o nome como valor deste atributo.

Tabela 4: Atributos do elemento *Application*

3.2.2. Metadados de modelo

São duas as informações passadas pela configuração de modelo: a cadeia de relacionamentos, informando quem se relaciona com quem, com os seus

possíveis estereótipos e os detalhes daquele objeto, que podem ser apenas um comentário ou um conjunto deles.

Abaixo segue a gramática usada na modelagem dos dados. Ela é muito parecida com a gramática anterior, porém apresenta algumas diferenciações.

```
model: (diagram | detail);

diagram: dgr_instance [';'] [diagram]
dgr_instance: ID_NODE '-' ID_NODE drg_value [';'] [dgr_instance]
drg_value: '[' edge_stmt ']'
edge_stmt: [sttp] ID_OBJECT "->" [sttp] ID_OBJECT [sttp]
sttp: '[' STRING ']'

detail: dtl_instance [';'] [detail]
dtl_instance: ID_NODE '.'ID_COMMENTS dtl_body [';''] [dtl_instance]
dtl_body: object_detail [';'] [dtl_body]
object_detail: ID_OBJECT '{' STRING '}'
```

Código 4: Gramática da linguagem de modelo

Um *ID_OBJECT* deve seguir a regra de formação do identificador do n*ode* especificado no arquivo de configuração. No diagrama de classe, o nome da classe é especificado como sendo pacote mais o nome, ou seja, é a composição de duas informações. Um *ID_OBJECT* deve seguir este formato, sendo escrito na seguinte estrutura: todas os identificadores devem estar entre aspas separadas pelo caractere ponto (‘**.**’). No exemplo do diagrama de classe, poderia ser: “br.puc-rio.inf.les”.”NomeDaClasse” .

Uma *STRING* é um texto livre escrito entre aspas duplas. Caso ela tenha em sua composição o caractere aspas duplas, ele deve ser precedido por uma barra invertida (‘\’).

O *ID_NODE* especificado nesta gramática deve existir no arquivo de configuração inicialmente.

Similar a um *ID_NODE*, um *ID_DETAIL* deve existir como um valor do atributo *detail* do elemento *Node*.

Essa linguagem pode estar escrita em diversos arquivos. A única regra é o fato de ser preciso informar, no arquivo de configuração, os nomes dos arquivos de modelo.

Nas seções a seguir são detalhados os dois tipos de dados passados para linguagem de modelo.

3.2.2.1. Relacionamentos

Trata-se do modo de informar à ferramenta quais são os objetos de determinados tipos e como eles se relacionam. Está especificado em um arquivo de configuração escrito na linguagem apresentada.

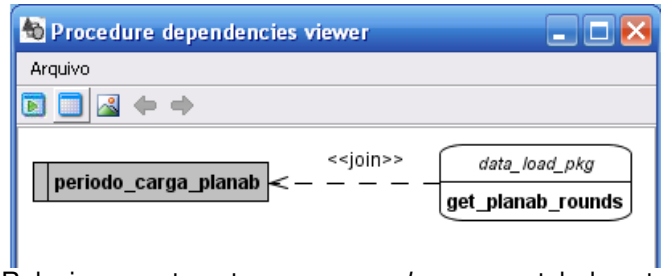


Figura 8: Relacionamento entre uma *procedure* e uma tabela estereotipado

```
diagram{
      "procedure"-"table"[
            "data_load_pkg"."get_planab_rounds"->["<<join>>"]"periodo_carga_planab"
      ]
}
```

Código 5: Exemplo de código gerador da imagem acima

No exemplo acima, o Código 5 é responsável pela entrada da informação de modelo gerador da Figura 8. Como o exemplo é didático, apenas uma linha de relacionamento foi incluída, porém poderiam haver inúmeras linhas, que seriam refletidas do mesmo modo no diagrama exibido. No exemplo dado é usado um qualificador logo após o operador de relacionamento ("**->**"). Nesse caso, a *string* informada é exibida no meio da seta de relacionamento. Caso tivesse um estereótipo escrito no início da linha, esse seria apresentado no início da seta de relacionamento, perto do nó de origem. Ao contrário, se estivesse no final da linha, ele seria exibido no final da linha do relacionamento, perto do elemento destino, neste caso *período_carga_planab*.

Um relacionamento, do ponto de vista do aplicativo, é sempre unidirecional. No Código 5, é apresentado um relacionamento do tipo de elemento *Procedure* com o tipo *Table*. Visualmente, caso não se queira direcionar a linha de ligação dos elementos, é preciso, no arquivo de configuração, dizer que o relacionamento acima não possui terminador no objeto destino. Isto passará a idéia, para o usuário que esteja utilizando a aplicação, de se tratar de um bidirecional.

3.2.2.2. Detalhes

Um detalhe pode ser qualquer informação. Um elemento pode conter diversos tipos de detalhes diferentes, que são separados e exibidos categorizados na interface da ferramenta. O Código 6 é um exemplo de dado que gera a informação exibida na Figura 9.

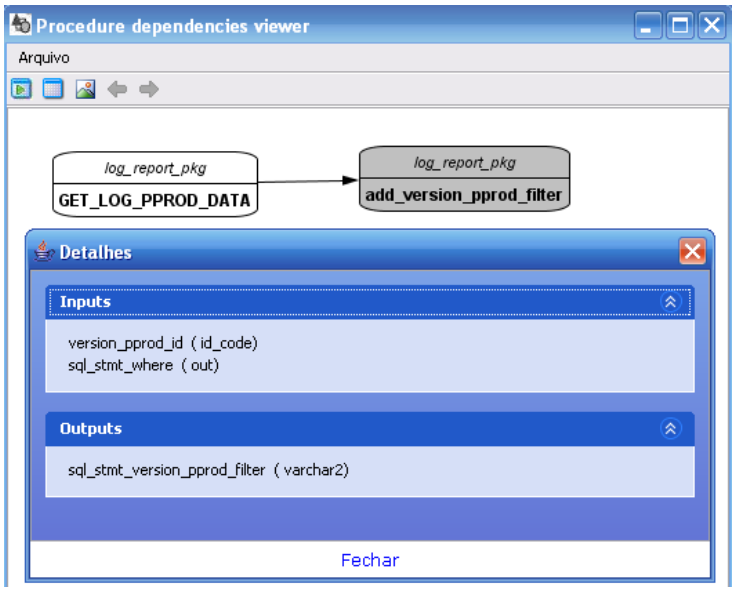


Figura 9: Exemplo de visualização de detalhes

```
detail{
      "procedure"."Inputs"[
            "log_report_pkg"."add_version_pprod_filter" {
                  version_pprod_id ( id_code)
                  sql_stmt_where ( out)
            }
      ];
      "procedure"."Outputs"[
            "log_report_pkg"."add_version_pprod_filter" {
                  sql_stmt_version_pprod_filter ( varchar2)
            }
      ]

}
```

Código 6: Código responsável por gerar a Figura 9.

No exemplo acima, uma *procedure* pode possuir dois tipos de detalhes diferentes: *Inputs* e *Outputs*. Estes tipos devem estar configurados primeiro no arquivo de configuração, para que aqui, no arquivo de modelo, seja informado o conteúdo dos detalhes e quais são os objetos aos quais ele pertence.

3.3. Instalação e uso da ferramenta

O primeiro passo é colocar os arquivos de configuração na mesma pasta do Jar *ProcedureDependency-1.0.jar* (localizado em: http://code.google.com/p/meta-designer/), ou adicionar a pasta onde se encontram os arquivos no *classpath* do sistema operacional. A ferramenta utiliza essas duas heurísticas para ler os metadados. O nome do arquivo de configuração deve ser **dependenciesViewer.config**.

Para utilizar o aplicativo, é necessário ter instalada a máquina virtual do Java, na versão 1.4 (http://java.sun.com), ou mais recente. Após sua instalação, é possível executar o Jar *ProcedureDependency-1.0.jar* (localizado em: http://code.google.com/p/meta-designer/) através da seguinte linha de comando, onde o driver "E" corresponde a pasta onde se encontra o aplicativo

```
E:\java -jar  ProcedureDependency-1.0.jar
```

Uma outra alternativa, no Windows, é clicar com o botão direito no Jar, e selecionar a opção Open With->Java™ Platform SE binary (Figura abaixo).

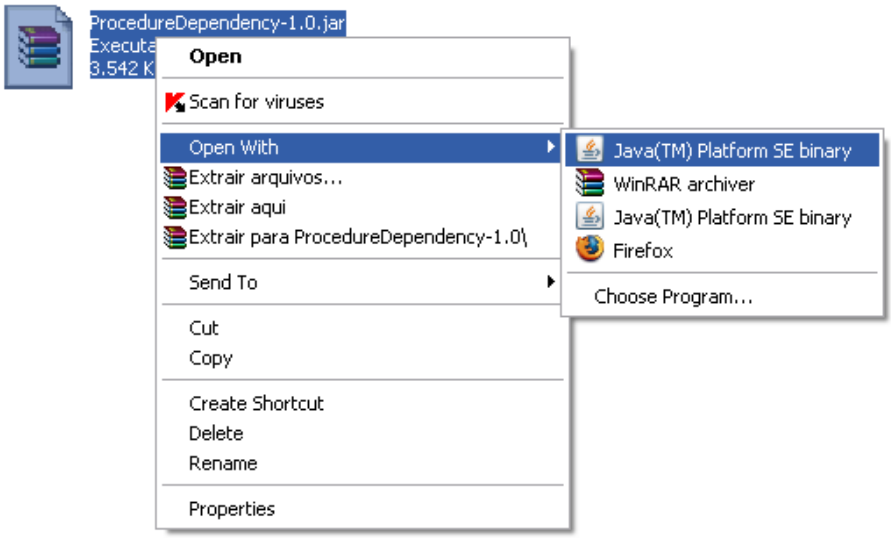


Figura 10: Execução do jar no windows

3.4. Modelo

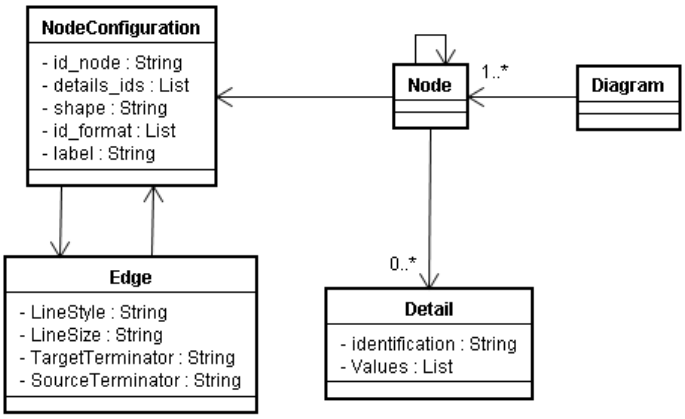


Figura 11: Diagrama de classe simplificado da ferramenta

Na Figura 11 é apresentado um diagrama de classes simplificado para fins didáticos do modelo da ferramenta. Ao ler os arquivos de configuração e modelo, toda a árvore de chamadas é traduzida para um grafo. Isso facilita na hora de caminhar e selecionar os nós que devem ser exibidos. A classe *Node* é um nó do grafo, e possui um ou vários outros nós relacionados. Cada entrada nessa associação representa uma aresta.

Um grafo possui um *NodeConfiguration*, que são as características definidas no arquivo de configuração. Esta classe é origem de um *Edge* que possui outro *NodeConfiguration* como destino. Um *Edge* guarda as informações de metadados do relacionamento de dois nós.

Essas duas últimas classes são os metadados que orientam a criação do grafo. Elas guardam as informações necessárias para visualizar o diagrama no *framework* apresentado no capítulo 4.

A classe *detail* organiza as informações de detalhes de um nó. Também é configurada no arquivo de metadados, pelo atributo *detail* de um *Node* (sessão 3.2.1.1).

Os metadados de *Menu* são interpretados e aplicados no momento de inicialização do sistema, não precisando ser guardados no modelo apresentado acima.

3.4.1.
O processo de desenvolvimento

Apesar de possuir apenas um único recurso humano alocado para o desenvolvimento da ferramenta, algumas precauções foram tomadas para dar um mínimo de qualidade ao produto final gerado. Essas medidas foram essenciais para que, no final, uma ferramenta estável e garantidamente testada fosse disponibilizada para qualquer usuário interessado. A intenção era não produzir um software apenas para o estudo de caso desta dissertação, e sim um artefato que pudesse ser utilizado fora do âmbito acadêmico. Nas próximas sessões seguem algumas práticas adotadas no desenvolvimento.

3.4.2.
Subversion

"Trabalho em equipe não é um problema de divisão e conquista apenas. É um problema de divisão, conquista e integração" (Beck & Andres, 2004). Apesar de, como dito anteriormente, ser desenvolvido por um único programador, foi tomada a precaução de utilização de uma ferramenta que possibilita o trabalho em grupo. Isto garante que, ao precisar da entrada de mais pessoas no projeto, o processo de desenvolvimento não será alterado. Além disso, as ferramentas que fazem este tipo de controle de integração realizam também o controle de versão dos artefatos desenvolvidos. Isto é vital para garantir certa liberdade e organização ao desenvolvedor. O programador sabe que ao fazer determinada alteração, e esta se mostrar equivocada, ele pode, por exemplo, recuperar a versão que não continha esse problema, comparar e identificar onde ocorreram as alterações que incluíram o *bug*[8]. Foi usado, para esta finalidade, o Subversion.

Subversion é uma aplicação *open source* para controle de versões. Também conhecido como SVN, o Subversion foi feito especificamente para ser um substituto moderno para o CVS[9]. Isso significa que o Subversion trata problemas que não são resolvidos pelo CVS e possui conceitos mais

[8] Um Bug é qualquer defeito encontrado em um programa de computador. A palavra é um anglicismo, e traduz literalmente como inseto.

[9] Do inglês *Concurrent Versions System*, o CVS implementa um sistema de controle de versões. Ele observa todo o trabalho e todas as alterações em um conjunto de arquivos e permite que vários desenvolvedores colaborem entre si (de forma totalmente separada)

amadurecidos de controle de versões de arquivos e projetos. Por esse motivo, o uso do SVN é recomendado em vez do CVS.

3.4.3. Issue tracker

Como será visto no capítulo que fala sobre o estudo de caso, o processo de análise de requisito foi iterativo e incremental. Primeiro foi desenvolvida uma versão da ferramenta específica para os interesses do projeto do LES. Ao final, ela foi evoluída para a sua atual fase. Na primeira fase, os *bugs* e as necessidades de evoluções foram cadastradas num sistema controlador de tarefas, fornecido gratuitamente pelo *Google*, chamado *Google Code*[10]. Isso deu agilidade ao processo, visto que o usuário, ou a pessoa que estava querendo uma alteração na ferramenta, não precisa ficar se comunicando por email e esperando uma resposta. Bastava abrir a ferramenta, cadastrar o seu pedido e acompanhar a evolução do estado da tarefa.

O *Google Code* também permite que o desenvolvedor cadastre o tempo estimado de cada tarefa e o tempo real de desenvolvimento. Isso permite o controle fino sobre as tarefas e o andamento do projeto. Como o projeto foi desenvolvido sem um compromisso comercial, e com certa liberdade de requisitos e prazos, esse processo não foi utilizado no desenvolvimento desta ferramenta.

3.4.4. Maven[11]

O Maven é uma ferramenta que está ganhando constantemente espaço no ambiente de desenvolvimento Java. Além de controlar o processo de construção de um software, o Maven oferece uma abordagem abrangente para gerenciar projetos de software. Desde a compilação até a distribuição, incluindo documentação e colaboração entre a equipe, o Maven oferece as abstrações necessárias para encorajar o reuso e diminuir muito do trabalho para fazer os *builds* de um projeto (Massol & Van Zyl, 2006).

No projeto, o Maven foi importante para automatizar a compilação e o controle de dependências do aplicativo. Isso foi de extrema relevância, pois

[10] http://code.google.com

[11] http://maven.apache.org/

como o projeto não usa o conceito de integração contínua, era possível, ao final de cada dia de trabalho, rodar o *build* que compila, executa os testes e gera o instalador. Ele garante que o instalador só é disponibilizado, se todos os testes unitários escritos estiverem rodando e passando. Isso fornece um mínimo de qualidade ao processo.

Alem de automatizar o *build*, no projeto o Maven é responsável por gerar a documentação da *API* e um relatório de cobertura de código.

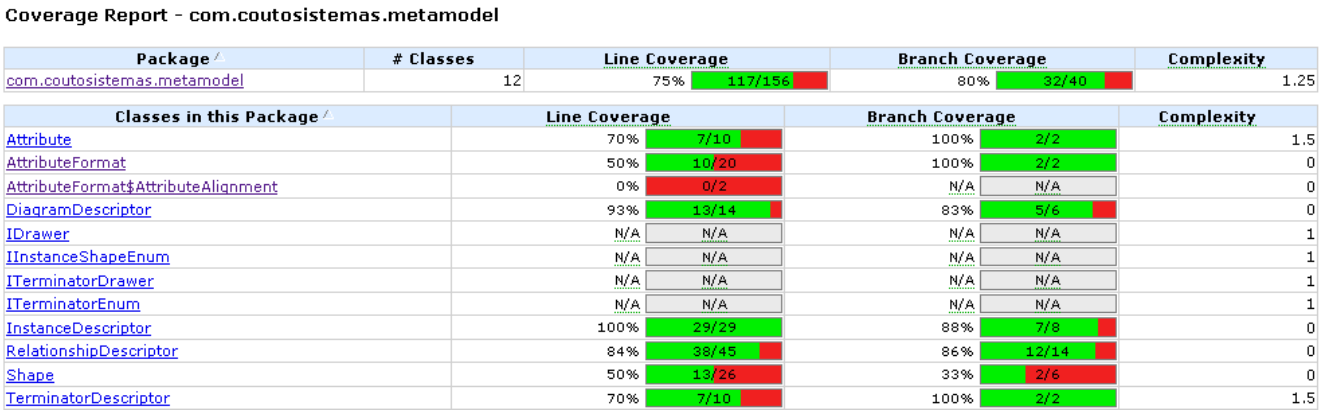
3.4.5. *Test-Driven Development*

Todo código produzido sem testes, é, por natureza, um código legado (FOWLER, 99). Esta afirmativa foi a diretriz básica para todo o desenvolvimento do projeto. O teste é uma ferramenta útil para garantir uma boa qualidade do código, e dar conforto ao desenvolvedor para realizar as devidas evoluções, correções e refatorações. Existem diversas técnicas de testes, e neste trabalho foi utilizado o *Test-Driven Development* (*TDD*).

Também conhecida como programação ou desenvolvimento em que se escreve um teste primeiro, TDD é uma abordagem incremental que envolve a criação de um caso de teste anteriormente à implementação do código necessário para que este passe. Depois disso, o teste é executado para provar que este realmente falha. Usando o conceito de design simples (do inglês *simple design*[12]) o método é implementado de forma a fazer com que o teste de unidade passe. Uma vez que isso aconteça, o programador usa o *refactoring* para limpar o código e mantê-lo sob as regras de design simples. Esses passos são feitos sucessivamente, até que cada funcionalidade esteja pronta.

[12] *Simple Design* é um princípio de XP e quer dizer que o design de um método, classe ou sistema deve ser o mais simples possível. Por exemplo, se um cliente concordar que na primeira versão apenas o usuário "teste" com a senha "123" vai ter acesso a todo o sistema, somente o código exato para que esta funcionalidade seja implementada deve ser escrito. Preocupações com sistemas de autenticação e restrições de acesso não importam neste momento. Um erro comum, no entanto, por parte dos programadores é confundir código simples com código fácil de escrever. Código fácil de escrever é todo aquele que é escrito sem preocupações de design. Nem sempre este tipo de código levará à solução mais simples de design. Esse entendimento é fundamental para o bom andamento de XP. Por isso, todo código fácil de escrever deve ser identificado e substituído através de *refactoring* por código simples.

Ao final do desenvolvimento, verificou-se através de relatórios de coberturas de código (Seção 3.4.4) que em média 80% do código se encontra testado. Os 20% que não possuem testes são classes de interface, onde a produção de testes requer outras abordagens não utilizadas neste trabalho.



Coverage Report - com.coutosistemas.metamodel

Package	# Classes	Line Coverage	Branch Coverage	Complexity
com.coutosistemas.metamodel	12	75% 117/156	80% 32/40	1.25

Classes in this Package	Line Coverage	Branch Coverage	Complexity
Attribute	70% 7/10	100% 2/2	1.5
AttributeFormat	50% 10/20	100% 2/2	0
AttributeFormat$AttributeAlignment	0% 0/2	N/A N/A	0
DiagramDescriptor	93% 13/14	83% 5/6	0
IDrawer	N/A N/A	N/A N/A	1
IInstanceShapeEnum	N/A N/A	N/A N/A	1
ITerminatorDrawer	N/A N/A	N/A N/A	1
ITerminatorEnum	N/A N/A	N/A N/A	1
InstanceDescriptor	100% 29/29	88% 7/8	0
RelationshipDescriptor	84% 38/45	86% 12/14	0
Shape	50% 13/26	33% 2/6	0
TerminatorDescriptor	70% 7/10	100% 2/2	1.5

Figura 12: Relatório de testes de um pacote específico.

3.5. Funcionalidades

O objetivo principal do aplicativo é gerar visualmente diagramas de dependências para as informações fornecidas pelos meta-dados de modelo. Para isso, a ferramenta possui uma série de outras funcionalidades que servem como meio para se alcançar o objetivo final.

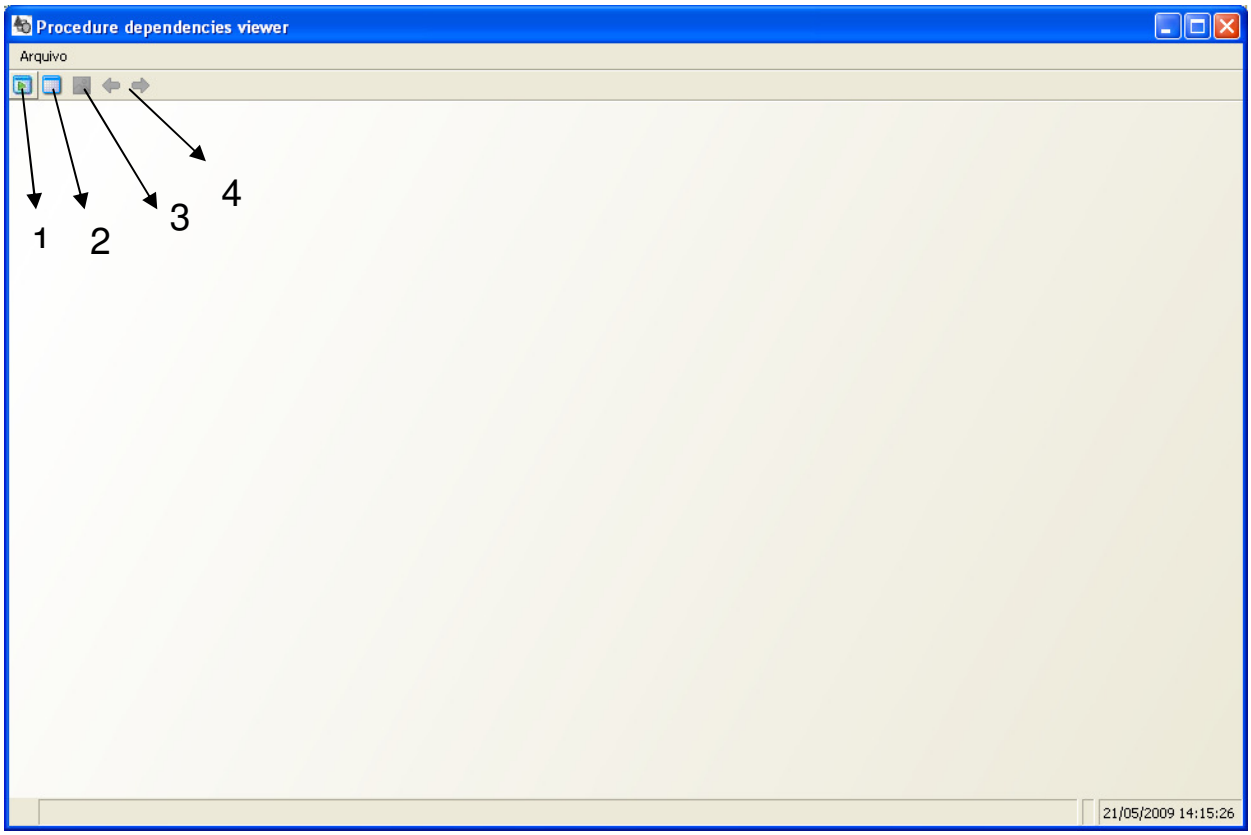


Figura 13: Imagem da tela principal do programa.

Na Figura acima é apresentada a tela de abertura do programa, primeiro contato do usuário com o aplicativo. Essa tela foi retirada da ferramenta configurada para rodar no estudo de caso.

No caso do aplicativo configurado acima, o botão 1 é o responsável por abrir a opção de gerar um diagrama a partir de uma *procedure*. O botão 2 abre o diagrama de dependências a partir de uma tabela, na ordem contrária, mostrando todas as *procedures* que referenciam aquela tabela. Estas duas funcionalidades são originadas pela modelagem de configuração (seção 3.2.1) definida pelo usuário. No aplicativo acima, foram modelados dois tipos de instância de negócio, a *procedure* e a tabela. Simultaneamente, o arquivo de configuração foi preparado para suportar buscas a partir dos dois objetos. Isso se refletiu na interface através destas duas funcionalidades.

O botão 3 exporta o diagrama exibido no momento para uma imagem fora do sistema. Finalmente, os controles indicados por 4 são responsáveis pela navegação no diagrama através dos nós. Segue abaixo um detalhamento de cada uma destas funcionalidades.

3.5.1.
Escolher uma entidade

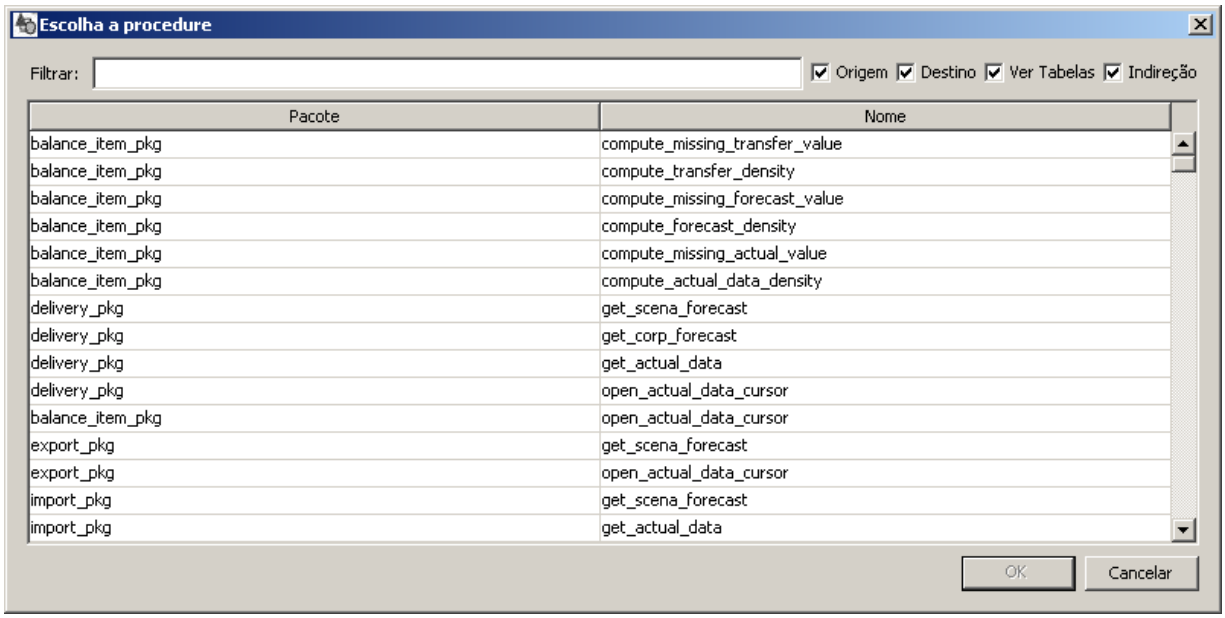


Figura 14: Tela que abre ao selecionar o botão de abrir *procedure*

Como dito anteriormente, toda a árvore e chamadas de todas as entidades é montada em forma de um grafo. Essa funcionalidade serve, então, para gerar um subgrafo a partir de uma entidade de modelo configurada. Este subgrafo, então, é transformado no diagrama procurado. A geração desse grafo menor é essencial, pois não teria como produzir um diagrama legível se toda a árvore de chamadas fosse traduzida em uma única imagem.

A existência dessa funcionalidade está condicionada a uma configuração específica dos metadados, já comentada anteriormente. Na ferramenta do estudo de caso, existem duas buscas por entidades, a busca pelas *procedures,* como pode ser visto na Figura 14, e a busca a partir de uma tabela. Porém, num sistema que possui cinco entidades, poderia haver cinco botões de busca para gerar um subgrafo a partir do objeto da entidade escolhido.

Na tela acima, o campo de texto serve para que o usuário digite uma palavra-chave, que automaticamente é usada para filtrar as linhas da tabela, retirando aquelas que não possuem nenhuma entrada do texto digitado. O filtro é feito "*on the fly",* sem necessidade de apertar nenhum botão ou ação especial. A tabela, por sua vez, mostra todas as informações de identificação da entidade, no caso da *procedure*, pacote e nome.

O botão "OK" se inicia desabilitado e só é liberado quando o usuário selecionar uma entrada na tabela. Ao ser acionado, ele imediatamente some com o diálogo e exibe o diagrama, destacando visualmente o nó que foi escolhido como ponto de partida.

Os campos de marcar ao lado do campo de texto servem para customizar a geração do subgrafo a partir do objeto selecionado. Observe a figura abaixo:

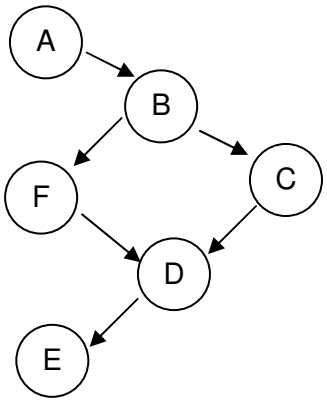


Figura 15: Exemplo de grafo

Na Figura acima, pode-se ver um exemplo de grafo. Imaginando que cada vértice é um objeto de uma entidade, podendo ser uma *procedure* ou uma tabela, no modelo de negócio exemplificado, entende-se melhor as opções de customização existentes. A opção de origem monta um subgrafo do objeto selecionado para frente. A opção de destino monta o grafo inverso, ou seja, do objeto selecionado para trás. “Ver tabela” habilita ou não a exibição das referências às tabelas. Neste exemplo só existem tabelas e *procedure*, porém, caso existisse também *view*, por exemplo, haveria uma opção “Ver View”. A opção de indireção se limita apenas ao primeiro ou a todos os níveis.

Seguem alguns exemplos, pensando sempre no caso de o usuário selecionar a *procedure* “C”. Ao marcar “Origem” apenas, o subgrafo extraído é o C->D->E. Caso seja escolhida a opção de “Destino”, são selecionados os vértices A ->B -> C. Então, com os dois marcados, além da indireção, o subgrafo fica A->B->C->D->E. Se a “Indireção” não estiver selecionada, será exibido apenas o primeiro nível, no caso “B->C->D”. Repare que, nesse caso, o vértice “F” nunca irá ser adicionado no grafo de C, pois ele não pertence à sua árvore direta de chamadas. As regras apresentadas acima servem para qualquer busca de qualquer entidade, pois no final todos os objetos, indiferente ao seu tipo, são vistos do mesmo modo pelo algoritmo de geração de subgrafos, ou seja, são meros vértices de um grafo.

3.5.2. Navegação

Na função explicada na seção anterior, o usuário define como será apresentado visualmente o diagrama final. Para isso, ele escolhe um elemento da tabela e no diagrama exibido ele é destacado, para indicar que ele é o ponto de partida para o grafo gerado. Após a geração do diagrama, é comum o usuário querer navegar por um dos nós relacionados, seja para fazer uma análise de impacto, seja para verificar quais elementos serão alterados. Essa funcionalidade pode ser vista através da opção exibida na figura abaixo.

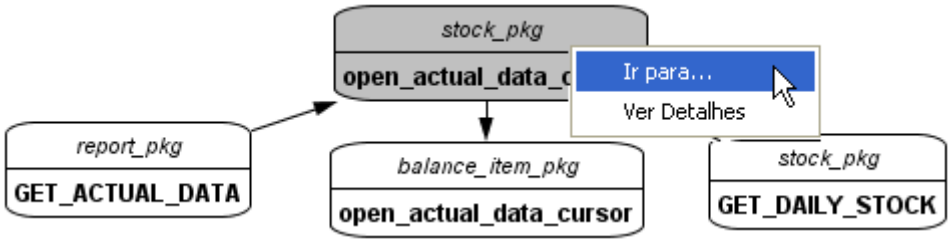


Figura 16: Imagem da funcionalidade de navegação

Para acessá-la basta clicar com o botão direito do mouse no elemento que será a base do novo diagrama gerado. Este herdará todas as configurações feitas para o diagrama original. Na verdade, é como se o diagrama atual fosse originado do botão disponível na interface do programa, do mesmo modo que o diagrama anterior, tendo como única diferença o nó base do grafo.

Com as constantes navegações, o usuário pode, através da opção 4 da Figura 13, voltar para o diagrama anterior ou, caso já tenha feito isso, prosseguir para o diagrama que se apresentava na tela.

Essa funcionalidade fornece muita agilidade ao processo de entendimento do modelo gerado e a análise de impacto que a ferramenta proporciona, visto que dada a complexidade da árvore de chamadas, nem sempre é possível exibir todos os níveis numa única imagem.

3.5.3. Detalhamento

As funcionalidades de escolha de entidade e navegação são muito úteis para a análise de impacto. A funcionalidade de exibição de detalhamento é vantajosa para a manutenção do sistema, pois facilita o entendimento do mesmo.

Assim como a busca por entidade, esta funcionalidade precisa ser inserida no arquivo de configuração do sistema, como visto na seção 3.2.1. Além disso, precisa ter correlação nos meta-dados de modelo, visto na seção 3.2.2. Com tudo corretamente configurado, basta ir ao nó escolhido e clicar com o botão direito, como pode ser visto na figura abaixo.

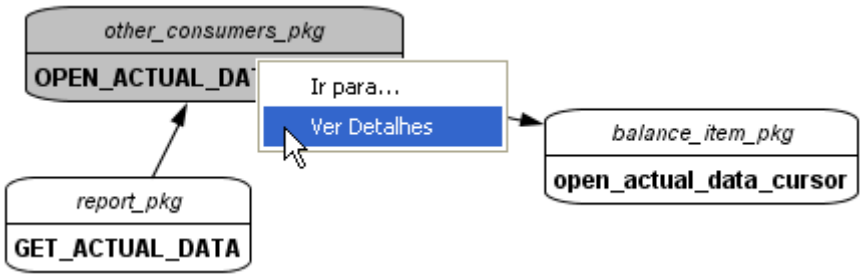


Figura 17: Opção de exibição de detalhes

Ao selecionar essa opção, é exibido um diálogo como o da tela abaixo.

Figura 18: Diálogo com os detalhes do elemento.

Esse diálogo pode ser qualquer informação que o configurador da ferramenta queira passar, desde um simples *javadoc*, até as entradas e saídas de uma *procedure,* como no exemplo acima.

3.6. O diagrama gerado

Como pode ser visto anteriormente, é possível customizar o diagrama exibido através de meta-configurações. Porém, já são fornecidos dois *shapes* básicos baseados no Diagrama de Fluxo de Dados, ou DFD. O diagrama final produzido é parecido com o exibido na Figura abaixo.

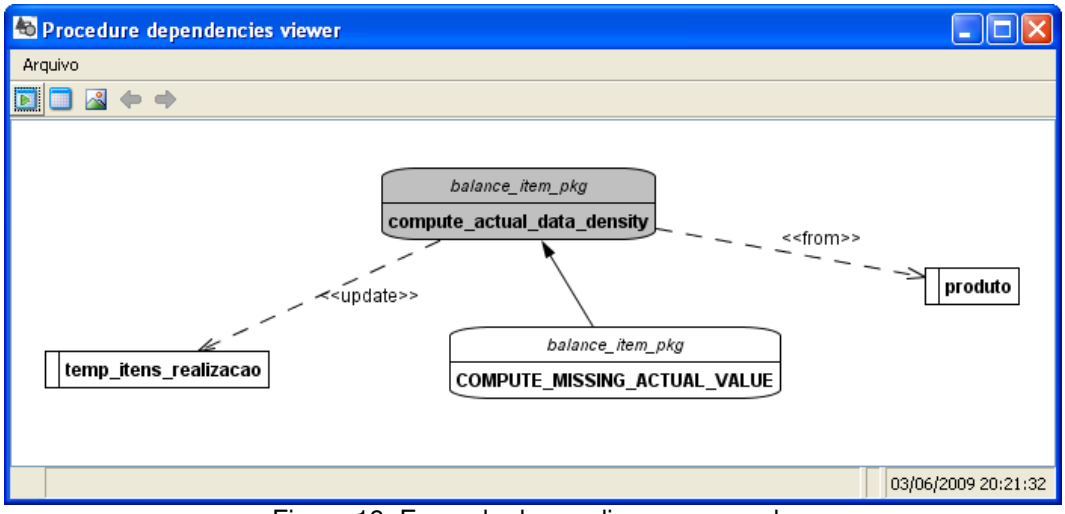


Figura 19: Exemplo de um diagrama gerado

Diagrama de Fluxo de Dados (DFD) é uma das ferramentas mais usadas de modelagem funcional de sistemas, e pode ser entendido como uma rede que ilustra como circulam os dados no seu interior. Um DFD é um diagrama baseado em etapas, com aumento gradativo de detalhes. Ele utiliza o princípio da modularização e da hierarquização.

Basicamente são quatro elementos que formam o DFD:

Entidades Externas: São categorias lógicas de objetos ou pessoas que representam origem ou destino de dados, e, que acionam um sistema e/ou recebem informações. Existem fora da fronteira do sistema. Podem ser, por exemplo, pessoas, sistemas ou unidades departamentais.

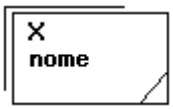


Figura 20: Entidade Externa

Fluxos de dados: são o meio por onde os dados e as informações trafegam. Modelam a passagem de dados. A seta indica o sentido do fluxo e tem o nome do fluxo associado. Não é possível o fluxo de entidade para entidade, entidade para deposito de dados, deposito de dados para deposito de dados, e assim por diante, pois sempre deve envolver algum Processo.

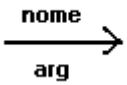


Figura 21: Fluxo de dados

Processos: São módulos do sistema. São atividades que transformam e fazem uso do fluxo de dados. Modificam as informações que entram para fluxo de saídas. Cada processo tem um nome único.

Figura 22: Processo

Depósito de Dados: São locais onde os dados são armazenados, como arquivos físicos, tabelas gerenciadas por um SGBD, etc. Como o processo, possui um nome único.

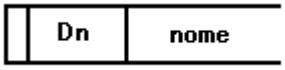


Figura 23: Depósito de dados

O motivo de mencionar esse modelo é que o diagrama default gerado normalmente será uma modificação do DFD, dado que a ferramenta suporta nativamente as suas representações. A fundamentação principal é o amplo conhecimento da ferramenta pelos analistas, servindo assim de um ótimo instrumento de comunicação, vital num processo de desenvolvimento [Aktas, 1987]. Apesar de o argumento anterior ter se enfraquecido ao longo dos anos devido ao apogeu da UML e modelos afins, pensando no usuário final da solução apresentada para o problema levantado na motivação deste documento, continua fortíssimo. Isto porque o foco são linguagens estruturadas, e não existe um diagrama mais conhecido e utilizado para modelá-las.

Em vez do DFD, outros diagramas foram pensados como alternativas para a base do projeto proposto. Um exemplo é o diagrama de seqüência, e que é amplamente difundido na comunidade. Ele permite descrever diferentes processos que ocorrem no sistema, modelando a troca de mensagens (ou eventos) entre os objetos deste mesmo sistema. Com as devidas modificações, ele seria um ótimo candidato a ser base do novo diagrama. Uma vantagem em relação ao DFD é o fato de ele modelar, na sua essência, a ordem e seqüência em que os eventos ocorrem. A sua principal desvantagem é a falta de escalabilidade. Ele foi feito para modelar apenas um processo ou caso de uso. A tela fica muito poluída quando colocado num contexto global, devido a sua forma rígida de ser apresentado, com as linhas verticais e processos na parte superior. O DFD provê maior flexibilidade na organização do layout, gerando uma melhor

distribuição e conseqüentemente melhor entendimento da mensagem a ser passada.