Anderson Pereira Ataides

Desenvolvimento de sistemas em Linux

Uma análise das ferramentas de desenvolvimento para Linux

por Anderson Pereira Ataides

Desenvolvimento de sistemas em Linux: Uma análise das ferramentas de desenvolvimento para Linux

Dedicatória

Dedico este trabalho ao meu amigo Anibal Santos Jukemura, que em nossa segunda viagem a Lavras, ao ver o que eu conseguia fazer no Linux, pediu-me que escrevesse sobre desenvolvimento em Linux descrevendo as ferramentas que utilizo para ter bons resultados.

Também dedico ao prof. Joaquim Quinteiro Uchôa na esperança que este trabalho possa contribuir em sua luta pela divulgação do software livre no mercado.

Agradecimentos

Agradeço todos meus professores do ARL pelo conhecimento adquirido durante o curso, principalmente ao prof. Joaquim que além de passar conhecimento, passou também incentivo e estímulo para usarmos e divulgarmos o software livre.

Agradeço meu orientador, o prof. Heitor, pela paciência em orientar um aluno tão enrolado como sei que sou, e pelas valiosíssimas dicas de desenvolvimento sem as quais não conseguiria concluir este trabalho.

Agradeço aos meus filhos e esposa pela tolerância em me ver em casa e mesmo assim me deixar estudar para concluir meu curso e principalmente este trabalho. Também os agradeço porque eles são meu estímulo a buscar novos conhecimentos para, conseqüentemtente, conseguir melhores oportunidades de negócio sendo possível dar-lhes melhor condição de vida.

Agradeço a meu pai por sempre estar me incentivando a seguir com meus estudos. Também o agradeço por me dar broncas na hora certa, quando me mostro desestimulado a cumprir uma tarefa. Também vai um agradecimento à minha mãe (*in memorian*), que de onde quer que ela esteja, com certeza intercedeu para que eu tivesse sucesso em minha jornada de estudos.

Um agradecimento especial ao meu irmão Heberts pela sua insistência para que eu esteja sempre buscando melhor qualificação para poder entrar no meio acadêmico.

Finalmente quero agradecer a Deus por ter me iluminado na busca de conhecimentos para desenvolver este trabalho.

Índice

Lista de Tabelas

Lista de Figuras

Capítulo 1. Introdução

Nos dias de hoje, onde as empresas estão sob pressão constante, seja dos fornecedores que querem vender mais caro, seja dos clientes que querem comprar mais barato, qualquer economia que se consiga obter, pode ser fundamental para se manterem funcionando e, principalmente, dando lucros. Atentos a essas exigências e em busca de produtividade a baixo custo, muitos diretores buscaram na informática um ponto de apoio para auxiliá-los em suas estratégias de negócios.

Mesmo com esta pressão e mesmo com a necessidade de aumento de produtividade, muitas empresas, a maioria pequena, ainda não conseguiram fazer esse investimento. Entre os motivos alegados pelos diretores estão os seguintes:

Preço do equipamento

> Com a rápida e constante evolução das máquinas, e conseqüente pressão para vender as obsoletas, o que se vê são equipamentos de informática cada vez mais baratos e com muitas facilidades para pagamento. Portanto esta justificativa não deve mais assustar tanto o empresário.

Falta de mão de obra

> Em uma rápida busca em jornais e em instituições como SENAC, SENAI, entre outras que oferecem treinamento, pode-se constatar que há inúmeros cursos técnicos, que estão "despejando" profissionais no mercado. Contando também as faculdades (somente em Goiânia são mais de dez) que anualmente formam mais turmas, observa-se que na verdade há um *excesso* de profissionais disponíveis no mercado. É claro que entre tanta gente, existem os bons e os maus profissionais, mas sabendo escolher, é possível conseguir um profissional que consiga realmente auxiliar no processo de informatização.

Falta de tempo para treinamentos

> Com tantas escolas de informática oferecendo cursos em vários

horários diferenciados, não há como usar a falta de tempo para não informatizar. Algumas destas escolas vão mais além: fecham turmas dentro da empresa, cobrando mais barato pelo pacote e dão o treinamento *in loco*, ou seja, dentro da empresa.

Mas a maior dificuldade é realmente o investimento inicial para a informatização. A grande maioria das empresas brasileiras é pequena e não tem caixa suficiente para bancar essa informatização. Mesmo vencida a barreira do custo dos equipamentos, existem os custos da mão de obra, dos próprios treinamentos e um outro custo que quase sempre é deixado de lado, mas que representa a maior fatia dos investimentos: o do *software*. Na maioria dos casos apenas o custo do *software* de gerenciamento é considerado no levantamento de custos.

Os custos com outros tipos de *software*, como sistema operacional e suítes *office*, geralmente são desprezados, sendo estes sistemas considerados básicos e que é uma obrigação do fabricante do equipamento fornecê-los instalados. Realmente os fabricantes fornecem os sistemas, mas com um pequeno detalhe: *piratas*. Se a empresa tentar fugir dessa pirataria, será preciso desembolsar alguns milhares de reais para ter sua máquina legalizada.

Felizmente, com o surgimento do Linux (http://www.linux.org), que popularizou a filosofia do *software* livre (http://www.fsf.org) ou ainda do *software open source (http://www.opensource.org)* ficou bem mais fácil legalizar esses sistemas "básicos". A empresa não precisa mais se preocupar com as "licenças de uso", ou com o medo de instalar um programa em várias máquinas sob o risco de violar os contratos de tais licenças.

Na filosofia do *software* livre, ao obtê-lo, seja gratuitamente através de um *download*[1], ou pela bondade de um amigo, ou mesmo pela compra, o *software* é do usuário e com o *software* ele pode fazer o que desejar, inclusive alterá-lo desde que as alterações mantenham o *software* livre. Assim diferentemente do *software* proprietário, onde é necessária uma licença de uso para cada instalação, ao adquirir o *software* livre, o usuário adquire também o direito de instalá-lo em quantas máquinas ele achar necessário.

Portanto, pode-se concluir que a utilização de *software* livre pode

trazer para a empresa grandes vantagens, dentre as quais pode-se destacar a economia e a independência do fornecedor. Reduzindo os gastos com a informatização, fica mais fácil para empresas pequenas colocar a informática à disposição de seus colaboradores a fim de ganhar produtividade e aumentar sua eficiência. Para empresas maiores, o caixa extra gerado pela economia obtida pela adoção de sistemas livres, pode ser direcionado para áreas realmente afins com o negócio da empresa.

1.1. Motivação

Uma das grandes motivações para o desenvolvimento desse trabalho foi o fato de existirem poucos, ou nenhum aplicativo comercial para Linux. Em se tratando de *software* livre o problema ainda é mais grave, pois os desenvolvedores desse tipo de aplicativo ganham dinheiro com a venda do *software*. Existem pelo menos dois modelos de negócio seguidos pelos fornecedores desse tipo de *software*: venda e aluguel. Na verdade, o que quase sempre acontece, é que a empresa adquire o *software* e depois o fornecedor oferece uma manutenção em troca de uma taxa mensal para atualizações do produto. Concluindo, acaba-se comprando e alugando o *software* ao mesmo tempo.

Outro grande problema no modelo proprietário, é que na maioria dos casos, os fornecedores cobram por licença de uso, ou seja, eles cobram a utilização em cada máquina da rede, o que onera mais ainda o custo de informatização. No caso de sistemas alugados, a história não é muito diferente, pois geralmente o valor do aluguel é calculado de acordo com o número de máquinas em que o sistema vai funcionar.

O modelo proprietário ainda apresenta o problema de prender o usuário a um único fornecedor, que detém o poder sobre o sistema. Ao usuário cabe apenas o direito de utilizar o sistema. O usuário depende do fornecedor para instalar, dar treinamento e manutenção, e ainda efetuar eventuais alterações que forem necessárias para que o sistema atenda melhor às suas necessidades. Se a alteração solicitada não for interessante para outros clientes (do fornecedor do sistema), pior ainda, pois eles vão deixar a alteração em segundo plano sob a justificativa que eles têm outras mais prioritárias para desenvolver.

A filosofia do *software* livre muda o poder das mãos do fornecedor para as mãos do cliente principalmente pelos seguintes motivos:

O *software* livre tem código aberto

> Teoricamente, qualquer programador com conhecimento na linguagem de programação na qual o sistema foi desenvolvido pode efetuar alterações e adaptá-lo a qualquer situação.

O *software* livre pode ser instalado em outras máquinas

> A partir do momento em que o usuário obtém o *software*, ele pode instalá-lo onde desejar, ou seja, ele pode instalar o sistema em uma ou em todas as máquinas da empresa sem ter que pagar mais por isto.

Esse trabalho foi elaborado baseando na experiência passada em uma pequena empresa, a Casa das Carretas, localizada em Goiânia-GO. A Casa das Carretas se encaixa no perfil da maioria das pequenas empresas brasileiras com potencial de crescimento porém com caixa restrito, o que dificulta gastos em investimentos. Com o crescimento natural no volume de dados da empresa, surgiu a necessidade de fazer um *upgrade* em uma máquina. Na época, todos os sistemas eram legalizados, desde o sistema operacional, passando pelo sistema de rede até o sistema gerencial. Como a máquina veio com um sistema operacional mais evoluído, decidiu-se evoluir toda a infra-estrutura da empresa. Em pouco tempo, houve a necessidade de um novo *upgrade* e, após um minucioso levantamento, decidiu-se pela utilização do *software* livre.

Ao partir para uma plataforma livre, surgiu outro dilema: o sistema gerencial era escrito para o outro sistema operacional. Tentou-se fazê-lo funcionar sob um emulador, mas diferenças no tratamento do sistema de arquivos levaram a corrupção dos arquivos e conseqüente perda de dados. Como o fornecedor do sistema não mostrou muita disposição em elaborar um sistema que funcionasse na plataforma adotada pela empresa, decidiu-se então partir para o desenvolvimento de um novo sistema com funcionalidade similar.

Tomada a decisão, foi feita uma busca para encontrar ferramentas que facilitassem o processo de desenvolvimento em Linux, de forma que fosse tão rápido como no Windows. No sistema da Microsoft, existem ferramentas consagradas de desenvolvimento e, devido à essa disponibilidade, não há como negar que é relativamente fácil desenvolver sistemas para Windows.

Depois de muita leitura e pesquisa em *sites* na *Internet*, leitura de documentos *how to*[2] e de alguns livros, foi possível aprender a utilizar algumas ferramentas e então elas foram adotadas no desenvolvimento dos sistemas da Casa das Carretas. Para mostrar aos programadores que desenvolver aplicativos em Linux pode ser tão fácil como no Windows, foi desenvolvido esse trabalho.

1.2. Objetivos

Existem alguns motivos que emperram a proliferação do Linux pelos computadores das empresas. Na grande maioria delas, o sistema é usado apenas nos servidores, enquanto as estações continuam com o sistema proprietário anterior, na maioria dos casos o Windows. Isso porque apesar de ter inúmeros aplicativos disponíveis, não tem o principal para a empresa: um *software* de gerenciamento.

Um dos objetivos desse projeto é incentivar os programadores a completar a gama de *software* livre para empresas comerciais. Já existe o sistema operacional, o banco de dados, as linguagens de programação as suítes *office*, todos livres, mas ainda não existem ofertas de *software* de gerenciamento livre. Existem muitos sistemas eficientes, mas nenhum distribuído sob a filosofia do *software* livre.

Para atingir esse objetivo, a proposta deste trabalho é mostrar que o desenvolvimento de sistemas em Linux pode ser tão fácil como o desenvolvimento em Windows, a plataforma mais adotada pelos fornecedores de *software* comercial. Essa "facilidade" porém só pode ser confirmada, se for possível mostrar ferramentas que ajude o desenvolvedor em suas tarefas.

No desenvolvimento deste trabalho, serão apresentadas algumas das

várias ferramentas disponíveis em Linux para auxiliar no desenvolvimento de sistemas. Além de mostrar a existência da ferramenta, também será mostrado como trabalhar com tais ferramentas. Para mostrar que as ferramentas apresentadas realmente produzem bons resultados, será descrito um sistema de vendas simples e a partir dessa descrição será mostrado como usar as ferramentas para desenvolver o sistema desde a modelagem de dados e do sistema, até a implementação.

Portanto, neste trabalho será visto que há como produzir sistemas em Linux de forma fácil, rápida e econômica, pois todas as ferramentas utilizadas são livres. Aliando a facilidade decorrente do uso das ferramentas certas com a economia obtida por elas serem livres, a intenção é convencer os desenvolvedores que é perfeitamente viável e vantajoso desenvolver sistemas para Linux. Viável pela facilidade e economia pela utilização das ferramentas descritas e vantajoso porque apresentar um sistema feito para Linux pode se tornar um diferencial competitivo para o desenvolvedor, tanto pelo pioneirismo como pela possibilidade de gerar economia para seus clientes.

1.3. Organização deste trabalho

Este trabalho descreve com o máximo de detalhes possível, o processo de desenvolvimento de um sistema em Linux. Ele contempla desde a sua especificação, obtida em conversas com o usuário, passando pela modelagem até a implementaço do sistema propriamente. Além da documentação do sistema, também procurou-se descrever de uma forma resumida e objetiva as ferramentas utilizadas em cada fase do projeto.

O Capítulo 2 apresenta as ferramentas escolhidas e os motivos que levaram à sua escolha. O Capítulo 3 apresenta o ponto de partida para o desenvolvimento do sistema: sua especificação. Ele detalha o processo de vendas de uma forma bem objetiva supondo em primeira análise o funcionamento perfeito, sem exceções a serem tratadas.

Descrito o problema e escolhidas as ferramentas, parte-se para o desenvolvimento do sistema. O Capítulo 4 apresenta os diagramas UML resultado da modelagem do sistema e o Capítulo 5 mostra a modelagem de

dados que representa a estrutura do banco de dados do sistema. Além de apresentar o resultado (modelos) também é feita uma breve explicação de como obter o resultado com a ferramenta utilizada.

O Capítulo 6 aborda a implementação do sistema. Para este trabalho, supõe-se que se tenha um mínimo de conhecimento da linguagem C/C++. Este capítulo mostra que, apesar de ser conhecida como uma linguagem difícil, é possível trabalhar com ela com relativa facilidade quando se usa as ferramentas adequadas.

O Capítulo 7 apresenta como desenvolver a documentação do sistema, tanto para o desenvolvedor como para o usuário. São mostrados de forma bem resumida e objetiva como usar o DOxygen e o DocBook para gerar manuais e páginas de ajuda para documentar um sistema.

Finalmente no Capítulo 8, são apresentadas as considerações finais sobre o trabalho, algumas das dificuldades enfrentadas, conslusões e projetos futuros.

Este trabalho foi escrito para que a leitura seja no mínimo interessante e que ajude a derrubar alguns mitos sobre o desenvolvimento de sistemas em Linux. Se esse material incentivar o desenvolvimento de um sistema em Linux mesmo que não seja livre, pode-se considerar que foi dado um grande passo no rumo do cumprimento de seu objetivo, pois convenceu alguém que realmente é possível adotar o Linux também nas estações de trabalho das empresas.

Notas

1. *Download* é o processo de baixar ou copiar arquivos de um computador remoto, no caso a *Internet*

2. *How to* é um guia que explica como fazer alguma coisa. Existem inúmeros documentos *how to* disponíveis na *Internet* que ensinam desde a configuração básica de um computador até a construção de *clusters*. A maioria deles pode ser vista na página *The Linux Documentation Project* (http://www.ldp.org/)

Capítulo 2. A escolha das ferramentas

Este capítulo mostra como foi feita a escolha das ferramentas utilizadas no desenvolvimento do sistema. Serão apresentados os critérios de escolha, uma breve apresentação das ferramentas e uma justificativa pela sua escolha.

2.1. Critérios de escolha

Até chegar ao ponto de começar o desenvolvimento do sistema, existe um processo que geralmente gera polêmicas: a escolha das ferramentas. A polêmica está no fato de que não há como dizer que uma ferramenta é melhor que a outra, mas que uma ferramenta *para mim* foi mais adequada, ou seja, por mais que se tente ser imparcial, a impressão pessoal tem muita influência na escolha. No sistema Windows é relativamente fácil fazer essa avaliação, pois existem ferramentas consagradas que facilitam muito o processo de desenvolvimento. Entretanto, no domínio do *software* livre, a escolha é mais trabalhosa pois existem ferramentas ótimas mas incompletas e outras completas mas difíceis de usar.

A escolha das ferramentas se deu considerando as seguintes variáveis:

Mercado

> Quando se fala de mercado, significa que é verificado se a ferramenta é disponibilizada pelas distribuições Linux, se ela funciona nas distribuições e ambientes mais populares e se tem respaldo da comunidade *open source* para continuidade de seu desenvolvimento.

Facilidade de instalação e uso

> Variável importante, pois não adianta ser eficiente se não oferecer facilidade de uso e instalação. Muitas pessoas desistem de usar sistemas livres (principalmente Linux) justamente por achar que eles são difíceis de usar.

Popularidade

O quesito popularidade é importante porque quanto mais pessoas utilizam a ferramenta, mais fontes de informação existirão para solucionar alguma dúvida.

Opinião pessoal

É importante deixar claro que esta variável foi considerada, pois assim fica mais fácil compreender a adoção de uma ferramenta em detrimento de outra que também atenda às outras variáveis.

Para o completo desenvolvimento do sistema, foram adotadas cinco ferramentas além do ambiente nativo do sistema e do sistema gerenciador de banco de dados. Para todas elas foram feitos alguns testes e analisadas as variáveis descritas. A escolha foi feita tentando o máximo de imparcialidade e deixando a opinião pessoal apenas como critério de desempate.

2.2. O ambiente

Diferentemente do que acontece no Windows, o Linux por si só não oferece um ambiente de trabalho, um gerenciador de janelas único. Existe um servidor gráfico que oferece as rotinas básicas de manipulação da tela, um cliente que acessa o servidor gráfico e diversos gerenciadores de janela que oferecem ao usuário um ambiente de trabalho. Entre esses gerenciadores de janela estão o WindowMaker (http://www.windowmaker.org/), o Blanes (http://labdid.if.usp.br/~blanes/), o AfterStep (http://www.afterstep.org/) o Gnome (http://www.gnome.org/) e o KDE (http://www.kde.org/). Na verdade, desde que se tenha as bibliotecas requeridas instaladas, um aplicativo pode funcionar em qualquer um dos gerenciadores disponíveis, ou seja, um aplicativo feito especificamente para o KDE roda no Blanes desde que as bibliotecas do KDE estejam instaladas no sistema.

A escolha do ambiente gráfico é polêmica porque a opinião pessoal tem um peso muito forte na decisão. Talvez esse seja o ponto onde a opinião pessoal tem peso mais forte. Como o sistema funciona independente do ambiente, a polêmica pode ser amenizada, mas o simples fato de ter que

instalar as bibliotecas do outro ambiente pode gerar um ponto de discórdia entre os usuários. Entre os váios ambientes disponíveis foi escolhido o KDE.

O primeiro grande motivo pela escolha do KDE é que ele está presente em praticamente todas as distribuições Linux do mercado, o que vai aumentar a compatibilidade do sistema. O segundo motivo é que o KDE é o ambiente gráfico padrão adotado por várias das distribuições mais populares como SuSE/Novell (http://www.suse.com), Mandrake (http://www.mandrakesoft.com), Xandros (http://www.xandros.com), e a brasileira Conectiva (http://www.conectiva.com.br). O terceiro e talvez o mais importante motivo pela escolha do KDE é que a biblioteca na qual ele se baseia, o Qt (http://www.trolltech.com), é totalmente orientada a objeto.

Um outro grande ponto a favor da dupla Qt/KDE é a documentação. Todas as classes são bem documentadas e a documentação não se restringe apenas a descrever as classes, seus métodos e seus atributos. Principalmente na biblioteca Qt, existem inúmeros exemplos e tutoriais que ensinam como utilizar as classes, o que facilita bastante para o desenvolvedor entender como uma classe é utilizada em conjunto com outras que compóem um *software*.

Assim foi escolhido o KDE, lembrando que para utilizá-lo no desenvolvimento e na hora de executar o sistema pronto não significa que o ambiente gráfico utilizado tenha que ser o KDE. Para que o sistema funcione é suficiente que as bibliotecas dele estejam instaladas independente de qual gerenciador de janelas o usuário estiver usando.

2.3. Banco de dados

A escolha de um sistema gerenciador de banco de dados (SGBD), não depende somente do desenvolvedor do sistema, nem só de critérios puramente técnicos. Dependendo do sistema escolhido, o desenvolvedor pode tornar o seu *software* inviável para algumas pequenas empresas que têm pouco ou quase nenhum recurso disponível para investimento.

A Casa das Carretas, empresa que serviu de modelo para o sistema descrito neste trabalho, é um bom exemplo desse fato. Em meados de 2001, quando era feito um estudo para fazer um *upgrade* das máquinas e dos programas que seriam utilizados, o sistema gerenciador de banco de dados foi um dos que entraram no orçamento. Segundo orçamentos realizados diretamente da Microsoft, Oracle e de revendas locais, só o custo da infra-estrutura para desenvolvimento (sistema gerenciador de banco de dados, linguagem de programação e ferramentas de modelagem) utrapassaria a casa dos vinte mil reais, e a empresa não teria condições na época de bancar esse custo.

Para fugir do alto custo das soluções proprietárias, foi feita uma pesquisa para encontrar um *software* de baixo custo que conseguisse atender as exigências técnicas para o desenvolvimento dos sistemas. Felizmente as distribuições Linux fornecem pelo menos dois sistemas gerenciadores de banco de dados, sendo eles o MySQL (http://www.mysql.com/) e o PostgreSQL (http://www.postgresql.org/). Ambos são ótimos candidatos a serem adotados no desenvolvimento de um sistema, pois implementam instruções SQL padrões e bons recursos de segurança.

Depois de uma análise nas empresas que ministram cursos em Goiânia, como SENAC (http://www.go.senac.br/), Sistemas Abertos (http://www.sistemasabertos.com.br/) entre outras, pode-se observar que a maioria delas ofereciam curso de MySQL e poucas ofereciam PostgreSQL. O MySQL era oferecido em curso separado ou em conjunto com alguma linguagem de programação como PHP, ou Java. Dessa forma, concluiu-se que escolhendo o MySQL, seria mais fácil encontrar profissionais no mercado para solucionar algum problema que pudesse surgir no desenvolvimento dos sistemas da Casa das Carretas.

Apesar de ter escolhido o MySQL, não há muita dificuldade na adoção de outro sistema. Devido à forma como o sistema foi projetado é fácil de alterar o SGBD. Para isso, deve-se alterar o código de algumas classes para fazer a conexáo com outro SGBD. Não é necessário alterar as classes de mais alto nível que tratam das *interfaces* com o usuário ou alguma outra que *usa* o banco de dados.

2.4. Ferramentas de modelagem

Para desenvolver um bom sistema, não é suficiente ter uma boa idéia, sentar à frente do computador e começar a implementar o sistema partindo diretamente para sua codificação. Para sistemas simples essa estratégia pode funcionar, mas para sistemas maiores uma boa documentação deve ser elaborada para orientar a implementação além de possibilitar a validação com o usuário de uma forma mais clara e consistente. O desenvolvimento de um sistema de computador deve funcionar como na construção de uma casa, onde existem os projetos de arquitetura que são aprovados pelos donos e os projetos de estrutura, fundação entre outros que direcionam a construção.

Atualmente dois modelos são adotados pelos desenvolvedores de sistema: a análise estruturada e a análise orientada a objetos. Ambas possuem suas vantagens e desvantagens, mas optou-se pela adoção, da análise orientada a objetos por dois motivos: a biblioteca do ambiente escolhido e a disponibilidade de ferramentas. Com relação à biblioteca, o KDE é fundamentado no Qt que é orientado a objetos e assim, usando a análise também orientada a objetos, ela fica mais consistente com a implementação do sistema. No caso da disponibilidade de ferramentas, todas as ferramentas analisadas ou oferecem recursos para ambos modelos ou apenas para o modelo orientado a objetos.

Dentro da análise orientada a objetos, a modelagem do sistema é feita usando-se a UML - *Unified Modeling Language* (http://www.uml.org) e portanto a ferramenta escolhida deve oferecer os recursos necessários para essa modelagem. Das várias ferramentas disponíveis, chegou-se a quatro que atenderam os requisitos de facilidade de instalação e utilização. As quatro ferramentas analisadas foram o dia (http://www.lysator.liu.se/~alla/dia), o TCM - *Toolkit for Conceptual Modeling* (http://www.cs.utwente.nl/~tcm/), o ArgoUML (http://www.argouml.org) e o Umbrello (http://www.umbrello.org). Das ferramentas analisadas, foram usados o TCM para a modelagem de dados e o Umbrello para desenhar os diagramas UML. O TCM foi escolhido principalmente por permitir documentar cada elemento do diagrama (mesmo que de forma primária) e gerar diagramas em vários formatos entre eles o *PostScript*.

A escolha do Umbrello se deu basicamente por dois motivos. O primeiro deles é que das ferramentas analisadas só o Umbrello implementa o diagrama de seqüência. O outro motivo é que o Umbrello é parte do projeto KDE a partir da versão 3.2 e, conseqüentemente, é disponibilizado junto com o ele. Assim, o Umbrello se torna a ferramenta mais conveniente para utilização.

2.5. Linguagem de programação e ambiente de desenvolvimento

Para decidir qual a linguagem de programação a ser utilizada, vários fatores precisam ser considerados e ainda assim a escolha pode não ser a ideal. Entre as linguagens disponíveis para o Linux, estão o Kylix (http://www.borland.com/kylix/index.html), Free Pascal (http://www.freepascal.org/), GNU Pascal (http://www.gnu-pascal.org/), GNU C/C++ (http://www.gnu.org/) entre outras. O Kylyx, apesar de ter uma versão gratuita, não é um *software* livre[1] e então ele foi descartado.

Um dos grandes problemas na escolha da linguagem é que as linguagens propriamente não proporcionam um ambiente de desenvolvimento rápido nem rotinas que tratam do "desenho" das telas ou de acesso ao banco de dados de forma nativa. Em tempos onde a produtividade tem que ser maximizada, ter que se preocupar, além do *layout* das telas, em escrever todas as rotinas de manipulação de telas e dados, implica em uma perda de tempo muito grande por parte do programador. Considerando as linguagens de programação disponíveis para Linux, apenas o Kylix, que foi descartado, oferece esse tipo de facilidade ao programador.

Então o problema não é só a escolha da linguagem de programação, mas a escolha de um ambiente que proporcione ao programador uma maneira rápida de codificar o sistema. Felizmente, os desenvolvedores dos ambientes gráficos, como o Gnome (http://www.gnome.org/) e KDE (http://www.kde.org/) disponibilizaram as bibliotecas para que os programadores de aplicativos pudessem utilizá-las em seus sistemas, liberando-os da tarefa de implementar rotinas para manipular janelas. Como foi escolhido o KDE, a linguagem escolhida foi o C++ utilizando o Qt (http://www.trolltech.com/) como biblioteca básica para gerar a *interface* com o usuário.

Além de fornecer um conjunto completo de classes para manipular janelas, o Qt também oferece o Designer (http://www.trolltech.com/products/qt/designer.html), um aplicativo que possibilita o desenvolvimento da *interface* com o usuário de forma visual, ou seja, é possível desenhar as telas sem precisar se preocupar como será o código fonte que será gerado para compilação.

Usando apenas o Qt e as ferramentas que ele disponibiliza, é possível gerar um sistema completo. No entanto, para desenvolver um sistema dessa forma, o programador precisa saber como utilizar todas as ferramentas para gerar o projeto completo. Além disto, ele também terá que saber trabalhar com *Makefiles* para adicionar outras bibliotecas (como as de acesso ao banco de dados) e para fazer a instalação do sistema.

Para que a programação seja feita de forma mais fácil e rápida, o programador deve concentrar todo seu trabalho no sistema modelado e detalhes básicos como manipular os *Makefiles* deveriam ser excluídos do escopo de seu trabalho. Na busca por facilitar o trabalho de programação, foram encontradas algumas ferramentas muito interessantes como o Anjuta (http://anjuta.sourceforge.net), o KDevelop (http://www.kdevelop.org) e o KDEStudio (http://freshmeat.net/projects/kdestudio/). O Anjuta foi descartado porque ele é próprio para desenvolver aplicativos para o Gnome e o ambiente escolhido foi o KDE. O KDEStudio é um ótimo candidato, mas ele não é disponibilizado em algumas distribuições Linux, o que restringe a sua utilização. A escolha então foi o KDevelop.

O KDevelop oferece bons recursos para auxiliar o programador a codificar o sistema. Ele é integrado ao Qt, e portanto é fácil acrescentar novas *interfaces* criadas com o Designer ao sistema que está sendo desenvolvido. Ele também é muito competente para trabalhar com classes oferecendo um navegador onde, ao clicar sobre o nome de um método ou atributo, o arquivo fonte correspondente é aberto na posição onde o elemento selecionado está definido. Com relação aos *Makefiles*, o KDevelop deixa para o programador apenas a tarefa de informar as bibliotecas que ele necessita e onde localizá-las. Ao acrescentar uma classe, ou uma biblioteca, o projeto é modificado para que os *Makefiles* sejam corretamente ajustados para compilar o sistema corretamente. Além dos recursos citados, o KDevelop ainda tem a grande vantagem de fazer parte do projeto KDE,

e conseqüentemente está presente na maioria das distribuições Linux.

Resolvido o problema da *interface*, sobrou apenas a dificuldade de acesso ao banco de dados com o C++. Assim como nos ambientes gráficos, os sistemas gerenciadores de banco de dados, disponibilizam uma API[2] (*Application Program Interface*) nativa em C para ser usada no desenvolvimento de um sistema. Para usar essa API com o C++, deve-se informar no arquivo contendo o fonte qual o *header* contendo os protótipos das funções e, no KDevelop, qual a biblioteca necessária para compilar o aplicativo corretamente.

Finalmente, a escolha da linguagem C++ também sofreu influências do próprio Linux e seus aplicativos, a maioria escrita em C. Já que a grande maioria dos programadores Linux usam C, por que procurar outra linguagem? Além do mais o C vem acompanhando qualquer distribuição Linux do mercado.

2.6. Documentação do sistema

Nenhum sistema é realmente bem desenvolvido se não tiver uma documentação que facilite, tanto ao desenvolvedor quanto ao usuário a compreensão e utilização de todas as partes que formam o sistema. Além disso, a forma na qual essa documentação é disponibilizada, também pode ser de fundamental importï¿½cia na "popularização" do sistema.

As ferramentas de modelagem e de implementação escolhidas, facilitam bastante o trabalho "braçal", mas não são tão eficientes para disponibilizar a documentação. Para se tornar um software popular e para atrair o interesse de programadores da comunidade *open source*, todo o sistema deve ser documentado de modo que o desenvolvedor consiga entender o funcionamento e implementação antes mesmo de ver o código fonte.

Para documentar as classes, os arquivos *header* foram todos documentados de acordo com as especificações do DOxygen (http://www.stack.nl/~dimitri/DOxygen/). O DOxygen é um sistema de documentação para programas escritos em C, C++, Java entre outras linguagens. Ele lê os arquivos fonte e gera uma documentação para a API

do sistema. Essa documentação pode ser gerada em vários formatos como páginas HTML (*Hypertext Markup Language*), PDF (*Portable Document Format*) e RTF (*Rich Text Format*). Logicamente, os comentários na codificação do sistema não foram esquecidos, mas a documentação estilo DOxygen foi a que mereceu mais atenção, pois as classes ficam documentadas facilitando o seu entendimento sem nem mesmo ver o código fonte.

Os dois motivos que levaram à adoção do DOxygen foram a integração do KDevelop e a sua utilização no projeto KDE. O KDevelop pode usar outros sistemas como o kdoc ou javadoc para documentar a API do projeto, mas como padrão ele sugere o DOxygen. O fato do KDevelop usar o DOxygen como padrão provavelmente é decorrência dele ter sido adotado pelo projeto KDE para documentar a sua própria API.

Para elaborar a documentação do sistema, manual do usuário e até mesmo este trabalho, poderia ter sido utilizado qualquer processador de texto como o OpenOffice (http://www.openoffice.org) ou o Microsoft Word (http://www.microsoft.com/products/office/). Porém, seguindo as especificações da *TLDP - The Linux Documentation Project (http://www.tldp.org/)*, foi utilizado o DocBook (http://www.docbook.org/), que gera a documentação em HTML, PDF, LaTeX, PostScript entre outros formatos a partir de um mesmo fonte XML (*Extensible Markup Language*) ou SGML (*Standard Generalized Markup Language*).

Notas

1. Conforme a *Free Software Foundation (http://www.fsf.org/)*, o fato de o *software* ser gratuito não implica que ele seja livre. Para ser livre, o *software* precisa no mínimo ter o código fonte disponível, o que não acontece com o Kylix.

2. API é um conjunto de rotinas que permitem a utilização de um *software* ou componente de *software*. Assim, ao dizer que um sistema gerenciador de banco de dados disponibiliza sua API, significa que existe uma biblioteca que implementa as rotinas que devem ser usadas para ter acesso ao banco de dados.

Capítulo 3. Descrição do sistema

Este capítulo, descreve o sistema a ser desenvolvido conforme as informações obtidas junto aos usuários do sistema. O processo de vendas, que é onde o sistema deve atuar, pode ser dividido em três fases: negociação, acerto no caixa (pagamento) e expedição. A descrição apresentada será usada para a modelagem do sistema.

3.1. A negociação

Durante o expediente, o vendedor atende os clientes via telefone ou diretamente no balcão. O vendedor, de acordo com as informações passadas pelo cliente, consulta a lista de mercadorias no computador, verifica a quantidade disponível e o preço da mercadoria. Para realizar essa consulta o vendedor pode pesquisar pela descrição, pelo grupo ou pelo próprio código do produto.

Os preços deverão ser dependentes da forma de pagamento. Existe um preço de venda sugerido (que considera um prazo de 30 dias) e, dependendo da forma de pagamento, pode ter mais ou menos descontos. Para não prender o vendedor a preços e descontos pré-fixados, o sistema deve permitir que ele conceda descontos até o limite máximo estabelecido para vendas a vista e para vendas a prazo. Para conceder o desconto, o vendedor pode digitar o preço da mercadoria diretamente ou ainda conceder desconto ao final da venda (desconto no rodapé da nota).

O cadastramento dos dados do cliente também está condicionado à forma de pagamento. Para vendas em dinheiro, o cadastramento é opcional e o vendedor tem orientação para oferecer o cadastro para posteriormente o cliente poder comprar faturado. Para vendas a prazo ou em cheque (a vista ou pré-datado) é feito um cadastro criterioso do cliente. Para fazer o cadastro do cliente, além dos dados completos (nome, endereço, telefone, etc), são exigidas 3 referências comerciais onde o cliente tem cadastro para compras a prazo. O vendedor tem autorização para fazer o cadastro completo do cliente, mas não tem autorização para liberar o crédito.

Dependendo da situação do cliente (inadimplência), o departamento financeiro pode bloquear o seu cadastro de forma a não permitir vendas

a prazo. Em casos extremos pode ser desejável bloquear por completo a venda para um determinado cliente - o departamento financeiro usa esse artifício para obrigar o cliente a, no mínimo, conversar com o gerente financeiro antes de efetuar qualquer compra.

Concluídos todos os passos (negociação e cadastramento), o vendedor faz o registro da venda, gerando uma ordem de venda que é impressa no estoque e fica aguardando o pagamento no caixa. O cliente então é encaminhado ao caixa para efetuar o pagamento e posteriormente à expedição para retirada da mercadoria. Nas vendas por telefone, em que será feita uma entrega, o entregador deve antes se dirigir ao caixa para entregar uma via da ordem de venda que fica como pendência para acertar na sua volta.

3.2. Acerto no caixa

Depois de concluída a negociação por parte do vendedor, o cliente é encaminhado ao caixa para fazer o acerto da venda. No caixa, o sistema deverá exibir uma lista das vendas que foram negociadas e que ainda não foram acertadas. O cliente fornece o nome e a venda, que deverá estar relacionada na lista, será baixada e liberada para expedição. Feito o acerto, uma nota (ou cupom) fiscal é emitida para a venda. Caso o cliente queira mudar a forma de pagamento, ele deve ser encaminhado de volta ao vendedor para ver se é possível a alteração (lembrando que os preços dependem da forma de pagamento escolhida).

Para o caso de entregas, o entregador deverá informar ao caixa quais as vendas que ele está levando para que elas sejam marcadas como parcialmente acertadas. Quando o entregador retornar, ele entrega ao caixa o recebimento propriamente dito. De posse do recebimento, a venda deverá ser acertada e então é eliminada da lista de pendências no caixa.

3.3. Expedição

Quando a venda é negociada pelo vendedor, uma ordem de venda é emitida no estoque para que os estoquistas possam separar as peças para expedição. Se a venda for para ser entregue, o estoquista convoca um en-

tregador, informa as vendas que ele vai entregar e o encaminha ao caixa para fazer o acerto preliminar das vendas. Quando a mercadoria for para despachar, o estoquista deve informar o número de volumes que a venda gerou (a nota fiscal contém essa informação). Como alternativa, o próprio estoquista pode emitir a nota fiscal que será impressa no caixa.

Na expedição, o sistema deve exibir uma lista das vendas que foram acertadas no caixa. O estoquista somente poderá expedir a mercadoria se a venda estiver na relação apresentada pelo sistema, ou seja, vendas acertadas. No caso de entregas, o acerto parcial é suficiente para o estoquista fazer a expedição, pois o acerto real será feito no retorno do entregador. Depois de expedida a mercadoria, o estoquista então faz a entrega da venda removendo-a da lista de pendências.

3.4. Relatórios de acompanhamento

Para acompanhamento e totalizações, o sistema deverá ser capaz de emitir pelo menos os seguintes relatórios:

- Lista de preços (por grupo, por descrição);
- Relatório de vendas por grupo e produtos;
- Relatório de vendas por cliente;
- Relatório de vendas por forma de pagamento;
- Totalização de vendas com comissão por vendedor;

Capítulo 4. Modelagem do sistema

Neste capítulo, será apresentada a modelagem do sistema elaborada a partir de sua descrição. Serão mostrados alguns dos diagramas elaborados para o sistema e como obtê-los usando a ferramenta escolhida, o Umbrello.

4.1. Diagrama de caso de uso

Segundo [FURLAN] a partir da especificação do sistema, é possível montar cenários que ajudam a compreender melhor as exigências do sistema. Esses cenários, na orientação a objetos, são modelados com a ajuda dos diagramas de caso de uso. Os diagramas de caso de uso fornecem um modo de descrever o funcionamento do sistema e como ele interage com o mundo externo, em uma visão de alto nível. No momento da elaboração dos diagramas de caso de uso, não é necessário saber como o sistema implementa o caso de uso, mas apenas que ele vai responder a uma requisição de usuário.

A partir de agora será mostrado como criar um diagrama de caso de uso com o Umbrello. Ao abrir o Umbrello, ele está pronto para desenhar um diagrama de classes. Para criar um novo diagrama de caso de uso, deve-se ir ao menu `Diagram`⟶`New`⟶`Use Case Diagram` [1]. Uma caixa de diálogo será exibida para seja digitado o nome do diagrama de caso de uso. O nome dado ao diagrama deste trabalho foi `kvendas`.

Informado o nome do diagrama, o Umbrello estará pronto para desenhar o diagrama de caso de uso. Para criar um elemento no diagrama, como um caso de uso mostrado na Figura 4-1 deve-se selecionar o ícone correspondente na barra de ferramentas localizada à direita da janela do Umbrello e clicar na área do diagrama. A associação entre os elementos é feita selecionando o tipo de ligação na barra de ferramentas e clicando nos elementos que serão conectados um ao outro. Depois de criados os elementos do diagrama (atores, casos de uso e ligações), pode ser feita a documentação de cada um deles. A documentação é feita em uma janela como a mostrada na Figura 4-2 que se abre ao dar um duplo clique sobre o elemento desejado

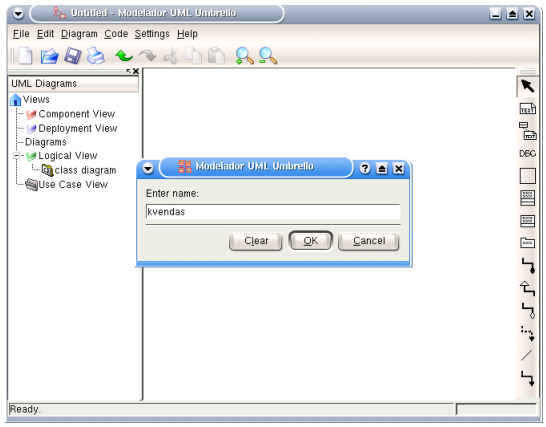

Figura 4-1. Umbrello - criando um diagrama de caso de uso

Os casos de uso devem ser documentados com o máximo de detalhes para que seja possível compreender claramente sua função no sistema. Eles devem fornecer uma descrição consistente e clara sobre as tarefas que devem ser cumpridas pelo sistema sem contudo especificar como estas tarefas serão implementadas.

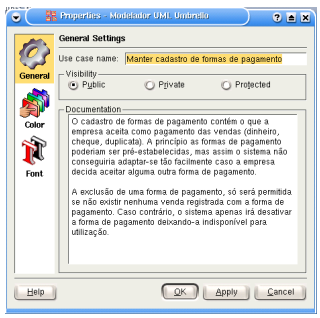

Figura 4-2. Umbrello - documentando um caso de uso

Depois de criado o diagrama, é possível imprimi-lo ou exportá-lo como uma imagem. O fato de poder exportar o diagrama como uma imagem é interessante pois fica fácil inclui-lo em outros documentos (como neste trabalho). O Umbrello consegue exportar o diagrama para os formatos de imagens mais populares como PNG, JPG, e BMP. Para criar a imagem, deve-se clicar com o botão direito do mouse no meio do diagrama e selecionar a opção `Export as Picture`. Depois é só dar um nome à figura para criar o arquivo.

A Figura 4-3 apresenta o diagrama de casos de uso completo do sistema.

Figura 4-3. Diagrama de casos de uso do sistema

4.2. Diagrama de classes

A elaboração do diagrama de classes não pode ser feita em um único passo, e nem é um processo isolado na modelagem do sistema. [FURLAN] diz que inicialmente são determinadas as classes principais do sistema e seus atributos. Essas classes são obtidas a partir do estudo da especificações do sistema e dos casos de uso. Depois de criar essas classes preliminares elas passam por um processo de normalização do modelo visando sua estabilidade e integridade. Além disso, deve-se buscar modelar classes mais simples para que elas possam ser mais fáceis de copreender, documentar e principalmente, implementar.

Continando o processo de desenvolvimento, saindo da análise inicial e partindo para o projeto do sistema, pode haver necessidade de criar novas classes que não foram previstas no modelo preliminar. As novas classes também devem integrar o diagrama de classes para que ele não fique inconsistente com a implementação, que deve seguir o que foi concebido no projeto.

Para criar um diagrama de classes no Umbrello, o primeiro passo é selecionar o menu `Diagram`⟶`New`⟶`Class Diagram`. Para criar uma nova classe deve-se selecionar o ícone correspondente a classe na barra de ferramentas à direita da janela do Umbrello e clicar na área do diagrama. Para usar uma classe esistente, deve-se arrastá-la da árvore mostrada à esquerda do Umbrello para a área do diagrama. A figura Figura 4-4 mostra o Umbrello modelando o diagrama de classes.

Da mesma forma que foi feito no diagrama de casos de uso, para ligar duas classes seleciona-se o tipo de ligação desejado na barra de ferramentas à direita e clica-se nas duas classes que devem ser ligadas. Atributos do relacionamento como multiplicidade e rótulo podem ser alterados clicando-se sobre o relacionamento com o botão direito do mouse e, no menu que aparece, selecionar a opção `Properties`. Uma janela então se abre para que as propriedades do relacionamento sejam alteradas. À esquerda existem algumas opções que, quando selecionadas, mudam a página de propriedades do relacionamento. A primeira página (*General*) permite alterar o nome do relacionamento e documentar o relacionamento. Na página *Roles* é possível alterar os atributos relativos a cada uma das

classes do relacionamento.

Para documentar uma classe, bem como criar os seus atributos e seus métodos, deve-se dar um duplo clique sobre a classe desejada. O Umbrello mostra uma janela similar à janela de propriedades do relacionamento, porém com opções diferentes. Na página *General*, é possível alterar o nome da classe e escrever a sua documentação. Na página *Attributes* são informados os atributos da classe. Na página *Operations* são criados os métodos da classe. Se os relacionamentos já foram feitos no diagrama, a página *Associations* vai exibi-los.

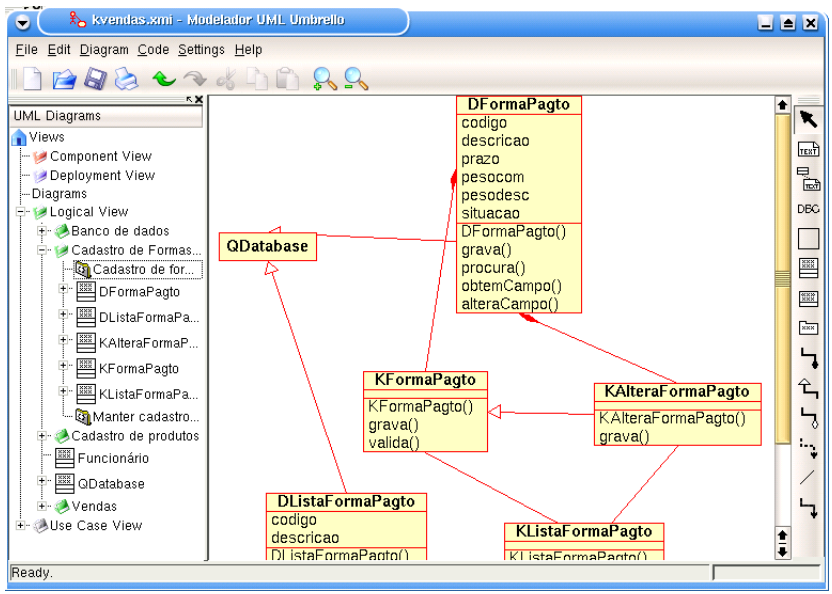

Figura 4-4. Umbrello - Diagrama de classes

A definição das classes no Umbrello é muito importante porque baseado nestas informações, ele gera o código fonte contendo o esqueleto da classe, restando ao programador apenas codificar cada método da classe

e efetuar alguns ajustes para acessar as bibliotecas necessárias para compilar o sistema. Como foi escolhida a linguagem de programação C++, o Umbrello vai gerar os arquivos *headers* (.h) e os arquivos fonte (.cpp). Ao gerar os arquivos, o Umbrello inclui também toda a documentação que foi inserida na definição da classe no formato reconhecido pelo DOxygen.

Para ilustrar o funcionamento do Umbrello, será mostrado como definir a classe `DListaFormaPagto`. Esta classe é responsável por obter os dados das formas de pagamento do banco de dados para que seja elaborada uma lista que será exibida ao usuário. Depois de criada a classe, deve-se dar um duplo clique sobre ela para abrir a janela de propriedades (Figura 4-5) que é onde são informados os atributos, métodos e documentação da classe.

A primeira tarefa é documentar a classe, indicando para que ela serve e fornecendo uma visão geral sobre seu funcionamento. Não é necessário neste momento entrar em detalhes sobre os atributos e métodos porque eles terão sua própria documentação.

Figura 4-5. Umbrello - documentando uma classe

Feita a documentação da classe, pode-se informar os seus atributos. Para isso deve-se selecionar a página *Attributes* e clicar no botão *New Attri-*

bute. Uma janela como a da Figura 4-6 é aberta para definir o novo atributo. Agora é só definir as propriedades do atributo e confirmá-lo. O Umbrello volta à tela anterior com o atributo criado na lista. Para documentá-lo, clica-se sobre ele e digita-se a documentação na área de texto *Documentation*, como mostrado na Figura 4-7.

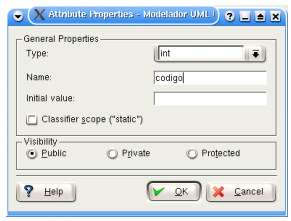

Figura 4-6. Umbrello - Criando um novo atributo

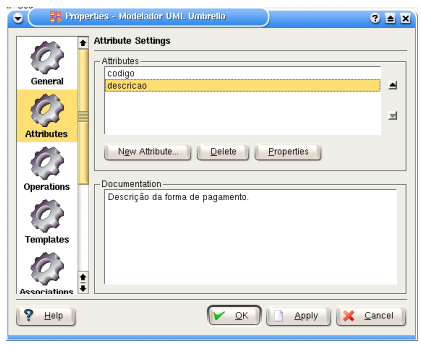

Figura 4-7. Umbrello - documentando um atributo

Depois de incluídos e documentados os atributos, pode-se incluir os métodos da classe. Para isso o primeiro passo é selecionar a página *Ope-*

rations. Selecionada a página dos métodos, deve-se clicar no botão *New Operation* para criar um método. Uma tela como a da Figura 4-8 é aberta onde o método será definido. Na tela das propriedades do método, pode-se incluir também os parâmetros que o método vai precisar para seu processamento. Para incluir um parâmetro, deve-se dar um clique no botão *New Parameter*. A tela de propriedades do parâmetro é similar à das propriedades de atributo. Ao confirmar os dados do parâmetros, o Umbrello volta às propriedades do método e, ao terminar de definir o método, é necessário confirmar para que o Umbrello volte à tela de definição da classe.

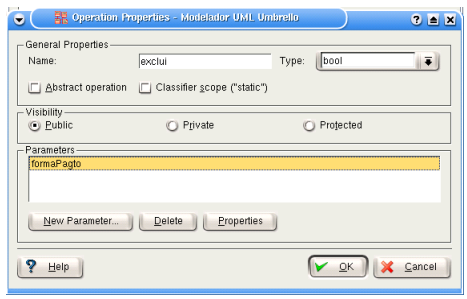

Figura 4-8. Umbrello - definindo um método

Assim como na criação de atributos, depois de confirmado um método, o Umbrello vai acrescentá-lo à lista e, para documentar o método criado, deve-se clicar sobre ele e preencher o texto *Documentation*. Da mesma forma que foi criada a classe `DListaFormaPagto` também serão criadas as outras classes do sistema. Nos diagramas de seqüência, a criação das classes é feita de forma semelhante à descrita.

Para ilustrar o funcionamento do Umbrello, tanto na modelagem como na geração de códigos fonte, foi elaborado apenas um trecho do diagrama de classes que mostra as classes criadas para manter o cadastro de formas de pagamento. Com a análise dos casos de uso várias outras clas-

ses deverão ser criadas, mas para o propósito deste trabalho, que é mostrar o funcionamento das ferramentas, o diagrama de classes da Figura 4-9 é suficiente para ilustrar como ficaria o resultado final do trabalho realizado com o auxílio do Umbrello.

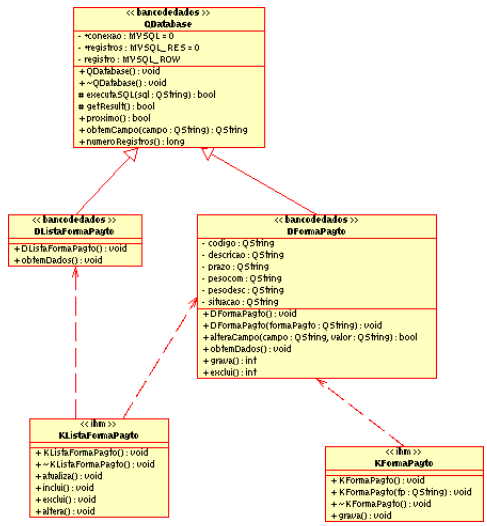

Figura 4-9. Trecho do diagrama de classes do sistema

4.3. Diagramas de seqüência

Sigunco [FURLAN], diagrama de seqüência mostra interações entre objetos organizada em uma seqüência de tempo e de mensagens trocadas, mas mão trata de associações entre os objetos. A definição das mensagens trocadas é feita baseando-se na documentação dos casos de uso.

Para elaborar o diagrama de seqüência, já se pode pensar na implementação. Algumas classes, não criadas no diagrama de classes preliminar, serão criadas agora para modelar a troca de mensagens no sistema, tanto do usuário com o sistema como mensagens trocadas entre as classes. Como

também estão sendo modeladas as mensagens trocadas com o usuário, no mínimo deverão ser criadas as classes de *interface*.

O ponto de partida para a elaboração dos diagramas de seqüência é a documentação dos casos de uso. Dessa forma para cada caso de uso existe pelo menos um diagrama de seqüência correspondente. Como são vários casos de uso no sistema, será mostrado a modelagem de apenas um diagrama de seqüência, os outros diagramas serão mostrados no Apêndice A.

No Capítulo 6 foi escolhido implementar o caso de uso `Manter cadastro de forma de pagamento` e por isto será mostrado como criar o diagrama de seqüência correspondente a este caso de uso. Para criar um diagrama de seqüência no Umbrello seleciona-se o menu `Diagram`⟶`New`⟶`Sequence Diagram`. A área de trabalho do Umbrello é limpa e fica pronta para iniciar a criação do diagrama de seqüência. A Figura 4-10 mostra o Umbrello trabalhando em um diagrama de seqüência.

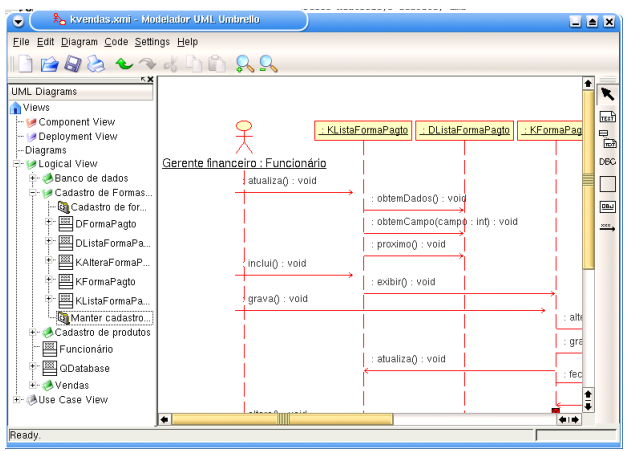

Figura 4-10. Umbrello - Diagrama de seqüência

Da mesma forma do diagrama de classes, para criar uma classe, deve-se clicar no ícone correspondente a classe na barra de ferramentas e clicar na área do diagrama. Para usar uma classe existente, arrasta-se a classe da árvore à esquerda da janela do Umbrello para a área do diagrama de seqüência. Para criar um ator, deve-se criar ou usar uma classe e dizer ao Umbrello que ela deve ser exibida no diagrama como um ator. Isto é feito dando um duplo clique na classe desejada e, na janela que se abre, marcar a opção *Draw as actor*.

A troca de mensagens entre as classes são modeladas selecionando o ícone correspondente a mensagem na barra de ferramentas à direita da janela do Umbrello e depois clicando nas linhas de vida das duas classes que vão trocar a mensagem. Depois da linha criada, deve-se dar um duplo clique sobre a mensagem para poder informar o seu rótulo em uma janela como a da Figura 4-11. Se a classe tiver algum método declarado, o Umbrello os apresenta para que seja escolhido qual deles implementa a mensagem. É possível ainda modelar uma mensagem diferente selecionando a opção *Custom operation* (operação personalizada).

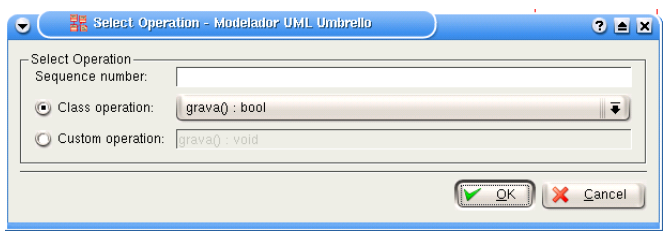

Figura 4-11. Umbrello - Associando a mensagem a um método da classe

Depois do diagrama de seqüência estar completo, ele pode ser exportado como uma figura para ser incorporado á documentação do sistema.

Este foi o procedimento usado na elaboração deste trabalho. A Figura 4-12 mostra o diagrama completo para o cadastro de formas de pagamento.

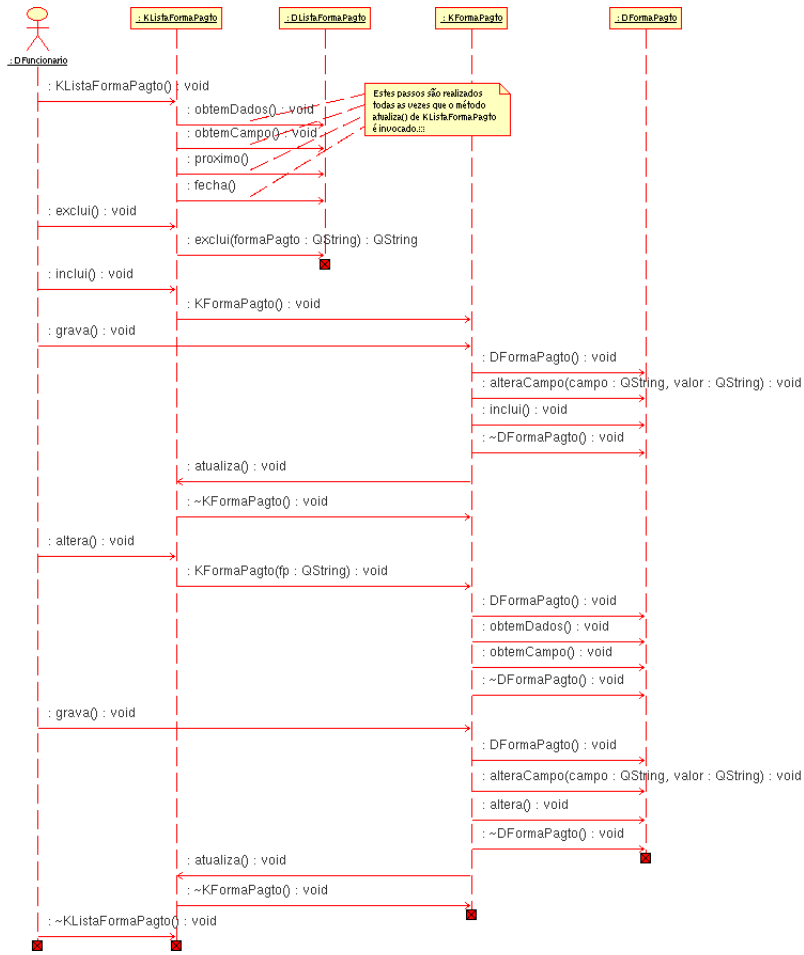

Figura 4-12. Diagrama de seqüência para o cadastro de formas de pagamento

Notas

1. A versão do Umbrello usada neste trabalho não estava traduzida para o português e por isso o texto do menu está em inglês. É possível que o Umbrello já tenha uma versão traduzida

Capítulo 5. Modelagem de dados

Neste capítulo, será realizada a modelagem de dados do sistema. Essa modelagem é que vai definir o o esquema do banco de dados, ou seja, as tabelas e respectivas estruturas. Primeiramente é apresentado o DER (Diagrama de Entidade Relacionamento) obtido segundo a especificação do sistema. Depois será mostrado como obtê-lo usando o TCM - *Toolkit for Conceptual Modeling.*

Depois de apresentado o DER, serã mostradas as estruturas das tabelas que deverão ser implementadas no banco de dados. Também será mostrado como criar as tabelas no banco de dados de forma rápida usando um script SQL.

5.1. Diagrama de Entidade-Relacionamento

O diagrama entidade-relacionamento expressa graficamente a estrutura global de um banco de dados. Segundo [SILBERSCHATZ]:

> "O modelo de dados entidade-relacionamento baseia-se na percepção de um universo contituído por um grupo básico de objetos chamados `entidades` e por `relacionamentos` entre esses objetos. Ele foi desenvolvido a fim de facilitar o projeto de bancos de dados permitindo a especificação de um `esquema de empreendimento`. Tal esquema representa a estrutura lógica global do banco de dados."

A partir da especificação do sistema, é que são definidas as entidades e como elas estão relacionadas. Para o sistema descrito no Capítulo 3, pode-se considerar válido o DER mostrado na Figura 5-1. É claro que o modelo apresentado não representa uma solução única, mas foi a que melhor se encaixou nos requisitos apresentados.

O diagrama da Figura 5-1 é bem simples, assim como deve ser simples um sistema para vendas. Uma particularidade no modelo que foge um pouco da especificação do sistema, é que não há nenhuma referência aos documentos fiscais exigidos (nota e cupom). A explicação é que cada empresa trabalha com um formulário personalizado e pelo fato de que os produtos podem ser regidos por legislações diferentes que alteram a forma

de calcular impostos. Devido a essa complexidade e para deixar o sistema mais genérico, decidiu-se por retirar essa funcionalidade do escopo deste trabalho.

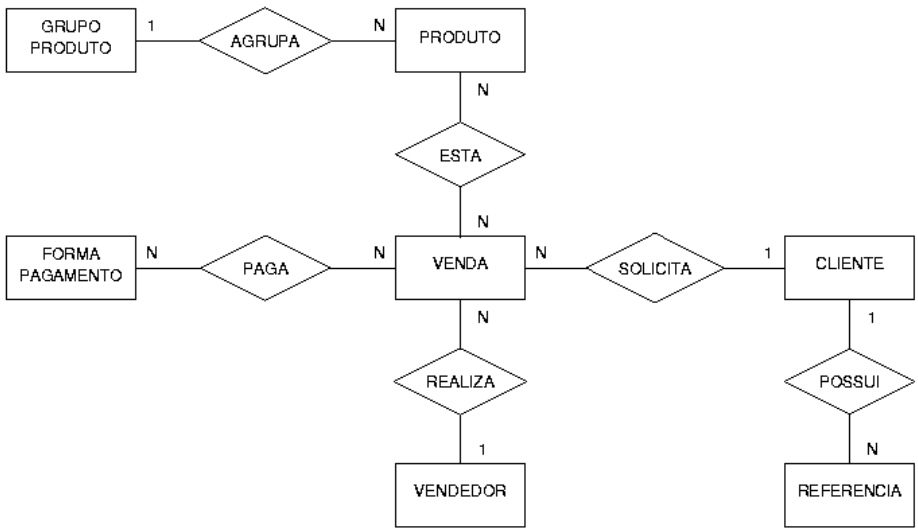

Figura 5-1. O diagrama entidade relacionamento

Como dito no Capítulo 2, foi escolhido o TCM para elaborar o modelo de dados e agora será mostrado como obtê-lo usando o TCM.

5.2. Gerando o DER com o TCM

O TCM - *Toolkit for Conceptual Modeling*, é uma ferramenta que fornece diversos editores gráficos para a modelagem de diferentes diagramas usados na especificação de sistemas. Esses diagramas representam a estrutura conceitual do sistema (daí o nome da ferramenta).

Para cada diagrama, o TCM oferece um editor específico. O TCM consegue gerar diagramas para a análise estruturada, análise orientada a objetos (diagramas UML) e também tem uma série de diagramas adicionais (genéricos). A tela principal do TCM é dividida em seções contendo os diagramas específicos para cada metodologia de desenvolvimento. Para elaborar o diagrama de Entidade Relacionamento, deve-se usar o editor *TESD (Entity Relationship Diagram).*

Selecionado o editor, uma janela como a da Figura 5-2 se abre para que o diagrama seja desenhado. A tela do TESD apresenta os elementos do diagrama e as ligações possíveis à esquerda da janela. São mostradas algumas informações sobre o documento na parte de baixo e à direita, ocupando a maior parte da tela, está a área de desenho.

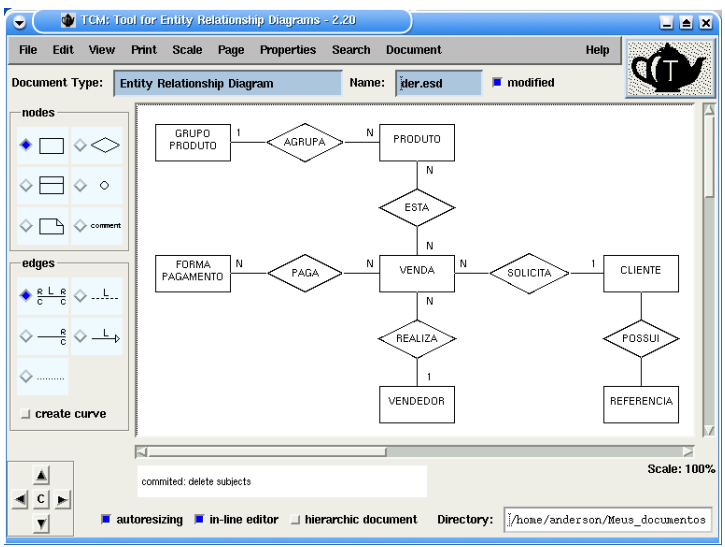

Figura 5-2. O Editor TESD - *Entity Relationship Diagram*

Para desenhar um elemento no diagrama, deve-se clicar sobre o íco-

ne correspondente ao elemento desejado e clicar na área de desenho. Para ligar dois elementos deve-se selecionar o tipo de ligação e, com o botão central do mouse[1], ligar os dois elementos. O TCM faz algumas validações nas ligações, evitando ligar elementos incompatíveis com o tipo de ligação selecionado.

É possível documentar cada elemento do diagrama como mostrado na Figura 5-3. Para isso deve-se clicar sobre o elemento desejado e selecionar o menu `Properties`⟶`Node/Edge Annotation`. Uma tela se abre permitindo fazer a documentação do elemento. Esse recurso pode ser usado, no caso do diagama entidade-relacionamento, para colocar a estrutura de cada elemento (atributos).

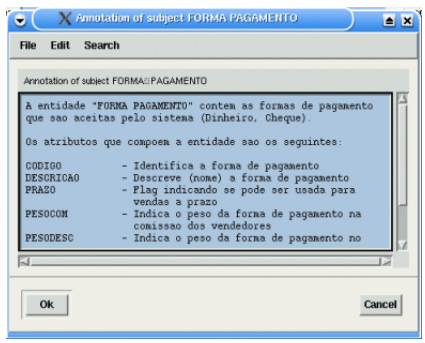

Figura 5-3. Documentação da entidade FORMA PAGAMENTO

Depois de desenhado o diagrama, ele pode ser impresso, ou exportado como uma figura para que ele possa fazer parte de um documento contendo a especificação do sistema (como neste trabalho). O TCM consegue exportar o diagrama para vários formatos, entre eles o PNG, *PostScript* (PS) e *Encapsulated PostScript* (EPS). Para converter o formato da imagem deve ser usado um editor gráfico como o Gimp (http://www.gimp.org/).

Para maiores informações sobre a utilização do TCM, recomenda-se a leitura de [TCMMANUAL].

5.3. Estutura do banco de dados

Depois de elaborado o DER, que mostra uma visão global do banco de dados, pode-se definir a estrutura do banco de dados. A grosso modo, deve-se criar uma tabela para cada entidade do diagrama, mas a estrutura final do banco de dados depende de algumas análises sobre o DER. Os relacionamentos *um para muitos*, por exemplo, são mapeados exportando-se a chave primária da entidade que está no lado *um* para a entidade do lado *muitos*. Nos relacionamentos *muitos para muitos* são criadas tabelas adicionais com as chaves de ambas as entidades que compõem o relacionamento bem como atributos adicionais requeridos pelo relacionamento.

Dessa forma foi definida a estrutura das tabelas do banco de dados que serão mostradas nas tabelas Tabela 5-1 a Tabela 5-9. Para cada tabela do banco de dados, é apresentada uma breve descrição e, para cada atributo, há uma explicação de sua função no sistema.

Tabela 5-1. GRUPOPRODUTO - Agrupa produtos semelhantes

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Identificação do grupo de produtos
DESCRICAO	Alfa-numérico	Descrição do grupo
COMISSAO	Porcentagem	Comissão paga nas vendas de peças do grupo
GRUPOPAI	Numérico	Código do grupo ao qual o grupo pertence (se for um subgrupo). Este atributo mapeia o relacionamento SUBGRUPO.

Tabela 5-2. PRODUTO - Cadastro de produtos

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Identifica o produto no sistema
GRUPO	Numérico	Grupo ao qual o produto pertence (chave de GRUPOPRODUTO)
DESCRICAO	Alfa-numérico	Descrição completa do produto
APELIDO	Alfa-numérico	Descrição resumida do produto
UNIDADE	Alfa-numérico	Identifica a unidade em que é vendida o produto
FRACAO	Sim/Não	Indica se pode vender quantidades fracionadas
DESCONTO	Porcentagem	Desconto máximo permitito ao produto
COMISSAO	Porcentagem	Comissão sobre o valor de venda
SITUACAO	Alfa-numérico	Indica se um produto está disponível para negociação (A=Ativo, I=Inativo)

Tabela 5-3. CLIENTE - Esta tabela contém os dados de cadastro do cliente

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Identifica o cliente no sistema

Campo	Tipo de dados	Descrição
NOME	Alfa-numérico	Nome do cliente (para jurídica representa a razão social)
APELIDO	Alfa-numérico	Apelido do cliente (para jurídica representa o nome fantasia)
ENDERECO	Alfa-numérico	Endereço completo da pessoa (rua, número, complemento)
BAIRRO	Alfa-numérico	Bairro
CIDADE	Alfa-numérico	Cidade
ESTADO	Alfa-numérico	Sigla do estado
CEP	Numérico	Número do CEP
TEL1	Numérico	Telefone completo (incluindo DDD)
TEL2	Numérico	Telefone completo (incluindo DDD)
FAX	Numérico	Fax (incluindo DDD)
EMAIL	Alfa-numérico	Caixa postal na Internet
CPF	Numérico	Número do CPF da pessoa (pessoa física)
IDENTIDADE	Numérico	Número da identidade
CNPJ	Numérico	Número CNPJ (pessoa jurídica)
INSCEST	Numérico	Número da inscrição estadual
DATACADASTRO	Data	Data de cadastro da pessoa

Campo	Tipo de dados	Descrição
TIPO	Alfa-numérico	Tipo de cliente (Oficina, Revenda, Consumidor, Transportadora)
CADASTRO	Numérico	Código do funcionário que cadastrou o cliente
VENDEDOR	Numérico	Código do vendedor que solicitou o cadastro
SITUACAO	Alfa-numérico	Situação de cadastro do cliente (Vista, Prazo, Bloqueado)

Tabela 5-4. REEFERENCIA - Referências do cliente

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Código do cliente (chave na tabela CLIENTE)
NUMERO	Numérico	Identifica a referência do cliente
NOME	Alfa-numérico	Nome da referência
TELEFONE	Alfa-numérico	Telefone da referência
CONTATO	Alfa-numérico	Pessoa que forneceu informação
DTULTIMACOMPRA	Data	Data da última compra
VLULTIMACOMPRA	Moeda	Valor da última compra
DTMAIORCOMPRA	Data	Data da maior compra
VLMAIORCOMPRA	Moeda	Valor da última compra

Campo	Tipo de dados	Descrição
ANOCADASTRO	Data	Ano de cadastro do cliente
CONCEITO	Alfa-numérico	Conceito dado ao cliente pela referência
OBSERVACAO	Alfa-numérico	Observações adicionais sobre o cliente

Tabela 5-5. USUARIO - Dados dos usuários do sistema

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Identificação do usuário
NOME	Alfa-numérico	Nome do usuário
VENDEDOR	Sim/Não	Indica se o usuário é um vendedor para permitir ou não vendas registradas em seu nome.
TELEFONE	Alfa-numérico	Telefone do usuário (incluindo DDD)
SENHA	Alfa-numérico	Senha de acesso ao sistema
SITUACAO	Alfa-numérico	Situação do usuário (Ativo,Inativo)

Tabela 5-6. FORMAPAGTO - Formas de pagamento aceitas na empresa

Campo	Tipo de dados	Descrição
CODIGO	Numérico	Identifica a forma de pagamento

Campo	Tipo de dados	Descrição
DESCRICAO	Alfa-numérico	Descrição da forma de pagamento
PRAZO	Sim/Não	Indica se pode usar esta forma em vendas à prazo
PESOCOM	Porcentagem	Indica o redutor aplicado sobre a comissão
PESODESC	Porcentagem	Indica o redutor aplicado sobre o desconto máximo
SITUACAO	Alfa-numérico	Situação da forma de paga-mento(Ativo,Inativo)

Tabela 5-7. VENDA - Registro das vendas (Orçamento, Ordem de venda e Venda)

Campo	Tipo de dados	Descrição
NUMERO	Numérico	Número da venda
DATA	Data	Data da venda
VENDEDOR	Numérico	Código do vendedor que realizou a venda
CLIENTE	Numérico	Código do cliente para o qual a venda foi realizada
TIPOPAGTO	Alfa-numérico	Tipo do pagamento (Vista, Prazo)
TIPODOCUMENTO	Alfa-numérico	Indica o tipo de documento gerado (Orçamento, Venda)

Campo	Tipo de dados	Descrição
INFORMACOES	Alfa-numérico	Informações complementares sobre a venda
VENDAFINAL	Numérico	Número da venda que agrupa esta e outras vendas para o mesmo cliente
USREGISTRO	Numérico	Usuário que registrou a venda
DTESTORNO	Data	Data de estorno da venda
USESTORNO	Numérico	Código do funcionário que fez o estorno davenda
MOTIVOESTORNO	Alfa-numérico	Motivo do estorno da venda

Tabela 5-8. ITEMVENDA - Relação de peças vendidas

Campo	Tipo de dados	Descrição
VENDA	Numérico	Número da venda (chave na tabela VENDA)
SEQUENCIA	Numérico	Seqüencial da peça na venda
PRODUTO	Numérico	Código do produto vendido
QUANTIDADE	Numérico	Quantidade vendida do produto
PRECOVENDA	Moeda	Preço negociado na venda
PRECOLISTA	Moeda	Preço de venda na lista

Tabela 5-9. PAGAMENTO - Desdobramento do pagamento da venda

Campo	Tipo de dados	Descrição
VENDA	Número	Número da venda à qual este pagamento se refere
SEQUENCIA	Número	Seqüencial da forma de pagamento para a venda
FORMAPAGTO	Número	Forma de pagamento utilizada
VENCIMENTO	Data	Data de vencimento da parcela
VALOR	Moeda	Valor da parcela

As tabelas mostraram apenas a estrutura do banco de dados, mas não informaram o que elas estão representando do DER. A Tabela 5-10 mostra a relação entre as tabelas do banco de dados e o DER, ou seja, ela diz qual elemento do DER gerou a tabela no banco de dados.

Tabela 5-10. Relação entre o DER e as tabelas do banco de dados

Tabela	Elemento do DER
GRUPOPRODUTO	Entidade GRUPOPRODUTO
PRODUTO	Entidade PRODUTO
CLIENTE	Entidade CLIENTE
REFERENCIA	Entidade REFERENCIA
USUARIO	Entidade VENDEDOR
FORMAPAGTO	Entidade FORMA PAGAMENTO
VENDA	Entidade VENDA
ITEMVENDA	Relacionamento ESTA entre as entidades VENDA e PRODUTO

Tabela	Elemento do DER
PAGAMENTO	Relacionamento PAGA entre VENDA e FORMA PAGAMENTO

5.4. Gerando as tabelas no banco de dados

Depois de especificadas as tabelas, pode-se partir para a criação dessas tabelas no banco de dados. É possível fazer isso com qualquer ferramenta de administração de banco de dados, mas ao mudar o SGBD, as tabelas terão que ser recriadas uma a uma.

Existem ferramentas que criam as tabelas no banco de dados baseado no DER diretamente, mas infelizmente o TCM não implementa esse recurso. Para amenizar a situação é possível aproveitar o fato de que os bancos de dados conhecem a linguagem SQL e criar um *script* para que a criação das tabelas seja feita de uma forma menos traumática.

Para este trabalho, o banco de dados escolhido foi o MySQL e na Figura 5-4, é mostrado um trecho de um *script* para montar as tabelas no MySQL. Para gerar o *script* para outro SGBD, deve-se fazer algumas pequenas alterações e ele vai continuar funcionando. O trecho mostrado na Figura 5-4 gera a tabela FORMAPAGTO no banco de dados.

```
-- ************************************************
-- Tabela FORMAPAGTO
-- Tabela contendo as formas de pagamento disponï¿½eis
-- para negociaï¿½o.
-- Restriï¿½es como nmero mï¿½imo de parcelas e prazos
-- mï¿½imos sï¿½ restriï¿½es de programa nï¿½ sendo
-- implementadas no banco de dados.
-- ************************************************
CREATE TABLE FORMAPAGTO (
  CODIGO TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  DESCRICAO VARCHAR(30) NOT NULL,
  PRAZO VARCHAR(1) NOT NULL DEFAULT 'S',
  PESOCOM DECIMAL(2,1) NOT NULL DEFAULT 1.0,
```

```
  PESODESC DECIMAL(2,1) NOT NULL DEFAULT 1.0,
  SITUACAO VARCHAR(1) NOT NULL DEFAULT 'A'
);
CREATE INDEX FPDESCRICAO ON FORMAPAGTO (DESCRICAO);
```

Figura 5-4. *Script* para criar a tabela FORMAPAGTO

Juntando as especificações de todas as tabelas a seren geradas, pode-se criar um *script* para ser enviado ao SGBD, criando todas as tabelas de uma só vez. Usando o sistema operacional Linux com o MySQL e supondo que o banco de dados `kvendas` tenha sido criado pelo administrador do banco de dados e que exista o *script* SQL `kvendas_db.sql`, contendo as instruções para criação das tabelas, deve-se o usar o comando da Figura 5-5 para enviar o *script* ao MySQL:

```
$ mysql kvendas < kvendas_db.sql
```

Figura 5-5. Enviando o script SQL ao MySQL

Notas

1. Caso o *mouse* tenha apenas dois botões, deve-se clicar no elemento usando os dois botões simultaneamente.

Capítulo 6. A implementação do sistema

Neste capï¿½ulo, é abordada a fase de implementação. Para a sua leitura, pressupõe-se que se tenha conhecimento básico de C++ e de programação orientada a objetos. Para mostrar a utilização das ferramentas, foi escolhido um caso de uso para ser implementado e, na descrição do processo, buscou-se colocar o máximo de detalhes possível sobre a implementação do caso de uso escolhido.

Para não correr o risco de ser cansativo, nem alongar demais o texto, foi escolhido um caso de uso simples. mas que contempla boa parte dos conceitos que devem ser utilizados na implementação do sistema. O caso de uso escolhido foi `Manter cadastro de formas de pagamento`.

O desenvolvimento desse caso de uso será mostrado desde o desenho da interface até o envio dos dados digitados pelo usuário ao banco de dados. Será visto como o KDevelop é integrado ao Designer e como fazer para incluir nos fontes a tela desenhada no Designer. A partir desse exemplo qualquer um dos outros casos de uso poderá ser implementado fazendo-se os devidos ajustes e, logicamente, usando-se *widgets*[1] apropriados para cada caso.

A criação das classes do sistema pode ser feita de duas formas: geração de código pelo Umbrello ou criação direta no KDevelop. Para demonstrar o potencial das ferramentas, algumas classes serão criadas pelo Umbrello e uma outra será criada diretamente no KDevelop. Assim, o desenvolvedor poderá escolher qual o modo que achar mais conveniente.

6.1. O que é necessário para implementar

Após ter modelado o sistema com a UML, o desenvolvedor tem em suas mãos as especificações das classes que deverão ser implementadas no sistema. Nesta seção será mostrada a implementação de um dos casos de uso modelados para mostrar a facilidade com que se pode implementar o sistema no Linux. As ferramentas utilizadas são o Designer para o desenho das telas do sistema e o KDevelop para manipular os códigos fonte.

O fato de dizer que a aplicação é escrita em C++ poderia causar

um certo pânico, pois ela possui a fama de ser difícil e que, por ser de baixo nível, nada está pronto para ser utilizado e assim, o programador deve implementar tudo para que o sistema funcione. Um dos objetivos de adotar o C++ neste trabalho é justamente mudar essa visão, pois o C++ pode ser tão simples de usar quanto linguagens de mais alto nível como o VisualBasic da Microsoft.

E o que é preciso saber de C++ para implementar o sistema? A resposta para essa pergunta parece ser complexa, mas os requisitos básicos para implementar um sistema em C++ usando as ferramentas que escolhidas se limitam ao mínimo necessário para implementar as classes criadas na modelagem. Rotinas de baixo nível para manipular janelas, ou acessar o banco de dados, nada disso precisa ser implementado. É preciso saber apenas como usar as bibliotecas que estão prontas para utilização.

Entre os pré-requisitos para implementar o sistema, estão os seguintes:

Orientação a objetos

Desde a modelagem do sistema, feita com auxílio da UML, os conceitos da orientação a objetos vêm sendo utilizados. Na implementação também não é diferente, pois as classes modeladas na fase de análise serão criadas agora na codificação do sistema.

O desenvolvedor deve estar familiarizado com os conceitos de herança e polimorfismo, pois eles são usados extensivamente na implementação da *interface* com o usuário e no acesso ao banco de dados.

Ponteiros

Talvez esse seja o conceito que mais assusta os programadores, pois a sua manipulação requer alguns cuidados, como alocar o ponteiro antes de utilizá-lo, e destruí-lo quando ele não for mais necessário. Depois de alguma prática, pode-se verificar que a coisa não é complicada e

que o cuidado é o mesmo necessário quando se abre ou fecha uma conexão ao banco de dados.

Algoritmo

Em linguagem mais popular: lógica de programação. Qual linguagem não necessita desse conhecimento?

Bibliotecas de desenvolvimento

Como fazer para conectar ao banco de dados? Como usar o Qt para desenhar as janelas? É preciso saber apenas usar as bibliotecas disponibilizadas pela maioria das distribuições Linux do mercado. Conectar ao banco de dados? `mysql_connect()`. Criar uma nova *interface*? Deve-se criar uma classe filha de `QWidget` ou similar. Criar uma linha de edição na janela? Acrescentar um `QLineEdit`. Nada complicado. O mais difícil é dizer para o compilador onde procurar as bibliotecas. Mas com o auxílio do KDevelop, nem esse trabalho é tão complicado assim.

Criatividade

A criatividade é um requisito importante no desenho da *interface* com o usuário para elaboração de janelas bem planejadas, com bom apelo visual, que não cansem o usuário e que sejam de fácil utilização. Também é muito importante a criatividade para conseguir resolver problemas de lógica, de linguagem. Mais uma vez: Qual linguagem não necessita desse conhecimento?

A lista apresentada contempla os principais pré-requisitos para implementar um sistema em Linux. O programador com experiência na linguagem C/C++ pode estranhar o fato de não ser citado a capacidade de trabalhar com *Makefiles*. O programador que tem conhecimento em programação em Linux vai perguntar sobre o *script* `configure` usado para gerar *Makefiles*. Para se ter desenvolvimento rápido, é preciso ocultar o máximo de detalhes básicos para que o programador se concentre na implementação apenas do que foi modelado. A tarefa de criar esses *scripts* de configuração é deixada para o KDevelop, que sabe o que é necessário para o sistema compilar. Bibliotecas adicionais, que não fazem parte do KDE,

podem ser informadas ao KDevelop e ele trata de juntá-las ao projeto.

6.2. Criando um projeto com o KDevelop

Independente do tamanho da aplicação, para usar o KDevelop, o primeiro passo é criar um projeto. Na criação do projeto é que o KDevelop colhe as informações necessárias para gerar os *scritps* de configuração responsáveis pela criação dos *Makefiles* que dizem ao compilador onde buscar as bibliotecas e quais são os pré-requsitos para gerar o programa final.

A Figura 6-1 mostra a tela principal do KDevelop configurada no modo IDEAI[2]. Depois do projeto criado, são acrescentadas abas nas laterais da janela, como `Classes` e `File Groups`.

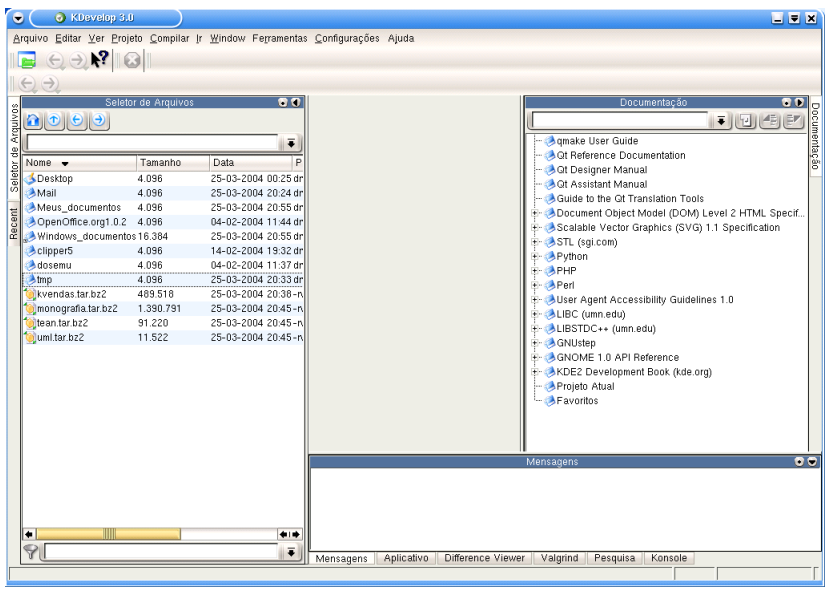

Figura 6-1. KDevelop - tela principal

Para criar um novo projeto no KDevelop, deve-se selecionar o menu `Projeto`⟶`Novo projeto` (*New Project* se o KDevelop estiver em inglês). O KDevelop então inicia o assistente de criação de projeto, como mostrado na Figura 6-2. É interessante notar que vários tipos de projetos são suportados pelo KDevelop, incluindo *scripts* php. No caso do projeto deste trabalho, deve ser celecionada a opção `C++`⟶`KDE`⟶`Simple KDE Application.` Na primeira tela do assistente, deve ser informado o nome do projeto e alguns dados adicionais para identificação do autor e do projeto. Clicando no botão `Próximo`, outras telas do assistente serão abertas e, para o projeto descrito, deve-se aceitar as opções padrão. O KDevelop vai criar um diretório para projeto e colocar nele todos os arquivos necessários para gerar o sistema.

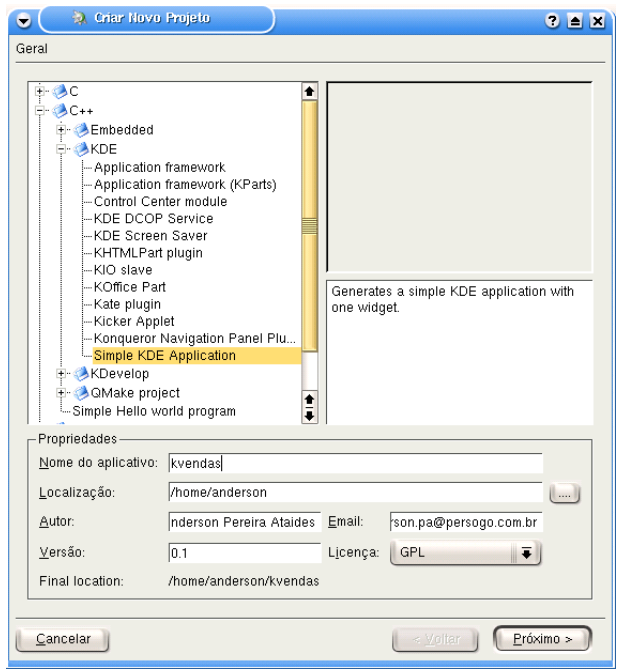

Figura 6-2. KDevelop - assistente de criação de projeto

O novo projeto já faz alguma coisa: mostra uma janela com os dizeres *Hello World*. Para testar seu funcionamento, deve-se mandar o KDevelop rodar o programa (`Compilar`⟶`Executar programa`). O KDevelop vai detectar que o projeto é novo e pergunta se você quer executar o *script* `configure`, ou seja, ao criar o projeto, o KDevelop se encarregou de criar os *scripts* de configuração. Aceitando a execução do *script*, o sistema vai compilar e logo após será executado (a janela do aplicativo será exibida na tela).

O projeto recém criado, tem todas as ligações necessárias para compilar usando as bibliotecas Qt e KDE, mas falta uma que é de fundamental importância ao projeto: a biblioteca do MySQL. No momento em que for preciso usar uma função do MySQL, será necessário informar duas coisas: os *headers* onde estão declarados os protótipos das funções e estruturas do MySQL, e a biblioteca para a geração do executável do sistema.

O arquivo *header* que deve ser usado é o `mysql.h`. Geralmente, as distribuições Linux colocam esses arquivos no diretório `/usr/include`. No caso específico do Conectiva Linux, os *headers* do MySQL são colocados em `/usr/include/mysql`, porém ele vem configurado para procurar os *headers* apenas no diretório `/usr/include`. Para contornar esse problema, deve-se informar o subdiretório onde está o MySQL, e a linha com o *include* ficará como mostrado na 4Figura 6-3.

```
#include <mysql/mysql.h>
```

Figura 6-3. Acrescentando um `include` em um arquivo fonte

Curiosamente, no Conectiva Linux, a biblioteca `libmysqlclient` está localizada no diretório padrão `/usr/lib` não sendo necessário, como no caso do *header*, informar o subdiretório onde o arquivo se localiza. Para informar ao KDevlop que ele precisa usar a biblioteca `libmysqlclient.so`, deve-se abrir a janela `Automake Manager`, mostrada na Figura 6-4.

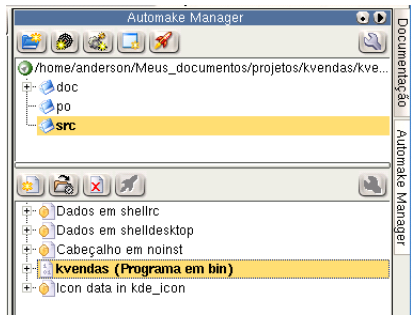

Figura 6-4. KDevelop - a janela *Automake Manager*

Com a janela *Automake Manager* aberta, na parte inferior, deve-se cicar no botão *Show Options*, ou ainda clicar com o botão direito sobre o alvo[3] `kvendas` e selecionar a opção *Options*. Uma caixa de diálogo se abre para que sejam informadas as opções do alvo selecionado. Para informar uma biblioteca, deve-se selecionar a guia `Bibliotecas` e, como a biblioteca do MySQL é externa ao projeto, deve ser usada a parte inferior (`Link libraries outside project (LDADD)`). Agora deve-se dar um clique no botão `Adicionar`. Uma caixa de diálogo como a da Figura 6-5 se abre para que seja informada a biblioteca que se deseja incluir. Nessa caixa, deve aparecer o texto `-l` e, sem apagar o que está escrito, deve ser acrescentado `mysqlclient`. Ao fechar a caixa de diálogo, a biblioteca do MySQL vai aparecer na lista de bibliotecas utilizadas pelo sistema.

O projeto está agora pronto para implementação. Para ver como é possível integrar o Umbrello com o KDevelop, será mostrado como gerar os arquivos fontes das classes no Umbrello e depois como incluir esses arquivos no projeto criado pelo KDevelop. Por enquanto, o KDevelop é deixado de lado para que as classes sejam geradas pelo Umbrello.

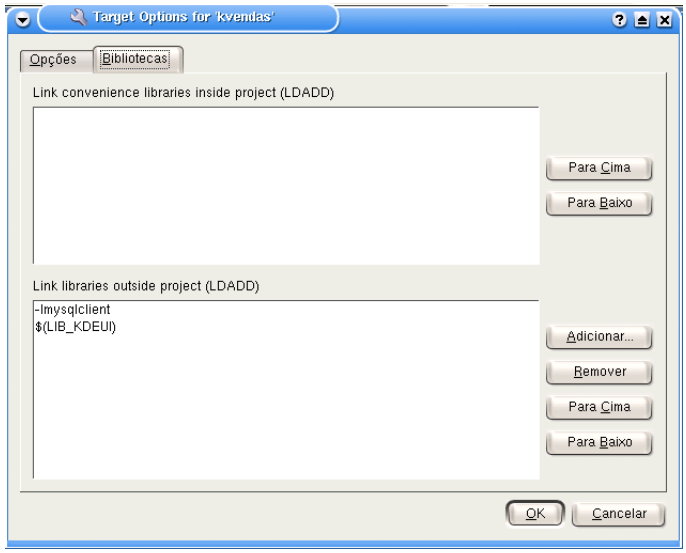

Figura 6-5. KDevelop - informando bibliotecas adicionais

6.3. Gerando códigos fonte com o Umbrello

Voltar a falar de uma ferramenta de modelagem estando na fase de implementação, por mais estranho que possa parecer, não é algo fora de contexto. Principalmente por usar a orientação a objetos, a implementação não deve ser um processo independente da modelagem, pois as classes que serão implementadas foram concebidas na modelagem. Mas o motivo maior de se voltar a falar sobre o Umbrello, é para mostrar como ele cria códigos fonte baseado nas classes que foram modeladas.

Os códigos fonte não foram gerados antes, porque para gerá-los, é necessário saber onde eles serão armazenados. Somente agora, depois do projeto criado no KDevelop, é conhecido o local onde os arquivos vão estar localizados. Dessa forma, pode-se voltar ao Umbrello e fazê-lo gerar os arquivos contendo o código fonte das classes baseado no modelo.

De acordo com o diagrama de seqüência mostrado no Capítulo 4,

as classes necessárias para implementar o cadastro de formas de pagamento são `KListaFormaPagto`, `DListaFormaPagto`, `KFormaPagto` e `DFormaPagto`. Mas de acordo com o diagrama de classes mostrado no mesmo capítulo, pode-se notar que as classes `DListaFormaPagto` e `DFormaPagto` são filhas de `QDatabase`. Assim, para implementar o cadastro de formas de pagamento, as cinco classes devem ser criadas no sistema.

Para ilustrar o funcionamento das duas ferramentas (Umbrello e KDevelop), as classes `QDatabase`, `DListaFormaPagto`, `DFormaPagto` e `KFormaPagto` serã geradas pelo Umbrello . A classe `KListaFormaPagto` será criada diretamente no KDevelop.

O procedimento para gerar os códigos fonte com o Umbrello inicia-se selecionando o menu `Code`⟶`Code Generation Wizard`. Uma janela como a mostrada pela Figura 6-6 é aberta mostrando todas as classes modeladas. Como padrão, o Umbrello sugere a geração do código de todas as classes, mas como apenas algumas devem ser implementadas, devem ser mantidas na lista *Classes Selected* apenas as classes desejadas.

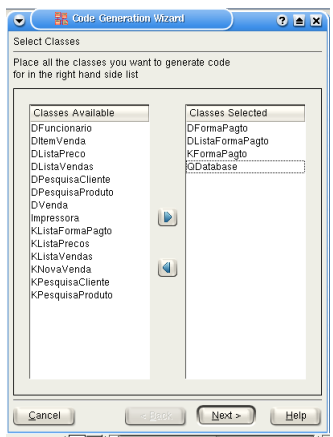

Figura 6-6. Umbrello - seleção das classes para codificação

Escolhidas as classes, deve-se clicar no botão *Next* para o assistente passar para a próxima página. Na segunda página, mostrada pela Figura 6-7, as informações mais importantes que serão alteradas são o diretório e a linguagem. Na linha de texto do diretório, deve ser colocado o caminho onde os fontes serão armazenados, ou seja, no diretório onde se localiza o projeto criado no KDevelop. Como padrão, o KDevelop coloca os fontes dentro do subdiretório `src` e é justamente o caminho que deve ser informado ao Umbrello para que ele gere os fontes.

Na parte inferior da janela do assistente, a linguagem selecionada deve ser `Cpp`. Ao abrir a lista, pode-se notar que o Umbrello consegue trabalhar com várias linguagens de programação. Informado o diretório e a linguagem, deve-se clicar em *Next*.

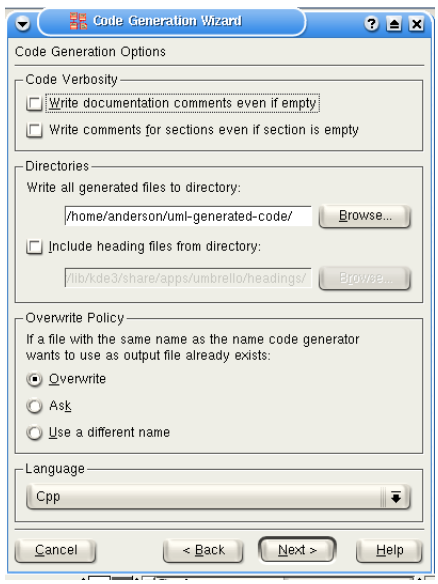

Figura 6-7. Umbrello - definindo as opções de criação dos fontes

A próxima página do assistente, apenas mostra as classes que serão geradas. Para gerar os fontes, deve-se clicar no botão *Generate*. Depois de concluída a geração dos arquivos, o boão muda para *Finish* e então deve-se clicar nele para fechar o assistente.

Agora falta fazer com que o KDevelop acrescente os arquivos gerados ao projeto. Para isso, deve-se usar o *Automake Manager*. Na parte inferior, deve-se clicar no botão *Add Existing Files* ou clicar sobre o alvo com o botão direito e selecionar a opção com o mesmo nome do botão. Uma janela se abre para que sejam selecionados os arquivos que devem ser incluídos no projeto. Note que exstem na lista alguns arquivos que têm o mesmo nome das classes criadas no Umbrello, porém todo escrito em minúsculo e com duas extensões: `.cpp` e `.h`. Todos eles devem ser selecionados[4], pois representam a implementação das classes.

O KDevelop a partir de agora conhece as classes criadas pelo Umbrello e o programador pode codificar os métodos, bem como efetuar alterações específicas para a implementação, como adição de instruções `include` por exemplo.

É importante lembrar que depois de alterado pelo KDevelop, os arquivos não podem ser gerados novamente pelo Umbrello, pois ele vai sobrescrever as alterações efetuadas. Portanto, qualquer alteração realizada no Umbrello, deve ser feita manualmente no KDevelop.

6.4. Criando os formulários do caso de uso

Para elaborar os formulários do sistema, não será usado o KDevelop, pois ele não fornece uma ferramenta para desenho. É possível, com o Kevelop, gerar os formulários (como o `KVendas` criado pelo KDevelop), mas será via código fonte, o que torna o desenvolvimento mais lento. Ao invés de trabalhar nos fontes, é mais fácil e intuitivo elaborar o desenho da *interface* visualmente usando o Designer. Depois de desenhados os formulários, o KDevelop os incorpora ao projeto restando apenas a tarefa de fazer as devidas chamadas para que ele seja exibido na tela.

Conforme a modelagem do sistema, duas classes fazem a *interface* com o usuário no cadastro de formas de pagamento: `KListaFormaPagto` e `KFormaPagto`. `KListaFormaPagto` é a primeira classe com a qual o usuário vai interagir, e através dela, será invocada a outra classe. Mas se os formulários não serão gerados no KDevelop, porque as classes foram criadas? Na verdade, `KListaFormaPagto` e `KFormaPagto` serão sub-classes de classes geradas a partir dos formulários desenhados no Designer. Posteriormente esse processo será explicado em mais detalhes. Portanto, para o cadatro de formas de pagamento, será preciso criar dois formulários.

Para criar um formulário, deve ser selecionado o menu `Arquivo`⟶`Novo` ou clicar no botão *New file* da barra de ferramentas. Uma caixa de diálogo como a da Figura 6-8 se abre para que seja escolhido o tipo de arquivo que será adicionado ao projeto. O campo `Diretório` deve estar preenchido com o camninho completo do diretório onde o projeto foi criado. Sem apagar o que está escrito, acrescenta-se `src`, que é o subdiretório do projeto onde os arquivos com os fontes estão armazenados.

No campo *Filename*, será informado o nome de `qlistaformapagto.ui`[5] e posteriormente será explicado porque não foi dado o nome de `klistaformapagto` (o nome da classe modelada). Na lista que tem abaixo, será usada a opção `Widget (.ui)` e depois deve-se clicar em `Ok`. Se o KDevelop perguntar sobre o *target*, é só aceitar as sugestões que o arquivo será acrescentado ao *target* atual.

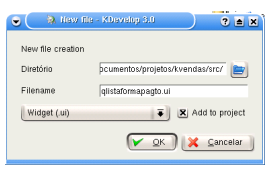

Figura 6-8. KDevelop - criando um novo arquivo

Nesse momento o Designer deve ser aberto[6] e se ele perguntar novamente o tipo de *interface* que será criada, deve ser selecionado `Widget`. Um formulário m branco é aberto para que seja desenhada a nova *interface*. À esquerda da tela do Designer está a janela *Toolbox* onde são selecionados os *widgets* que serão acrescentados ao formulário. À direita estão outras janelas das quais a principal é a *Property Editor/Signal Handlers*, onde são definidas as propriedades dos *widgets*. A figura Figura 6-9 mostra a tela principal do Designer.

Como o aplicativo está sendo desenvolvido para funcionar com o KDE, se estiverem disponíveis, os *widgets* específicos para o KDE devem ser usados.

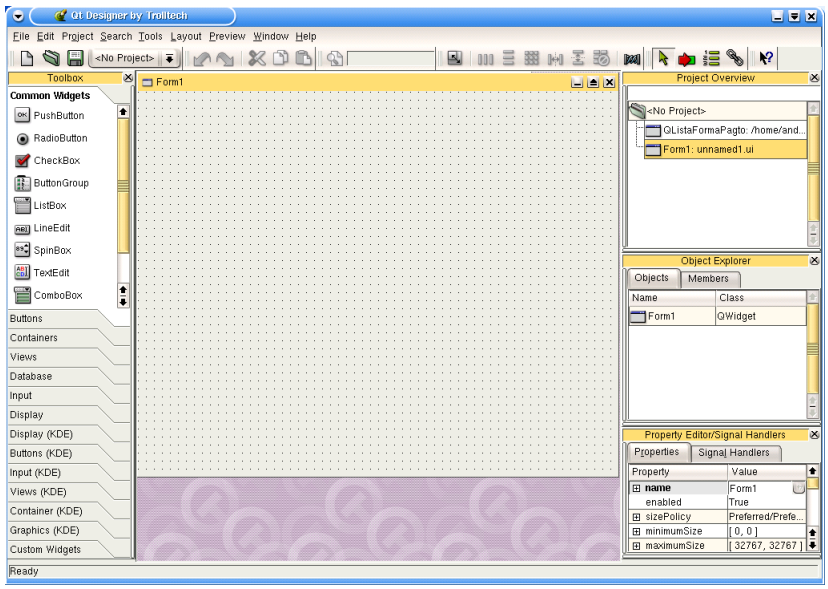

Figura 6-9. Designer - tela principal

6.4.1. Desenhando a tela de cadastro de formas de pagamento

A tela de cadastro de formas de pagamento deve apresentar uma lista, onde serão exibidas as formas de pagamento cadastradas no banco de dados, e alguns botões que permitem ao usuário acrescentar, excluir ou alterar formas de pagamento. Para mostrar a lista de formas de pagamento, será usado o *widget* `KListView`. Para criá-lo, deve-se selecionar a guia `Views (KDE)` e então o elemento `KListView`. Depois deve-se clicar no meio do formulário em branco. Não é preciso se preocupar com o *layout* da janela agora, mas apenas acrescentar os seus elementos. O Qt dispõe de *widgets* especializados em *layout* e a organização da janela será deixada a cargo desses elementos.

Agora são acrescentados os botões que vão fazer o formulário responder aos comandos do usuário. Para criar os botões, seleciona-se a guia `Buttons (KDE)` na lista de *widgets* e e acrescenta-se o *widget* `KPushButton`. Deverão ser acrescentados ao formulário cinco botões um abaixo do outro à direita do formulário. Esses cinco botões serão: Adicionar, Modificar, Apagar, Atualizar e Fechar.

6.4.2. Organizando os *widgets* no formulário

Depois de todos os *widgets* criados no formulário, deve-se então organizá-los na tela. Nesse ponto, o projetista da *interface* pode ser tentado a organizar o formulário manualmente, mas dessa forma não é possível garantir que a forma da janela se mantenha caso o usuário decida redimensionar a janela. Ao invés de organizar os *widgets* manualmente, serão usados *widgets* próprios para organizar a janela automaticamente. A primeira tarefa é acrescentar dois espaçadores verticais, um acima e outro abaixo do quarto botão (na ordem informada para criação dos botões, o quarto botão corresponde a Atualizar). Os dois espaçadores criados vão "separar" o botão dos outros. A figura Figura 6-10 mostra como ficaria o formulário com os *widgets* criados.

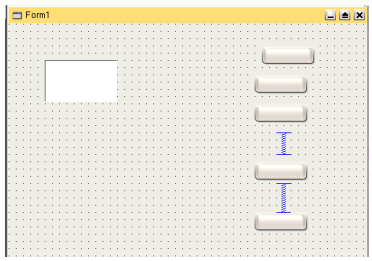

Figura 6-10. Designer - formulário faltando layout

Depois de acrescentados os espaçadores, todos os botões juntamente com os espaçadores devem ser selecionados e em seguida deve usado o menu `Layout`⟶`Lay Out Vertically`. Depois de informado o *layout*, os botões ficam organizados. O próximo passo é clicar na área vazia do formulário e selecionar o menu `Layout`⟶`Lsy Out Horizontally`. Agora todo o formulário estará bem organizado. Se o formulário for redimensionado, a organização dos *widgets* se mantém.

Para ter uma idéia de como o formulário vai ficar no sistema pronto, deve-se abrir o menu `Preview` e selecionar um dos formatos da lista. Outra opção para previsualizar o formulário é pressionar `Control+T`. A Figura 6-11 mostra essa previsualização.

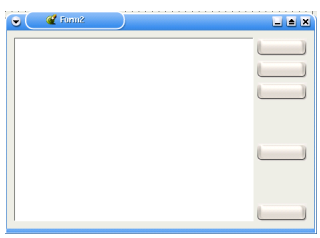

Figura 6-11. Designer - previsualização no formao Keramik

6.4.3. Alterando as propriedades dos elementos

Agora que o formulário está organizado, os *widgets* devem ser alterados para que tenham as características desejadas para o sistema. O primeiro passo será alterar o texto dos botões. Para isto, deve-se dar um clique duplo sobre o botão. Uma caixa de diálogo se abre para que seja digitado o texto que vai aparecer no botão. Para que o botão tenha uma tecla de acesso rápido (aquela que aparece sublinhada e que o usuário pressiona Alt+letra para acessá-lo) coloca-se o símbolo "&" antes da letra que deverá ser usada para acesso rápido. Dessa forma o botão `Adicionar`, por exemplo, terá o texto "&Adicionar".

Em seguida serão definidas as colunas que serão exibidas na lista. No Designer é possível deixar pronto o nome e a ordem em que os campos serão exibidos na lista. Para isso, deve-se dar um clique duplo sobre o *widget* criado para representar a lista. Uma caixa de diálogo como a da Figura 6-12 é aberta para que sejam alteradas algumas propriedades da lista. Agora deve ser selecionada a aba *Columns* para que as propriedades das colunas sejam alteradas. Para criar novas colunas, deve-se clicar em *New Column* e digitar o seu nome na linha de texto *Text*.

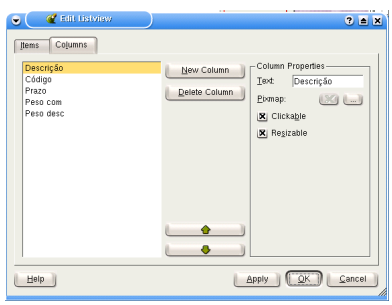

Figura 6-12. Designer - Alterando as propriedades das colunas

Existem duas opções abaixo do nome da coluna, onde a primeira (*clickable*), se for selecionada vai permitir ao usuário clicar no nome da coluna e o `ListView` vai ordenar a lista por esta coluna. Essa opção deve estar marcada apenas para os campos `Código` e `Descrição`. A segunda opção (*Resizable*) permite ao usuário redimensionar a coluna de acordo com sua vontade. Definidas as propriedades das colunas, deve-se clicar `Ok` e o `KListView` deverá ser redesenhado de acordo com as modificações realizadas nas colunas.

Até agora, foi feito apenas o desenho da tela, não houve nenhuma preocupação de como os *widgets* serão criados ou usados na implementação do sistema. A partir de agora serão trabalhadas as propriedades[7] dos *widgets* para que eles funcionem de acordo com as necessidades do sistema modelado e que sejam fáceis de identificar na hora de trabalhar com os códigos fonte.

O primeiro *widget* que será alterado será o próprio formulário. Para trabalhar nas propriedades do formulário, nenhum outro *widget* pode estar selecionado. A primeira propriedade listada na barra de propriedades é o nome do formulário. O nome mostrado deve ser `Form1`, que será alterado para `QListaFormaPagto`. Outra propriedade que deve ser alterada é *caption* que é o título da janela. Esse valor (que deve estar `Form1`) deve ser alterado para "Cadastro de formas de pagamento".

Alteradas as propriedades do formulário, serão agora alteradas as propriedades do `KListView`. O nome do *widget* será alterado para `listaFormaPagto` e a propriedade `allColumnsShowFocus` para `true`. Esta última propriedade faz com que, ao clicar um elemento na lista, a linha inteira seja selecionada. A última propriedade que será alterada para `listaFormaPagto` (note que agora a lista tem um nome conhecido, e ele pode ser udaso para referenciar o *widget*) é `whatsThis`. Esta propriedade faz com que a função *What's this* (O que é isto) seja ativada. Logo à direita da linha de texto existe um botão com reticências (...). Ao clicar sobre ele, um editor de texto como o da Figura 6-13 é aberto para que seja digitado o texto de ajuda para o *widget* selecionado, no caso `listaFormaPagto`.

Acima da janela do editor, na barra de ferramentas, existem alguns botões de formatação. Ao clicar em qualquer um deles serão adicionadas ao texto *tags* HTML correspondente à formatação escolhida. Esse recurso de suportar HTML é muito interessante, pois o texto pode ser formatado como se estivsse sendo montada uma página web.

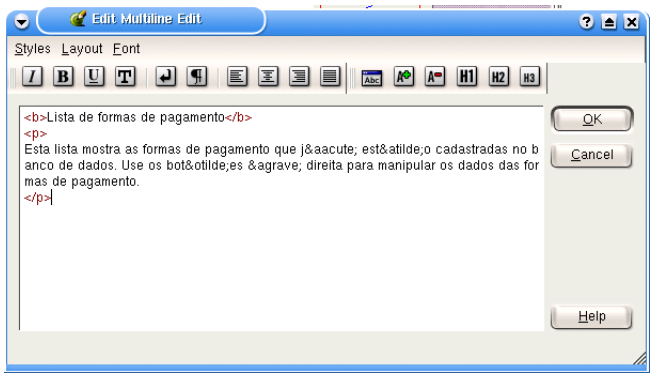

Figura 6-13. Designer - Diálogo para digitar o texto *What's this*

Para os botões, serão alteradas apenas duas propriedades: `name` e `toolTip`. A propriedade `toolTip` faz com que apareça um pequeno texto quando o mouse estaciona sobre o *widget*. As propriedades para cada botão serão alteradas de acordo com a Tabela 6-1.

Tabela 6-1. Propriedades dos botões

Botão	***name***	***toolTip***
Adicionar	botaoAdicionar	Cria uma nova forma de pagamento.

Botão	***name***	***toolTip***
Modificar	botaoModificar	Altera os dados da forma de pagamento selecionada
Apagar	botaoApagar	Apaga uma forma de pagamento
Atualizar	botaoAtualizar	Atualiza a lista
Fechar	botaoFechar	Fecha esta janela

Agora o formulário está pronto para responder a eventos resultantes da interação com o usuário. A próxima etapa será justamente informar quais os eventos serão tratados pelo formulário.

6.4.4. Interação com o usuário

Depois de pronto o formulário, com *layout* definido e com todos os *widgets* nomeados, deve-se pensar nos eventos que deverão ser tratados pelo formulário. Alguns eventos como minimizar, maximizar, redimensionar deverão ser tratados pela própria biblioteca Qt, a não ser que seja necessário algum tratamento especial. Porém, ao clicar no botão `Adicionar` será necessário ter um código diferenciado para que seja acrescentada uma forma de pagamento à lista. A codificação do evento propriamente, será feita no KDevelop, mas no Designer é possível dizer pelo menos o caminho a tomar quando ocorrer algum evento.

O Qt trabalha com um mecanismo para tratamento de eventos que ele denomina de *signal and slot*, onde um *widget*, ao receber um evento, emite um sinal que pode ser interceptado por outros *widgets* através de um *slot*. Tanto o *signal* como o *slot* são métodos especiais implementados na classe que representa o *widget*. Na verdade, este mecanismo não serve apenas para tratamento de eventos, mas para fazer uma troca de mensagens entre classes sem que uma conheça a estrutura da outra.

Para fazer com que um *widget* responda a um sinal emitido por outro, deve ser implementado um *slot* na classe e, ao instanciar a classe, o *signal* que deve ser capturado deve ser conectado ao *slot* criado na clas-

se. É assim que a classe `QListaFormaPagto` responderá aos cliques nos botões acrescentados na *interface*. Para maiores informações sobre o mecanismo *signal and slot*, recomenda-se a leitura de [QTMANUAL]. A partir de agora o Designer será instruído a fazer essas conexões.

Para iníciar, será informado como o formulário irá responder ao clique do botão `Fechar`. Obviamente, ao clicar no botão fechar, deseja-se que a janela se feche e a escolha por iniciar com este botão foi para mostrar um fato: existem alguns *slots* implementados internamente pelo Qt para os *widgets*. Para fazer a primeira conexão (clicar em Fechar = Fechar a janela) deve-se selecionar o menu `Edit`⟶`Connections`. Uma caixa de diálogo como a da Figura 6-14 é aberta mostrando uma lista com as colunas mostradas na Tabela 6-2.

Tabela 6-2. Conexsão *signal/slot*

Coluna	**Descrição**
Sender	Indica o *widget* que emite o sinal.
Signal	Qual o sinal deverá ser capturado.
Receiver	Indica qual o *widget* que vai receber o sinal
Slot	Qual o slot será processado quando o sinal for capturado

Manipulando a lista de conexões, pode-se perceber que os mesmos sinais podem ser conectados a tantos *slots* quanto forem necessários. Dessa forma, se houverem vários widgets dependentes de um outro, esse mecanismo pode ser usado para atualizar os diversos *widgets* dependentes. Primeiramente será feita a conexão do sinal `clicked()` do *widget* `botaoFechar` com o *slot* `close()` do *widget* `QListaFormaPagto`. Resumindo, está sendo informado que, quando o botão `Fechar` for pressionado, o método `close()` do formulário (representado pela classe `QListaFormaPagto`) deverá ser executado.

Para fazer a conexão, clique no botão *New*. Na coluna *Sender* vai ter uma lista com todos os *widgets* disponíveis no formulário (daí a im-

portância de dar nome a todos os *widgets*). Na coluna *Sender* escolhe-se o *widget* `botaoFechar`; na coluna *Signal* é escolhido o sinal `clicked()`; na coluna *Receiver* escolhe-se o *widget* `QListaFormaPagto` e na coluna *Slot*, seleciona-se o *slot* `close()`. Se o formulário for visualizado agora o botão Fechar vai funcionar corretamente, porque ele está conectado a um *slot* implementado pelo Qt.

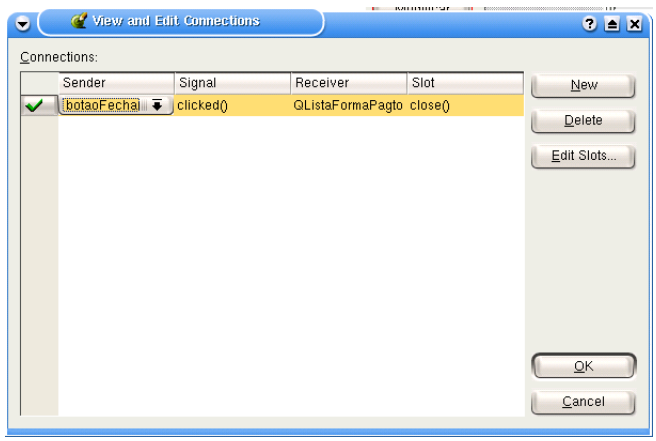

Figura 6-14. Designer - conectando *signal* e *slot*

Para os outros botões, deverá haver um procedimento personalizado, pois o Qt não sabe o que fazer para manipular formas de pagamento. O Designer pode deixar definidos os *slots* que deverão ser usados ao clicar nos botões do formulário. Agora será mostrado em detalhes a conexão do botão Adicionar. Na coluna *Sender* escolhe-se `botaoAdicionar`; na coluna *Signal* escolhe-se `clicked()`; na coluna *Receiver* a opção `QListaFormaPagto`. Agora vem a personalização: criar um novo *slot*.

A criação de um novo *slot* é uma tarefa muito simples. Na janela de conexões, deve-se clicar no botão *Edit Slots*. Uma segunda caixa de

diálogo como a da Figura 6-15 aparece com uma lista que deverá conter os *slots* personalizados. Para criar um novo *slot*, deve-se clicar no botão *New Function*. Nesse momento, os campos abaixo da tela serão disponibilizados para que sejam digitados os dados do novo *slot*. No campo *Function* deve ser digitado `adicionar` e os outros campos, são deixados como sugerido. Depois clica-se em *Ok*. Ao voltar, o slot criado vai aparecer na lista e então ele deve ser selecionado para confirmar a conexão.

Da mesma forma, pode-se criar os *slots* para os outros botões. Deve-se criar então o *slot* `modificar()` para associar ao botão `botaoModificar`, o *slot* `apagar()` para associar a `botaoApagar` e o *slot* `atualizar()` para associar a `botaoAtualizar`.

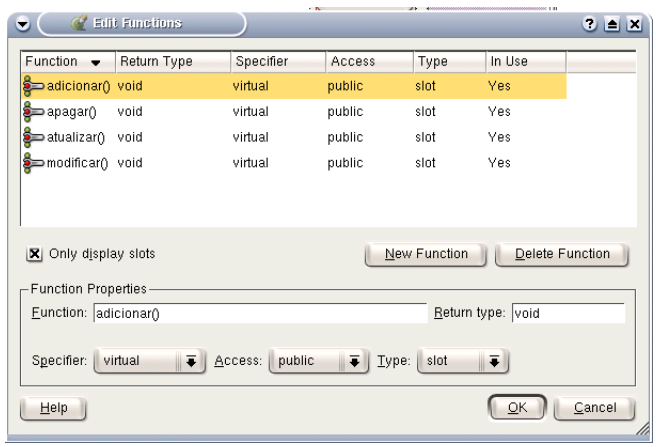

Figura 6-15. Designer - incluindo novos *slots*

Depois de criados os *slots* pode-se perceber uma particularidade: os seus nomes correspondem às mensagens trocadas pela classe `KListaFormaPagto` no diagrama de seqüência `Manter cadastro de formas de pagamento`. Isto porque obviamente deve haver coerência

entre o modelo e a implementação. Depois de criados todos os *slots* deve-se fazer a conexão com os respectivos *widgets*.

Finalmente, o formulário `QListaFormaPagto` está pronto para a codificação, ou seja, a interface está pronta, faltando apenas implementar os *slots* que foram criados.

6.5. O formulário QFormaPagto

Depois de mostrar em detalhes a criação do formulário, `QListaFormaPagto`, será criado o segundo, que manipula os dados da forma de pagamento. Mais uma vez, no KDevelop, deve-se criar um novo arquivo. O nome do arquivo será `qformapagto.ui` e o tipo do arquivo será `Dialog (.ui)`. O tipo *dialog* se justifica porque este formulário será mostrado como uma caixa de diálogo para que sejam digitados os dados da forma de pagamento.

O formulário ficará na forma mostrada na Figura 6-16. A seqüência para sua criação é semelhante à do formulário `QListaFormaPagto`. Primeiro, cria-se os *widgets*, depois o o formulário é organizado, as propriedades são alteradas e finalmente são definidos os *slots* que respondem aos eventos.

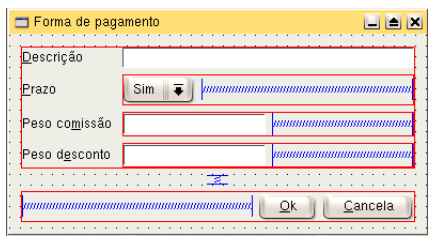

Figura 6-16. Designer - O formulário `QFormaPagto`

Os nomes dos *widgets* estão obedecendo ao padrão `widget+Nome`. Assim o primeiro label vai se chamar `labelDescricao`, a primeira linha de edição `lineDescricao` e assim sucessivamente. Merece destaque o segundo campo: um `KComboBox`. Para colocar as opções "Sim" e "Não", deve-se dar um duplo clique sobre o *widget* e uma caixa de diálogo como a da Figura 6-17, semelhante a usada para definir as colunas do `KListView`, será exibida. Nessa caixa, para acrescentar um novo item, clica-se em *New Item* e digitar o texto no campo *Text*.

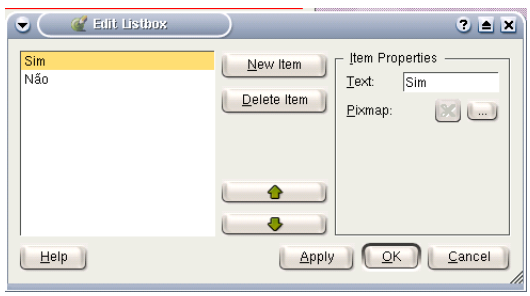

Figura 6-17. Designer - Definindo as opções de um `KComboBox`

Para chegar á forma final primeiro são selecionados os *widgets* que representam os campos junto com os espaçadores à sua frente e aplica-se o *layout* horizontal (este passo foi feito um a um). Depois seleciona-se os campos junto com os rótulos e aplica-se o *layout* em tabela (*on a grid*). Em seguida, são selecionados os botões com o espaçador a sua esquerda e em seguida aplica-se o *layout* horizontal. Finalmente, seleciona-se o formulário e aplica-se o *layout* vertical.

Os textos dos *widgets* são inseridos da mesma forma usada da criação de `QListaFormaPagto`. No caso dos rótulos, há uma observação adicional: Para conseguir a tecla rápida, deve ser feito como nos botões: coloca-se o sinal "&" antes da letra de acesso rápido. Porém, ao fazer ist

o, pode-se perceber que o símbolo "&" vai aparecer e a letra não será sublinhada. O motivo é porque o rótulo não está associado a nenhum *widget* capaz de receber o foco[8]. Para ligar o rótulo com o *widget* que ele faz referência, deve ser usado o comando *Set Buddy* (algo como defina o companheiro). Ao selecionar o menu `Tools`⟶`Set Buddy` ou clicar no botão *Set Buddy*, o cursor do *mouse* se transformará em uma pequena cruz, e com ele dessa forma, o rótulo deve ser ligado ao *widget* referenciado. Feita a ligação, o símbolo "&" vai sumir e a letra logo após será sublinhada.

Falta agora fazer as conexões *signal/slot*. Para `QFormaPagto` serão feitas duas conexões: a do botão Grava (`botaoGrava`) com o *slot* `grava()` que deve ser criado e o botão Cancela (`botaoCancelar`) com o *slot* `close()`.

Terminado o desenho de `QFormaPagto`, pode-se encerrar o Designer e voltar ao KDevelop para que o sistema seja codificado. Assim, pode-se considerar encerrado o trabalho no Designer (a não ser, é óbvio, que seja necessário efetuar alguma alteração na interface). Para maiores esclarecimentos sobre a utilização do Designer, é recomendada a leitura de [QTMANUAL].

6.6. Criação das classes no KDevelop

Ao fechar o Designer e voltar para o KDevelop, nenhuma classe foi adicionada ao projeto, mas apenas dois arquivos com extenão `.ui` (*user interface*). No momento da compilação, os arquivos gerados no Designer serão convertidos em fontes C++ e é suficiente saber que serão criados os arquivos `qlistaformapagto.h` e `qformapagto.h` onde estarão declaradas as classes `QListaFormaPagto` e `QFormaPagto`. Na modelagem, foram concebidas as classes `KListaFormaPagto` e `KFormaPagto` e, para que elas representem os formulários criados no Designer, será usado o conceito de herança: as classes serão filhas de `QListaFormaPagto` e `QFormaPagto`. Para que a classe funcione de acordo com a modelagem, devem ser reimplementados os métodos correespondentes aos *slots* que foram definidos no Designer.

A classe `KFormaPagto` foi gerada pelo Umbrello, mas o Umbrello

não sabe que ela deve ser filha de `QFormaPagto`. Assim, deve-se alterar sua declaração para que ela se torne filha da classe gerada pelo Designer. Depois é só partir para a codificação de seus métodos. A classe `KFormaPagto` foi deixada para ser criada no KDevelop para ilustrar o seu funcionamento.

Para criar uma classe no KDevelop, deve-se selecionar o menu `Projeto`⟶`New Class`. Uma caixa de diálogo como a da Figura 6-18 irá se abrir. Esta caixa de diálogo contém duas abas para que sejam informados os dados da classe. No campo `Nome` deve-se colocar o nome da classe: `KListaFormaPagto`. Ao colocar o nome da classe, os campos *Header* e *Implementation* são preenchidos com o nome da classe todo em minúsculo com as respectivas extensões `.h` e `.cpp`. São esses os arquivos que serão manipulados depois da classe criada.

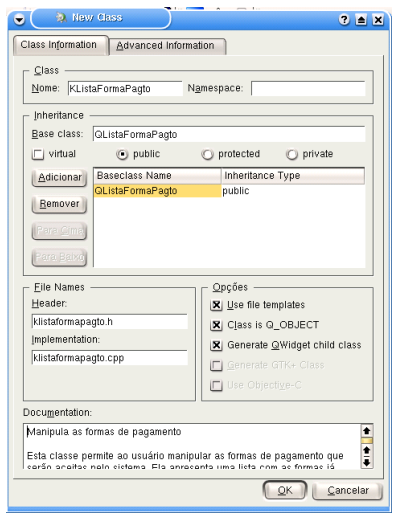

Figura 6-18. KDevelop - criando uma nova classe

À direita da janela, exite um grupo de opções da classe. Das opções

disponíveis, será marcado *Generate QWidget child class*. Ao marcar esta opção, a opção *Class is Q_OBJECT* também é selecionada. Isto é necessário para que a classe consiga trabalhar com o mecanismo *signal and slot* do Qt. O campo *Documentation* também é interessante ser preenchido. Por enquanto pode-se colocar exatamente a mesma documentação colocada na modelagem. Mais tarde, será visto como formatar esse texto para gerar uma documentação da API do sistema em diversos formatos (HTML, PDF, entre outros).

Preenchidos todos os campos, ao clicar em `Ok`, o KDevelop vai criar dois arquivos implementando a classe `KListaFormaPagto`. A classe é criada com dois métodos: o construtor `KListaFormaPagto` e o destrutor `~KListaFormapagto`. Para ver que a classe foi incorporada à estrutura do projeto, é só abrir a janela de classes do KDevelop que ela vai aparecer dentro do grupo `Classes`.

Criada a classe, devem ser acrescentados os seus métodos conforme a modelagem da classe. Para isso, deve-se clicar sobre o nome da classe com o botão direito e no menu que aparece, selecionar a opção `Add Method`. Uma caixa de diálogo como a mostrada pela Figura 6-19 se abre para que sejam digitados os dados do método.

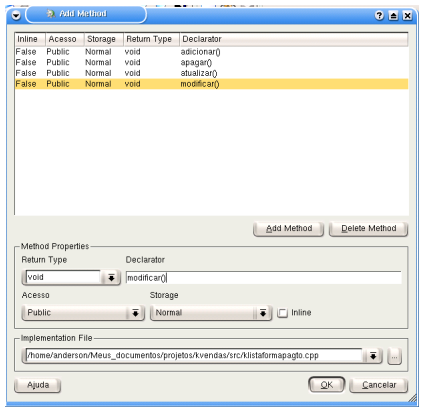

Figura 6-19. KDevelop - criação de métodos

O primeiro método que será acrescentado é o método `atualiza()`. Este método não vai retornar nenhum valor e, portanto o campo *Return Type* será `void`. O campo *Declarator* deve ser preenchido com o nome do método seguido dos parâmetros (se existir algum). Para a classe modelada, o campo deve ser preenchido com `atualizar()`. No campo *Acesso*, deve ser escolhida a opção `Public Slots` e no campo *Storage*, escolhe-se a opção `Virtual`. O acesso público se justifica porque ao clicar no botão `Atualizar`, o usuário na verdade está chamando este método. Da mesma forma, são criados os *slots* restantes: `inclui`, `exclui` e `altera`.

Criadas as classes, pode-se então partir para a codificação dos métodos, ou seja, trabalhar efetivamente com o C++. As classes de *interface* precisam das classes que acessam o banco de dados para funcionar e, portanto, a implementação do sistema vai ser iniciada por elas.

6.7. Implementação das classes

Ao criar uma classe, são acrescentados ao projeto dois arquivos que têm o mesmo nome da classe, porém em minúsculo um com extensão `.h` e outro `.cpp`. Se, na janela de classes, for dado um clique sobre o nome

da classe, o KDevelop abre uma janela mostrando o *header* e se o clique for sobre um dos métodos, o KDvelop abre o `.cpp` na posição onde está implementado o método.

Primeiramente, serão implmentadas as classes que acessam o banco de dados e depois, as classes de *interface* com o usuário, que fazem uso dessas classes. Depois de tudo codificado, será modificado o código da função `main()` para que ela chame a janela de cadastro de formas de pagamento ao invés da janelinha criada junto com o projeto.

Para cada classe, serão mostrados os fontes de cada método e será explicado o que ele está fazendo. Nessa explicação ficará explícito que o que se precisa de C++ é apenas o necessário para implementar os métodos. Nada de rotinas de baixo nível, ou chamadas complicadas de sistema, mas apenas o que as bibliotecas utilizadas exigem para funcionar.

Ao gerar os arquivos com os códigos fonte, tanto o Umbrello como o KDevelop geram várias linha de comentários no formato do DOxygen além de outras informações adicionais. Para mostrar a implementação das classes, todas as linhas adicionais foram removidas e assim, os fontes que serão mostrados contêm apenas as linhas válidas para o compilador.

6.7.1. Classes de acesso ao banco de dados

Antes de começar a implementar as classes de banco de dados, é necessário entender como o MySQL funciona. Basicamente a tarefa de obter dados do banco de dados consiste na seguinte seqüência de passos:

1. Abrir uma conexão com o banco de dados;
2. enviar uma instrução SQL;
3. receber o conjunto de registros resultado da instrução SQL;
4. obter os dados de cada registro do conjunto;
5. desalocar o conjunto de registros;

6. e finalmente, fechar a conexão com o banco de dados.

Para cada passo, deve existir uma variável para armazenar o resultado da operação realizada. Estas variáveis serão os atributos da classe. Portanto os atributos necessários para gerenciar a conexão com o MySQL são os seguintes:

conexão

> Este atributo vai conter um ponteiro que será o ponto de comunicação entre o sistema e o banco de dados. Ele é do tipo `MYSQL` que é uma `struct` definida em `mysql.h`.

registros

> Este atributo, será um ponteiro inicializado com `NULL` e será usado logo após executar uma instrução SQL do tipo `SELECT` para receber o conjunto de registros retornados pela instrução. O tipo deste atributo é `MYSQL_RES`, outra `struct` definida em `mysql.h`.

registro

> Este atributo vai conter os dados de um único registro do conjunto de regsitros `registros`. O tipo é `MYSQL_ROW`, também definido em `mysql.h`. Este atributo não deve ser um ponteiro.

Os detalhes da conexão com o banco de dados serão implementados na classe `QDatabase`, sendo que as outras classes devem apenas conhecer a linguagem SQL e usar os métodos implementados em `QDatabase`. Portanto, os atributos mostrados são da classe `QDatabase` e não serão usados diretamente, mas via métodos implementados nesta classe.

Todas as classes de banco de dados (`QDatabase, QListaFormaPagto e DFormaPagto` foram criadas pelo Umbrello, e portanto, a estrutura delas está pronta nos arquivos fontes gerado por ele. Assim, resta apenas o trabalho de alterar os arquivos fonte para implementar os métodos das classes.

6.7.1.1. Classe `QDatabase`

A classe `QDatabase` é a que vai conversar diretamente com a API do SGBD, ou seja, as chamadas às rotinas implementadas na biblioteca do MySQL, serão feitas por esta classe. As classes filhas de `QDatabase` não precisam saber conversar com o MySQL, pois `QDatabase`, vai cuidar dessa conexão.

A figura Figura 6-20 mostra a estrutura da classe `QDatabase`. É importante notar que ela é filha de `QObject`. Na verdade, não era necessário que ela fosse filha de `QObject`, mas isto foi feito por dois motivos. Um deles é que praticamente todas as classes da biblioteca Qt são filhas de `QObject` e seus construtores recebem um objeto desta classe ou de um de seus descendentes. Quando um objeto é destruído, ele destrói também os objetos que o receberam como filho. Se, por exemplo, for instanciada uma classe de *interface* e em seguida for instanciada outra classe recebendo a *interface*, quando a instaância da *interface* for destruída, automaticamente o segundo objeto também será destruído. Dessa forma não há o perigo de "esquecer" de destruir um objeto, pois a biblioteca Qt vai fazer isto automaticamente.

O segundo motivo pelo qual `QDatabase` é filha de `QObject`, é para ilustrar a utilização do mecanismo *signal and slot* do Qt, mais especificamente, a implementação de um sinal. Em `QDatabase`, sempre que ocorrer um erro na comunicação com o SGBD, um sinal de erro será emitido e que pode ser conectado a um *slot*.

```
 1: class QDatabase : public QObject
 2: {
 3:     Q_OBJECT
 4: public:
 5:     QDatabase(QObject *parent=0, const char *name=0);
 6:     ~QDatabase(  );
 7:
 8:     bool proximo(  );
 9:     QString obtemCampo( QString campo );
10:     long numeroRegistros(  );
11:
```

```
12:     bool executaSQL( QString sql );
13:     bool getResult(  );
14:
15: private:
16:     void initAttributes();
17:
18:     MYSQL *conexao;
19:     MYSQL_RES *registros;
20:     MYSQL_ROW registro;
21:
22: signals:
23:     void erro(const char *);
24:
25: public slots:
26:     void msgErro(const char *strErro);
27: };
```

Figura 6-20. Estrutura da classe `QDatabase`

A linha 3 da Figura 6-20, é uma macro da biblioteca Qt, necessária para toda classe descendente de `QObject`. O método `initAttributes()` foi criado pelo Umbrello para que os atributos da classe sejam inicializados com os seus valores padrão. A linha 24 declara um sinal para a classe, que será emitido todas as vezes que ocorrer um erro na conexão com o SGBD. É importante notar que este método não é implementado. A macro `Q_OBJECT` cuida da conversão desta declaração de sinal em um código que o compilador entende.

Por fim a linha 27 declara um *slot* que recebe um texto. Ele será usado nas classes filhas de `QDatabase` para exibir as mensagens de erro emitidas pela classe. Na verdade não é necessário que este *slot* específico seja usado, mas qualquer um que receba uma *string* implementado em qualquer classe. Este *slot* foi criado para ser o padrão. Para ilustrar este fato, a classe `DListaFormaPagto` usa este *slot* e a classe `KFormaPagto` conecta o sinal `erro` a um *slot* implementado em sua própria estrutura.

Depois de mostrar a estrutura da classe `QDatabase`, a partir de agora será mostrada a implementação de seus métodos. A Figura 6-21 implementa o construtor. A sua primeira tarefa é inicializar os atributos. Depois

disto ele faz a conexão com o SGBD. As linhas 8 e 10 ilustram a utilização do sinal `erro`. Se um erro ocorre, o sinal é emitido.

```
 1: QDatabase::QDatabase(QObject *parent,
 2:   const char *name) : QObject(parent, name){
 3:
 4:   initAttributes();
 5:   if ( (conexao = mysql_init(NULL)) )
 6:     if (! (mysql_real_connect(conexao, host,
 7:         usuario, senha, db, 0, NULL, 0)))
 8:       emit erro(mysql_error(conexao));
 9:   else
10:     emit erro(mysql_error(conexao));
11: }
```

Figura 6-21. O construtor da classe `QDatabase`

O método `initAttributes()` mostrado na Figura 6-22 é um método criado pelo Umbrello para inicializar os atributos da classe com seus valores padrão. Esta é uma forma interessante de trabalho, principalmente se a classe tem mais de um construtor e os valores padrão dos atributos forem o mesmo, pois não é necessário repetir para cada construtor as atribuições, mas apenas chamar o método `initAttributes()`.

```
void QDatabase::initAttributes( ){
  conexao = 0;
  registros = 0;
}
```

Figura 6-22. Implementação do método `initAttributes()`

A Figura 6-23 implementa o destrutor da classe. Ele primeiramente garante que o conjunto de registros será fechado e depois fecha a conexão com o SGBD.

```
QDatabase::~QDatabase(  ){

  if (registros) mysql_free_result(registros);
  mysql_close(conexao);

}
```

Figura 6-23. Destrutor de `QDatabase`

Na Figura 6-24 está implementado o método `executaSQL`, que é responsável pelo envio de instruções SQL ao SGBD. Ele recebe uma *string* contendo a instrução SQL e, se o SGBD conseguir executá-la, é retornado `true`, caso contrário o método retorna `false`. Também neste método, caso ocorra um erro, um sinal é emitido.

```
bool QDatabase::executaSQL( QString sql ){

  bool retorno;

  retorno = ( mysql_real_query(conexao, sql,
              sql.length()) == 0);
  if (!retorno) emit erro(mysql_error(conexao));
  return retorno;

}
```

Figura 6-24. Implementação do método `executaSql()`

Depois de executar uma instrução SQL do tipo `SELECT`, o primeiro passo é obter o conjunto de registros que o SGBD vai retornar. Isto é feito pelo método `getResult()`, mostrado na Figura 6-25. A primeira providência do método é confirmar se a conexão com o banco de dados está aberta e em seguida é verificado se o conjunto de registros está aberto. Se o conjunto de registros estiver aberto, ele é fechado antes. Depois dos testes, o resultado da instrução SQL executada é obtido do SGBD. Mais uma vez, se ocorrer algum erro, o sinal é emitido.

```
bool QDatabase::getResult(  ){

  if (!conexao) return false;

  if (registros) mysql_free_result(registros);
  registros = mysql_store_result(conexao);
  if (mysql_errno(conexao) != 0){
    emit erro(mysql_error(conexao));
    return false;
  }else
    return true;
}
```

Figura 6-25. Implementação do método `getResult()`

Depois de executar a instrução SQL e obter o conjunto de registros, deve-se usar o método `proximo()` mostrado na Figura 6-26. A cada chamada a este método, um registro é obtido do banco de dados. Quando o último registro for lido, na próxima leitura, a função `mysql_fetch_row()` vai retornar um valor `NULL`, fazendo com que o método retorne o valor `false`. Este valor de retorno pode ser usado como condição de parada na leitura dos registros do conjunto.

```
bool QDatabase::proximo(  ){
  return ((registro = mysql_fetch_row(registros)) ?
          true : false);
}
```

Figura 6-26. Implementação do método `proximo()`

Para obter os dados de cada registro, deve ser usado o método `obtemCampo()` mostrado na Figura 6-27. O método percorre a lista de campos que a instrução SQL retorna e ao encontrar o campo desejado, retorna o seu valor. Não é necessário conhecer a instrução executada para isso, pois o conjunto de registros tem todas as informações sobre os campos.

```
QString QDatabase::obtemCampo( QString campo ){

  unsigned int numCampos;
  unsigned int i;
  MYSQL_FIELD *campos;

  numCampos = mysql_num_fields(registros);
  campos = mysql_fetch_fields(registros);
  for(i = 0; (i < numCampos) &&
      (QString(campos[i].name) != campo); i++);
  if (i < numCampos)
    return QString(registro[i]);
  else
    return "";
}
```

Figura 6-27. Implementação do método `obtemCampo()`

Em algumas situações pode ser necessário saber o número total de registros que a instrução SQL retornou e isto é a tarefa do método `numeroRegistros` implementado como mostrado na Figura 6-28. Este método apenas retorna o valor da função `mysql_num_rows()`, implementada na biblioteca do MySQL.

```
long QDatabase::numeroRegistros(  ){
  return mysql_num_rows(registros);
}
```

Figura 6-28. Implementação do método `numeroRegistros()`

Finalmente, é mostrada na Figura 6-29 a implementação do método `msgErro()`. Este método é um *slot* de `QDatabase` que mostra na tela uma caixa de diálogo contendo um texto representando um erro que é passado como parâmetro. Ele foi implementado para ser o *slot* padrão para mostrar mensagens de erro do SGBD.

```
void QDatabase::msgErro(const char *strErro)
{
  QMessageBox::warning(0, "Banco de dados", strErro);
}
```

Figura 6-29. Implementação do *slot* `msgErro()`

Assim foi mostrada a implementação completa da classe `QDatabase`. Ficou evidente na implementação dos métodos que a conexão com o SGBD não é uma tarefa difícil. A grande vantagem de implementar o acesso à API do SGBD nessa classe é que, se ao invés do MySQL for usado outro SGBD como o PostgreSQL, apenas a classe `QDatabase` precisa ser alterada, pois as suas filhas apenas trabalham com instruções SQL que funcionam para qualquer SGBD.

6.7.1.2. Classe `DListaFormaPagto`

A Figura 6-30 mostra a estrutura da classe `DListaFormaPagto`. Na declaração da classe, é importante notar que ela é uma filha de `QDatabase` que implementa o acesso à API do MySQL. Ela contém apenas dois métodos: o construtor e `obtemDados()`.

```
class DListaFormaPagto : public QDatabase
{
public:
    DListaFormaPagto(QObject *parent=0, const char *name=0);
    bool obtemDados(  );
};
```

Figura 6-30. Estrutura da classe `DListaFormaPagto`

A Figura 6-31 mostra a codificação do construtor `DListaFormaPagto()`. Ele é muito simples apenas conectando o sinal emitido quando um erro ocorre a um *slot*.

```
DListaFormaPagto::DListaFormaPagto(QObject *parent,
  const char *name) : QDatabase(parent, name)
{
  connect(this, SIGNAL(erro(const char *)), this,
    SLOT(msgErro(const char *)));
}
```

Figura 6-31. O construtor de `DListaFormaPagto`

A Figura 6-32 mostra a implementação do método `obtemDados()`. Este método define a instrução SQL que deve ser enviada ao SGBD para obter uma lista com as formas de pagamento cadastradas no sistema. Definida a instrução SQL, executa-se o método `executaSQL()`, e se ele executar com sucesso, é executado tambéo método `getResult()`, ambos definidos na classe `QDatabase`.

```
bool DListaFormaPagto::obtemDados(  ){

  QString sql = "SELECT CODIGO,DESCRICAO,PRAZO,PESOCOM,"
  sql += "PESODESC FROM FORMAPAGTO WHERE SITUACAO='A'";
  return ( (executaSQL(sql)) ? getResult() : false);

}
```

Figura 6-32. Implementação do método `obtemDados()`

6.7.1.3. A classe `DFormaPagto`

A classe `DFormaPagto` foi criada para manipular os dados de uma forma de pagamento. Para isso, ela tem, além dos atributos usados para conectar ao banco de dados, atributos que representam as colunas da tabela `FORMAPAGTO`. A Figura 6-33 mostra a sua estrutura. A classe tem dois construtores, sendo um chamado quando for para criar uma nova forma de pagamento e o outro, que recebe o código da forma de pagamento como parâmetro, chamado quando for para alterar uma forma de pagamento existente.

```
class DFormaPagto : public QDatabase
{

public:
    DFormaPagto(QObject *parent=0, const char *name=0);
    DFormaPagto(QString formaPagto, QObject *parent=0,
                const char *name=0);

    bool alteraCampo( QString campo, QString valor );
    bool obtemDados(  );
    bool grava(  );
    bool exclui(  );
    bool valida(  )

private:
    QString codigo;
    QString descricao;
    QString prazo;
    QString pesocom;
    QString pesodesc;
    QString situacao;

};
```

Figura 6-33. Estrutura da classe `DFormaPagto`

A Figura 6-34 mostra a implementação dos dois métodos da classe. o primeiro não faz nada a não ser, em seu cabeçalho, chamar o construtor de `QDatabase`. O segundo construtor atribui o valor do código da forma de pagamento que será alterada. Depois disso ele busca os seus dados no banco de dados, chamando os métodos `obtemDados()` e `proximo()`.

```
DFormaPagto::DFormaPagto(QObject *parent,
  const char *name) : QDatabase(parent, name)
{
}

DFormaPagto::DFormaPagto(QString formaPagto,
  QObject *parent, const char *name)
```

```
  : QDatabase(parent, name)
{
  codigo = formaPagto;
  obtemDados();
  proximo();
}
```

Figura 6-34. Construtores da classe `DFormaPagto`

O método `obtemDados()` é implementado da mesma forma que em `DListaFormaPagto` alterando apenas a instrução SQL que é enviada ao banco de dados e por isto não será mostrado. A Figura 6-35 mostra a implementação do método `alteraCampo()`. Ele altera o valor de um dos atributos da classe. O atributo a ser alterado e seu valor são os parâmetros que o método recebe.

```
bool DFormaPagto::alteraCampo(QString campo,
  QString valor )
{

  if (campo == "DESCRICAO") descricao = valor;
  else if (campo == "PRAZO") prazo = valor;
  else if (campo == "PESOCOM") pesocom = valor;
  else if (campo == "PESODESC") pesodesc = valor;
  else return false;

  return true;

}
```

Figura 6-35. Implementação do método `alteraCampo()`

O próximo método, mostrado pela Figura 6-36, é o `grava()`, que grava os dados da forma de pagamento no banco de dados. Este método deve ser chamado sempre depois de `alteraCampo()` que atualiza os atributos da classe com os dados da forma de pagamento que devem ser enviados ao banco de dados. Antes de gravar os dados, o método `valida()` é chamado para verificar se os dados da forma de pagamento são válidos.

Depois de validados os dados, é feito um teste no atributo `codigo` para ver se ele tem algum valor. Se estiver vazio, significa que uma nova forma de pagamento está sendo criada. Caso contrário, uma forma de pagamento está sendo alterada. Para o primeiro caso é elaborada uma instrução INSERT e no segundo uma instrução UPDATE.

```
bool DFormaPagto::grava(  )
{
  QString sql;

  if (valida()){
    if (codigo.isEmpty()){
      sql = "INSERT INTO FORMAPAGTO (DESCRICAO,PRAZO,"
      sql += "PESOCOM,PESODESC) VALUES ('"
      sql += descricao + "','" + prazo + "'," +
             pesocom + "," + pesodesc + ")";
    else
      sql = "UPDATE FORMAPAGTO SET DESCRICAO='" +
            descricao + "',PRAZO='" + prazo + "',"
            + "PESOCOM=" + pesocom + ",PESODESC=" +
            pesodesc + " WHERE CODIGO=" + codigo;
    return executaSQL(sql);
  }else return false;

}
```

Figura 6-36. Implementação do método `grava()`

O método `exclui()`, mostrado na Figura 6-37, faz a exclusão de uma forma de pagamento. O comportamento da classe depende de ela ter sido usada em uma venda. Se a forma de pagamento ela já foi usada alguma vez, ela não é excluída, mas apenas marcada como inativa e se nunca foi utilizada, ela é excluída do sistema.

```
bool DFormaPagto::exclui(  ){

  QString sql;
  QDatabase dbVerifica;
```

```
  sql = "SELECT FORMAPAGTO FROM PAGAMENTO WHERE FORMAPAGTO="
        + codigo
  if (dbVerifica.executaSQL(sql) && dbVerifica.getResult()){
    if (dbVerifica.numeroRegistros() > 0)
      sql = QString("UPDATE FORMAPAGTO SET SITUACAO='I' "
            + "WHERE CODIGO=" + codigo;
    else
      sql = "DELETE FROM FORMAPAGTO WHERE CODIGO="
            + codigo;
    return executaSQL(sql);
  }else return false;

}
```

Figura 6-37. Implementação do método `exclui()`

Finalmente, para a classe `DFormaPagto`, é mostrado o método `valida()`, implementado como na Figura 6-38. Ele é muito simples porque a validação da forma de pagamento consiste em apenas verificar o preenchimento ou não de seus campos. Para fazer a verificação, é usada uma única expressão lóca que retorna `true` se os dados são válidos.

```
bool DFormaPagto::valida()
{
  return (!(descricao.isEmpty() && pesocom.isEmpty()
          && pesodesc.isEmpty()) &&
          ( (prazo=="S") || prazo=="N") );
}
```

Figura 6-38. Implementação do método `valida()`

6.7.2. Classes de *interface* com o usuário

Depois de implementadas as classes que vão fazer a comunicação com o SGBD, são mostradas as classes que possibilitam ao usuário interagir com o sistema. Todas as classes mostradas a partir de agora são filhas

de uma classe definida pela biblioteca Qt. Mais especificamente, a classe `KListaFormaPagto` é descendente de `QWidget` e a classe `KFormaPagto` é descendente de `QDialog`. Porém elas não são descendentes diretas das classes do Qt, mas de classes criadas pela *interface* concebida com o auxílio do Designer.

Na Seção 6.6 foi dito que, a partir da interface criada no Designer, são geradas classes que a implementam. Essas classes poderiam ser utilizadas para implementar o que foi modelado para o sistema, mas se alguma alteração for realizada na *interface* pelo Designer, ele recria os arquivos que implementam as classes e, portanto, a codificação teria que ser refeita. A melhor forma de trabalhar com classes de *interface* é criando-se classes filhas das criadas pelo Designer. Assim, mesmo que alguma modificação seja feita no Designer, a subclasse não é alterada e nenhum trabalho deve ser refeito. Este é o motivo pelo qual as classes `KListaFormaPagto` e `KFormaPagto` foram criadas como filhas de `QListaFormaPagto` e `QFormaPagto` criadas a partir da *interface* desenhada no Designer.

6.7.2.1. A classe `KListaFormaPagto`

A Figura 6-39 mostra a estrutura da classe `KListaFormaPagto`. Ela é declarada como filha de `QListaFormaPagto` que será gerada no momento da compilação do sistema. A sua função é montar uma lista de formas de pagamento cadastradas no banco de dados e fornecer meios para que o usuário possa incluir, alterar e excluir elementos desta lista. É importante notar que seus métodos, com exceção do construtor, são os mesmos métodos acrescentados no momento da criação da *interface* no Designer e representam os *slots* que vão tratar os eventos decorrentes da interação do usuário.

```
class KListaFormaPagto : public QListaFormaPagto
{
Q_OBJECT
public:
    KListaFormaPagto(QWidget *parent=0,
      const char *name=0, WFlags f=Qt::WDestructiveClose);
```

```
public slots:
    virtual void atualiza();
    virtual void inclui();
    virtual void exclui();
    virtual void altera();
};
```

Figura 6-39. Estrutura da classe `KListaFormaPagto`

Como pode ser visto na Figura 6-40 o construtor apenas chama o método `atualiza()` mostrado na mesma figura. Esse método é responsável pela montagem da lista de formas de pagamento cadatradas no sistema. Para obter os dados, ela usa uma instância de `DListaFormapagto`.

```
KListaFormaPagto::KListaFormaPagto(QWidget *parent,
  const char *name, WFlags f)
  : QListaFormaPagto(parent, name, f)
{
  atualiza();
}

void KListaFormaPagto::atualiza()
{

  listaFormaPagto->clear();

  DListaFormaPagto formaPagto;
  if (formaPagto.obtemDados())
    while (formaPagto.proximo())
      new KListViewItem(listaFormaPagto,
          formaPagto.obtemCampo("DESCRICAO"),
          formaPagto.obtemCampo("CODIGO"),
          formaPagto.obtemCampo("PRAZO"),
          formaPagto.obtemCampo("PESOCOM"),
          formaPagto.obtemCampo("PESODESC"));

}
```

Figura 6-40. Implementação do construtor e o método `atualiza()`

Os métodos `inclui()` e `altera()` mostrados na Figura 6-41 são muito parecidos. Ambos criam uma instância de `DFormaPagto` para manipular os dados da forma de pagamento diferindo apenas no construtor que é usado. A outra diferença entre os métodos é que em `altera()` é feito um teste para certificar que o usuário selecionou um elemento da lista para ser alterado.

```
void KListaFormaPagto::inclui()
{

  KFormaPagto *novaForma = new KFormaPagto(this,
    "novaFormaPagto", Qt::WDestructiveClose);
  if (novaForma->exec() == QDialog::Accepted)
    atualiza();
  delete novaForma;

}

void KListaFormaPagto::altera()
{

  if (!listaFormaPagto->selectedItem()){
    QMessageBox::warning(this, "Forma de pagamento",
      "Vocï¿½deve selecionar uma forma de pagamento da "
      "lista antes de modificar.");
    return;
  }
  KFormaPagto *formaPagto = new KFormaPagto(
    listaFormaPagto->selectedItem()->text(1), this,
    "alteraFormaPagto", Qt::WDestructiveClose);
  if (formaPagto->exec() == QDialog::Accepted)
    atualiza();
  delete formaPagto;

}
```

Figura 6-41. Implementação dos métodos `inclui()` e `altera()`

O último método de `KListaFormaPagto` é o `exclui()`. Ele primeiramente verifica se foi selecionada uma forma de paamento da lista, depois avisa o usuário que a forma de pagamento será excluída do sistema e finalmente cria uma instânacia de `DFormaPagto` para excluir a forma de pagamento. A Figura 6-42 mostra a sua implementação.

```
void KListaFormaPagto::exclui()
{
  if (!listaFormaPagto->selectedItem()){
    QMessageBox::warning(this, "Forma de pagamento",
      "Vocï¿½deve selecionar uma forma de pagamento da "
      "lista antes de apagar.");
    return;
  }

  if (QMessageBox::information(this, "Forma de pagamento",
      "A forma de pagamento abaixo: \n\n" +
      listaFormaPagto->selectedItem()->text(0)
      + "\n\nserï¿½apagada do banco de dados.\n"
      "Vocï¿½confirma a exclusï¿½?",
      "&Sim", "&Nï¿½", 0, 0, 1) == 0)
  {
    DFormaPagto formaPagto(
      listaFormaPagto->selectedItem()->text(1));
    if (formaPagto.exclui()) atualiza();
  }

}
```

Figura 6-42. Implementação do método `exclui()`

6.7.2.2. A classe `KFormaPagto`

A classe `KFormaPagto` monta uma janela que permite ao usuário digitar os dados de uma forma de pagamento e gravá-la no sistema. Ela tem dois construtores, sendo um chamado quando uma forma de pagamento deve ser incluída e outro quando a forma de pagamento deve ser alterada.

O atributo `codigo` irá armazenar o código da forma de pagamento que deve ser alterada. A Figura 6-43 mostra a sua estrutura.

```
class KFormaPagto : public QFormaPagto
{
public:
    KFormaPagto(QWidget *parent, const char *name=0,
      WFlags f=Qt::WDestructiveClose);
    KFormaPagto(QString fp, QWidget *parent=0,
      const char *name=0, WFlags f=Qt::WDestructiveClose);

    void grava(  );

private:
    QString codigo;

};
```

Figura 6-43. Estrutura da classe `KFormaPagto`

O primeiro construtor da classe, mostrado na Figura 6-44, deve ser chamado quando uma nova forma de pagamento estiver sendo adicionada ao sistema. Ele apenas cria validadores para impedir que o usuário digite dados inválidos. O segundo construtor, além de criar estes validadores, obtém a forma de pagamento cujo código foi passado como parâmetro e atualiza os campos da janela.

```
KFormaPagto::KFormaPagto(QWidget *parent,
  const char *name, WFlags f)
  : QFormaPagto(parent, name, f)
{
  codigo = "";

  QRegExp rx("^\\d?\\,\\d{2}$");
  linePesoCom->setValidator(
    new QRegExpValidator(rx, linePesoCom));
  linePesoDesc->setValidator(
    new QRegExpValidator(rx, linePesoDesc));
```

```
}

KFormaPagto::KFormaPagto(QString fp, QWidget *parent,
  const char *name, WFlags f)
  : QFormaPagto(parent, name, f)
{
  codigo = fp;

  QRegExp rx("^\\d?\\,\\d{2}$");
  linePesoCom->setValidator(
    new QRegExpValidator(rx, linePesoCom));
  linePesoDesc->setValidator(
    new QRegExpValidator(rx, linePesoDesc));

  // Obtï¿½ os dados da forma de pagamento cujo cï¿½igo
  // ï¿½fp
  DFormaPagto formaPagto(codigo);
  lineDescricao->setText(
    formaPagto.obtemCampo("DESCRICAO"));
  linePesoCom->setText(
    formaPagto.obtemCampo("PESOCOM").replace(".", ","));
  linePesoDesc->setText(
    formaPagto.obtemCampo("PESODESC").replace(".", ","));
  comboPrazo->setCurrentItem(
    ((formaPagto.obtemCampo("PRAZO") == "S") ? 0 : 1) );

}
```

Figura 6-44. Construtores de `KFormaPagto`

O método `grava()` faz seu trabalho dependendo do atributo `codigo`. Se ele tiver um valor, a forma de pagamento deve ser alterada e então é criada uma instância de `DFormaPagto` chamando seu construtor que recebe o código da forma de pagamento a alterar. Caso o atributo esteja vazio, a instância de `DFormaPagto` é criada usando o outro construtor. Depois disso, os campos da forma de pagamento são alterados e enviados ao banco de dados. Finalmente se a gravação foi bem sucedida, o método `accept()` que encerra o processamento da janela e indica seu resultado como `QDialog::accepted`. O método é implementado de acordo com a Figura 6-45.

```
void KFormaPagto::grava()
{
  DFormaPagto *fp;
  if (codigo.isEmpty())
    fp = new DFormaPagto();
  else
    fp = new DFormaPagto(codigo);

  fp->alteraCampo(
    "DESCRICAO", lineDescricao->text());
  fp->alteraCampo(
    "PRAZO", comboPrazo->currentText().left(1));
  QString p = linePesoCom->text();
  p.replace(',', '.');
  fp->alteraCampo("PESOCOM", p);
  p = linePesoDesc->text();
  p.replace(',', '.');
  fp->alteraCampo("PESODESC", p);
  bool resultado = fp->grava();

  delete fp;
  if (resultado) accept(); else reject();

}
```

Figura 6-45. Implementação do método `grava()`

6.7.3. A função `main()`

Depois de implementadas as classes, deve-se alterar a função `main()` criada pelo KDevelop para chamar a classe `KListaFormaPagto` que vai dar acesso ao cadastro de formas de pagamento.

Uma parte do código gerado pelo KDevelop deverá ser ignorada, pois ele está chamando a classe `Kvendas` que ele criou junto do projeto. Para implementação do sistema completo esse código pode ser usado como modelo para chamar a classe que vai representar a janela principal do aplicativo.

```
#include "kvendas.h"
#include <kapplication.h>
#include <kaboutdata.h>
#include <kcmdlineargs.h>
#include <klocale.h>

#include "klistaformapagto.h"

static const char *description =
    I18N_NOOP("A KDE KPart Application");

static const char *version = "0.1";

static KCmdLineOptions options[] =
{
//{ "+[URL]", I18N_NOOP( "Document to open." ), 0 },
  { 0, 0, 0 }
};

int main(int argc, char **argv)
{
    KAboutData about("kvendas", I18N_NOOP("KVendas"),
                     version, description,
                     KAboutData::License_GPL,
                     "(C) 2004 Anderson Pereira Ataides",
                     0, 0, "anderson.pa@persogo.com.br");
    about.addAuthor("Anderson Pereira Ataides", 0,
                    "anderson.pa@persogo.com.br" );
    KCmdLineArgs::init(argc, argv, &about);
    KCmdLineArgs::addCmdLineOptions( options );
    KApplication app;
    KListaFormaPagto *mainWin = 0;

    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    mainWin = new KListaFormaPagto(0, "listaFormaPagto");
    app.setMainWidget( mainWin );
    mainWin->show();
    args->clear();
```

```
    int ret = app.exec();

    delete mainWin;
    return ret;
}
```

As linhas 1 a 5 foram geradas pelo KDevelop e são necessárias devido a utilização de algumas estrutruras que estão declaradas nestes *headers*. O *include* da linha 7 já faz parte do projeto, pois é nesse arquivo que está declarada a classe `KListaFormaPagto`.

Da linha 9 à linha 34, são códigos gerados pelo KDevelop e são necessários porque o sistema é um sistema KDE e ele vai precisar dessa implementação. Dessas linhas, a mais importante é a 32, onde é criado o objeto `app` que representa a aplicação.

Na linha 36, é criado o objeto que é a janela de lista de formas de pagamento. Na linha 37 é informado ao aplicativo que o *widget* criado vai ser o principal. Na linha 38 o *widget* se torna visível. A linha 41 entra no loop da aplicação e o controle passa para a janela principal: o *widget* `mainWin`.

Quando a janela for fechada, a janela principal é encerrada e a aplicação sai do loop. Na linha 43, então o *widget* é desalocado e a execução do aplicativo é encerrada.

A implementação do caso de uso então está completa. Para ver o sistema funcionando é só mandar o KDevelop compilar e executar o sistema. Como pode-se ver na implementação, o conhecimento necessário de C++ se resume ao necessário para utilizar as bibliotecas escolhidas. Em nenhum momento foi necessário implementar rotinas de tratamento de tela, manipulação de arquivos ou conexões em baixo nível. Foram usadas apenas apenas funções pré-definidas nas bibliotecas.

A partir de agora a tarefa seria implementar cada um dos outros casos de uso e finalmente a janela principal do aplicativo para chamar cada uma das janelas que implementam os outros casos de uso. Para aprender mais sobre o KDevelop, e como programar aplicativos usando a biblioteca Qt, recomenda-se a leitura de [KDEVMANUAL] e [QTMANUAL].

Notas

1. `Widget` é um elemento visível na *interface*, como uma linha de texto, um botão e até mesmo a própria janela contendo os elementos. Corresponde ao conceito de `controle` no jargão Windows.

2. O KDeveop pode ser configurado para apresentar as janelas de diferentes modos como em forma de abas ou MDI (*Multiple Document Interface* - Interface de mltiplos documentos). O modo de exibição pode ser alterado em `Configurações`⟶`Configurar Gideon`.

3. A partir da versão 3, o KDevelop pode gerenciar mais de um alvo para implementar um sistema. Seria como se houvessem subsistemas dentro do sistema. Esses *targets*, então, representam os subsistemas e, ao criar um novo arquivo, o KDevlop pergunta a qual subsistema o novo arquivo vai pertencer. Como no projeto descrito neste trabalho não existirá nenhum subsistema, sempre deverá ser usado o *target* atual.

4. Para selecionar mais de um arquivo em uma lista, clica-se no primeiro para selecioná-lo e com a tecla `Ctrl` pressionada, clca-se nos outros arquivos.

5. Para este trabalho, convencionou-se que todos os nomes de arquivos serão escritos com caracteres minúsculos, seguindo o padrão Linux.

6. Se, ao invés do Designer, uma outra janela aparecer pedindo qual o aplicativo deve ser usado para abir o arquivo, deve ser digitado `designer`. Para que o KDevelop sempre abra o Designer automaticamente, deve-se clicar na opção `Lembrar da associação de aplicativo para este arquivo`.

7. Se a barra de propriedades não estiver visível, deve-se clicar com o botão direito na barra de ferramentas e marcar a opção `Property Editor/Signal Handler`.

8. Receber o foco significa tornar-se o elemento ativo. Por exemplo, ao clicar em uma linha de texto, o cursor passa a piscar nesta linha, tornando-a ativa.

Capítulo 7. Documentação do sistema

Neste capítulo, serão mostradas as ferramentas de documentação disponíveis no Linux. Para documentar o sistema e elaborar um manual de usuário, foi adotado o DocBook e, para documentação das API's das classes, foi usado o DOxygen. Essas ferramentas são muito interessantes porque a partir de um único código fonte, elas geram documentação em vários formatos como páginas web (HTML), *PostScript* (PS), *Portable Document Format* (PDF) entre outros.

7.1. Documentando o sistema com o DocBook

Para escrever a documentação do sistema, bem como criar manual de utilização, podería ser escolhido qualquer processador de texto disponível no mercado, porém isso deixa o desenvolvedor preso ao formato que o processador escolhido disponibiliza, além do que no momento de escrever o texto, deve-se estar preocupado com a formatação do documento.

Para que o sistema possa ser mais flexível, a documentação deve estar disponível nos mais variados formatos que podem ser do gosto de quem vai usar, ou estudar essa documentação. Felizmente, existe o DocBook, uma ferramenta para escrever textos técnicos que, a partir de um arquivo ou conjunto de arquivos escritos em formato XML ou SGML, ele consegue gerar o documento nos formatos mais populares entre a comunidade do *software* livre.

Um outro motivo que levou a adotar essa ferramenta, foi que ela é recomentada pela *TLDP - The Linux Documentation Project (http://www.tldp.org/)* entidade que padroniza a documentação em sistemas Linux.

O DocBook, não é um programa, nem um processador de textos. Ele na verdade é um DTD (*Document Type Definition* - Definição de Tipo de Documento). Assim como o HTML, ele é uma linguagem de marcação definida em SGML/XML. Ao instalar o DocBook no sistema, devem ser instaladas também algumas ferramentas que vão ler o arquivo SGML/XML e, de acordo com o DTD e as folhas de estilo, vão gerar o documento final. Essa conversão é feita usando um dos *scripts* disponibilizados pelas

distribuições Linux. Para gerar o documento em formato HTML, usa-se o `docbook2html`. Para gerar em *Postscript*, é usado o `docbook2ps`.

A grande vantagem de usar esse tipo de ferramenta para escrever a documentação, é que não é preciso se preocupar com a forma com que o documento será gerado, mas apenas com seu conteúdo. Ao escrever um parágrafo, não é informado ao DocBook o espaçamento ou o tamanho das letras, apenas é dito onde começa e onde termina o parágrafo. Também não é necessário se preocupar em montar índices, lista de tabelas e figuras. Quando as ferramentas forem processar o arquivo, estes elementos são criados automaticamente.

Um exemplo de documento escrito usando o DocBook é este trabalho. Ele foi escrito em formato SGML e depois foi processado para gerar o documento final. Esse modo de escrever pode poupar um bocado de tempo do autor, que tem a única preocupação de escrever o texto, de entender o conteúdo. A formatação é toda feita pelo DTD com as folhas de estilo.

Na Figura 7-1, é mostrado um trecho do fonte deste documento em formato SGML. Note que são usadas *tags* similares às do HTML para indicar o início e o fim de um tipo de texto (parágrafo, texto itálico).

```
<para>
Para ilustrar o funcionamento do Umbrello, ser&aacute;
mostrado como definir a classe
<classname>DListaFormaPagto</classname>.
Esta classe &eacute; respons&aacute;vel por obter
os dados das formas de pagamento do banco de dados para
que seja elaborada uma lista que ser&aacute; exibida
ao usu&aacute;rio. Depois de criada a classe, deve-se
dar um duplo clique sobre ela para abrir a janela de
propriedades que &eacute; onde s&atilde;o
informados os atributos, m&eacute;todos e
documenta&ccedil;&atilde;o da classe.
</para>
<para>
A primeira tarefa &eacute; documentar a classe,
indicando para que ela serve e fornecendo uma
```

```
vis&atilde;o geral sobre seu funcionamento.
N&atilde;o &eacute; necess&aacute;rio neste
momento entrar em detalhes sobre os atributos e
m&eacute;todos porque eles ter&atilde;o sua
pr&oacute;pria documenta&ccedil;&atilde;o.
</para>

<figure>
  <title>Umbrello - documentando uma classe</title>
  <graphic format="png" align="center" scale="25"
    fileref="figuras/umbrello08.png">
</figure>
```

Figura 7-1. Trecho de um arquivo no formato SGML

Para escrever o arquivo SGML, pode ser usado qualquer editor de texto que consiga gravar texto puro. Se for usado processador de texto como OpenOffice ou Word, deve-se ter o cuidado de, ao gravar o arquivo, selecionar o formato texto, para que ele não seja gravado no formato proprietário do processador escolhido.

Depois do documento todo digitado, deve-se usar um dos *scripts* disponíveis para gerar o documento em sua forma final. O Docbook gera o documento em uma forma padrão que pode ser alterada. Para fazer essa alteração, deve-se trabalhar com as folhas de estilo que define a forma final do documento.

Para aprender como escrever documentos usando o DocBook, é indicada a leitura de [DOCBOOK], que é um guia para o DTD, També é recomendada a leitura de [DSSSL] e [XSLT] para aprender a trabalhar com folhas de estilo para gerar o documento final a partir de fontes SGML e XML respectivamente.

7.2. O KDE DocBook

O KDE DocBook, é um subconjunto do DocBook elaborado para gerar documentação de aplicativos KDE. Ao criar um projeto no KDevelop, um documento XML é gerado como modelo para que o desenvolvedor

escreva o seu próprio manual do sistema. O KDevelop coloca este arquivo dentro do subdiretório `doc/en` e dá a ele o nome de `index.docbook`.

Dessa forma, o manual do usuário também será escrito usando uma variação do DocBook, e conseqüentemente o desenvolvedor deve se preocupar apenas com seu conteúdo, pois a forma final será ditada pelo DTD definido pelo KDE DocBook. Mais uma informação importante é que, por conhecer o KDE DocBook, o próprio KDevelop chama os programas necessários para gerar a documentação final.

7.3. Documentando as classes com DOxygen

A criação de uma documentação da API do sistema pode parecer redundância, pois todas as classes do sistema foram devidamente documentadas quando foram criadas no Umbrello. Porém nem sempre o usuário tem o Umbrello instalado para ver essa documentação. É muito mais provavel que ele tenha um navegador ou um leitor de arquivos PDF instalado em seu computador.

Um outro ponto é que no momento da implementação, pode surgir a necessidade de explicar o motivo de um parâmetro ter tal formato, ou como será em detalhes o funcionamento da classe. Esse tipo de documentação geralmente é feito no próprio arquivo fonte e assim o próprio arquivo deve ser aberto para ver tal documentação.

Para evitar abrir os fontes, existem ferramentas como o KDoc e DOxygen que conseguem ler esses comentários e gera, assim como o DocBook, documentos em vários formatos para facilitar o acesso a essa documentação. Além da documentação escrita, no caso do DOxygen, também é gerada a hierarquia das classes e, na documentação de cada classe, é mostrado um pequeno diagrama de classes mostrando onde a classe está na hierarquia.

Entre as duas ferramentas, foi escolhido o DOxygen. O KDevelop conhece tanto o KDoc como o DOxygen e consegue executar qualquer uma delas gerando a documentação das classes. Como padrão o KDevelop usa o DOxygen, fato que influenciou bastante na sua escolha. Outro motivo que

influiu na escolha foi que o KDE usa o DOxygen para documentar sua API.

Portanto, para gerar a documentação da API do sistema, deve-se pedir para o KDevelop rodar rodar o DOxygen. Ao rodar o DOxygen, serão criados subdiretórios adicionais dentro do diretório onde estão os arquivos fontes do sistema. Dentro desses subdiretórios, o DOxygen coloca os arquivos gerados a partir dos arquivos fontes do projeto. Depois, é só disponibilizar esses arquivos para que o futuro colaborador do sistema não precise nem abrir os arquivos fontes para aprender como trabalham as classes do sistema.

Mais informações sobre o DOxygen, pode ser obtida a partir da leitura de [DOXYGEN].

```
/**
 * \brief Manipula as formas de pagamento
 *
 * Esta classe permite ao usuï¿½io manipular as formas
 * de pagamento que serï¿½ aceitas pelo sistema. Ela
 * apresenta uma lista com as formas jï¿½cadastradas e
 * permite ao usuï¿½io incluir, alterar e excluir formas
 * de pagamento do banco de dados.
 *
 * Ela ï¿½uma subclasse de QListaFormaPagto, que ï¿½a
 * classe de interface gerada pelo Designer.
 *
 * \author Anderson Pereira Ataides
 **/
```

Figura 7-2. Trecho de documentação no estilo DOxygen

Capítulo 8. Considerações finais

O desenvolvimento de sistemas em Linux, assim como em qualquer outra plataforma somente terá sucesso se houverem ferramentas que auxiliem o trabalho desde a concepção até a implementação. Todas as ferramentas que foram utilizadas, facilitam bastante o trabalho de desenvolvimento em todas as suas fases e, portanto, desenvolver sistemas em Linux não pode ser considerado como uma tarefa que só pode ser executada por *hacker*. Como foi apresentado, sabendo como usar as ferramentas e tendo o mínimo de conhecimento em C++ para implementar as classes, é possível implementar sistemas em Linux com muita facilidade.

Em um sistema comercial, a complexidade se encontra muito mais nas regras de negócio do que na implementação do sistema e, devido a essa característica, saber usar as ferramentas de modelagem acaba tendo um peso maior sobre o desenvolvimento, pois como foi visto, depois de modelado, a implementação exigiu apenas saber o mínimo da linguagem de programação para implementá-lo.

Apesar de as ferramentas utilizadas serem bem elaboradas, elas apresentaram alguns problemas durante a utilização que foram os seguintes:

Umbrello

Algumas vezes, ao tentar mover elementos entre pastas o Umbrello não faz a movimentação, sendo necessário repetir a operação. Alguns movimentos realmente não são possíveis, como passar um diagrama de caso de uso de *Use case view* para *Logical view*, mas os movimentos restritos são sinalizados pelo próprio Umbrello. O problema é que ele não respeitou movimentos que ele sinaliza como sendo permitidos.

No diagrama de seqüência, ao criar uma mensagem, se for selecionado um dos métodos descritos para associar a mensagem, depois de fechar, o Umbrello não se lembra da associação e coloca a operação selecionada como *custom operation*.

Ao documentar alguns elementos, o Umbrello "perde" o que

foi digitado no campo *Documentation*. Este problema ocorre principalmente com o último elemento acrescentado à classe (método ou atributo). O problema foi parcialmente contornado, acrescentando um elemento por vez, fechando a janela e salvando o arquivo.

Ainda falando sobre a documentação, ao gerar o código fonte, o Umbrello não respeita os saltos de linha digitados na documentação. Assim, nos códigos fonte, o Umbrello junta toda a documentação ficando como se fosse um único parágrafo.

KDevelop

Ao usar o *Automake manager* para acrescentar os arquivos gerados pelo Umbrello ao projeto, o KDevelop terminou inesperadamente. Ao entrar novamente, os arquivos não foram acrescentados e a operação teve que ser repetida. O problema ocorreu eventualmente e pode ser que tenha ocorrido por falha no sistema de arquivos ou outro motivo externo ao KDevelop.

O gerenciador de classes do Umbrello com freqüência perdeu a referência da definição dos métodos das classes. Assim, ao clicar sobre o nome de um método, ele sempre abria a estrutura da classe ao invés de abrir o ponto onde o método é implementado.

Apesar dos problemas, as ferramentas se comportaram muito bem e ajudaram bastante tanto a modelagem como a implementação do sistema.

8.1. Conclusões

Depois de ver a facilidade na utilização das ferramentas analisadas, é possível concluir que o desenvolvimento de sistemas em Linux é viável, podendo inclusive oferecer vantagens tanto para o desenvolvedor como para o usuário. Sob o ponto de vista econômico, a utilização de ferramentas livres vai ajudar a diminuir a relação custo x benefício, pois a produtivi-

dade aumenta e o custo diminui. O custo, porém não pode ser considerado zero, pois o tempo de aprendizado deve ser contado no cálculo de custos.

Por apresentar interfaces simples e por serem bem documentadas, as ferramentas apresentadas não vão impor um grande tempo de aprendizado, o que vai possibilitar que em pouco tempo o desenvolvedor consiga produzir sistemas bem projetados e documentados.

Com a escolha do C++ como linguagem de programação usando o KDevelop para gerenciar os arquivos fonte, foi possível mostrar que, apesar de ser considerada uma linguagem difícil, é relativamente fácil utilizá-la para codificar as classes de um sistema. Talvez a maior dificuldade seja aprender a usar as ferramentas, mas a própria documentação ajuda a vencer esta dificuldade.

Portanto, pode-se considerar que este trabalho consegue cumprir seu objetivo de provar que o desenvolvimento de sistemas em Linux pode ser tão simples e prazeroso como desenvolver em plataformas mais populares como o Microsoft Windows. Ao mostrar essa simplicidade, é possível convencer os fornecedores de sistemas comerciais a desenvolver uma versão de suas aplicações para o Linux.

8.2. Contribuições

Se ficou provado que no Linux existem ferramentas competentes e que facilitam muito o desenvolvimento de sistemas, por que existem tão poucos programadores desenvolvendo para Linux? A resposta está justamente na dificuldade enfrentada para escolher as ferramentas: pouca divulgação. A partir do momento em que existe um trabalho que descreve pelo menos algumas dessas ferramentas, existe a possibilidade de que o número de desenvolvedores Linux aumente, facilitando assim que o Linux saia dos servidores das empresas para ocupar lugar também nas mesas de trabalho dos seus funcionários.

Além da contribuição para a popularização do Linux, também pode-se dizer que há uma importante contribuição para usuários Linux no Brasil, pois a maioria das documentações e estudos sobre o Linux e seus aplicati-

vos estão em inglês, o que dificulta sua utilização por grande parte dos seus possíveis usuários. Apesar de existirem projetos de tradução de documentação de manuais e documentação dos sistemas, existem poucos trabalhos em português de avaliação de ferramentas como este trabalho.

Ao convencer os desenvolvedores da viabilidade de se projetar e implementar sistemas em Linux, também consegue-se contribuir para o mercado, já que as empresas poderão reduzir seus custos com aquisição de software, pois o Linux, assim como as ferramentas apresentadas, são livres e podem ser obtidas a um custo muito baixo, visto que existem distribuições Linux acompanhando revistas de informática que são vendidas em bancas de revista.

Portanto o projeto consegue dar contribuições importantes à comunidade, principalmente a usuária de *software* livre que tem à disposição mais um trabalho que trata do desenvolvimento de sistemas usando ferramentas dessa natureza.

8.3. Trabalhos futuros

Existem alguns projetos que podem ser desenvolvidos e acrescentados ao protótipo desenvolvido, usando o conhecimento obtido a partir do estudo deste trabalho. Entre eles pode-se citar uma implementação mais completa do sistema, incluindo a parte de emissão de documentos fiscais, retirada do escopo deste trabalho para deixar o sistema desenvolvido mais genérico. Outra funcionalidade que falta no protótipo é o controle de estoque que, para ser implementado, deve ser incluído no projeto o controle da entrada de mercadorias.

Saindo do escopo das vendas e estoque, usando o protótipo como modelo, podem ser desenvolvidos sistemas para todos os setores de uma empresa, como sistemas do departamento financeiro, ou folha de pagamento. Como foi mostrado a simplicidade de desenvolver sistemas em Linux, abre-se um leque de possibilidades que atualmente é pouco explorado, pois a grande maioria dos sistemas tanto os já existentes como os novos são desenvolvidos para o Windows.

Uma implementação interessante de se fazer é tornar a *interface* dinâmica, ou seja, gerada não no momento da compilação, mas no momento da execução do aplicativo. Isto é possível de se fazer, ligando bibliotecas do Designer ao aplicativo, fazendo com que os formulários fiquem independente do resto da aplicação. Dessa forma, a qualquer momendo, mesmo depois do sistema compilado, o *layout* das janelas pode ser completamente modificado sem ter que recompilar o sistema. Este recurso é interessante porque fica fácil personalizar o sistema.

Com relação ao projeto do sistema, podem ser criadas classes para tratar as regras de negócio. Do modo que o protótipo foi criado, essas regras foram incorporadas às classes de dados, que validam os dados antes de enviá-los ao SGBD.

Com relação às ferramentas, existe a possibilidade de se mostrar também outras que, apesar de competentes, foram descartadas no desenvolvimento do protótipo de sistema comercial. Um exemplo é a dupla GTK/GNOME, que na maioria das distribuições Linux são oferecidas como alternativa ao Qt/KDE, usado neste trabalho. Da mesma forma, pode ser analisado o Glade, como alternativa ao KDevelop e o PostgreSQL substituindo o MySQL. Com uma análise dessa, o futuro desenvolvedor terá em suas mãos pontos de vista diferentes sob alguns aspectos do desenvolvimento de sistemas Linux, mas convergentes em um ponto: facilitar o seu trabalho de projetar sistemas completos para o Linux.

Apêndice A. Diagramas de seqüência

Neste apêndice serão mostrados alguns dos diagramas de seqüência modelados no sistema. Para poupar tempo e agilizar o processo de desenvolvimento, nem tudo precisa ser modelado. Devido às semelhanças entre vários diagramas, apenas alguns precisam ser modelados para entender a lógica e orientar a implementação. Nos outros casos, a implementação pode seguir a mesma lógica, e o que iria mudar no diagrama seriam apenas as classes que o compõem.

O primeiro diagrama (Figura A-1) modela a manutenção das formas de pagamento do sistema. Ele pode ser usado para entender a lógica de todos os cadastros do sistema, pois são todos parecidos: Apresenta-se uma tela com uma lista do que já está cadastrado e o usuário então pode incluir, alterar ou excluir elementos no sistema.

O diagrama mostrado na Figura A-3 modela a negociação com o cliente. O diagrama da Figura A-2 modela a emissão da lista de preços de produtos. Este diagrama serve de modelo para todos os outros relatórios que tem uma lógica bem parecida: o sistema exibe uma tela pedindo os parâmetros do relatório, os dados são lidos do banco de dados e enviados à impressora.

Os diagramas mostrados podem não estar totalmente em conformidade com a implementação, pois pode ser que haja a necessidade de incluir algum método ou atributo para melhorar a codificação do sistema. [FURLAN] diz na modelagem de um estudo de caso:

> "... Conforme a modelagem avança rumo a uma impoementação física, há um aumento considerável na quantidade de detalhes a serem cobertos sendo difícil tratá-los de maneira estática como o onteúdo de um livro."

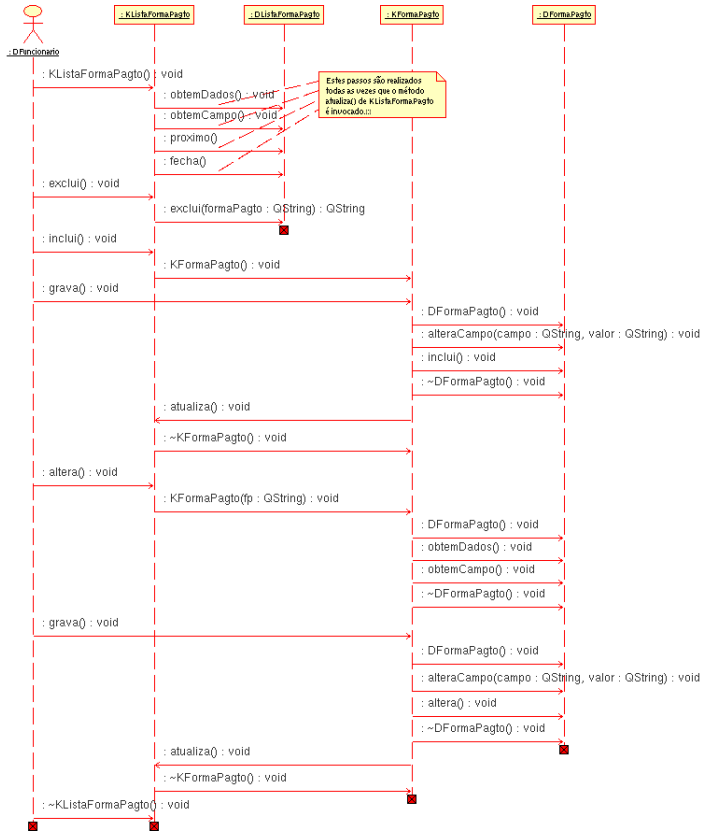

Figura A-1. Manutenção do cadastro de formas de pagamento

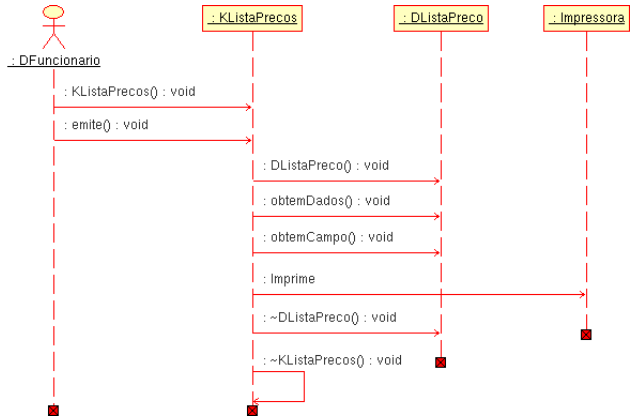

Figura A-2. Emissão de lista de preço

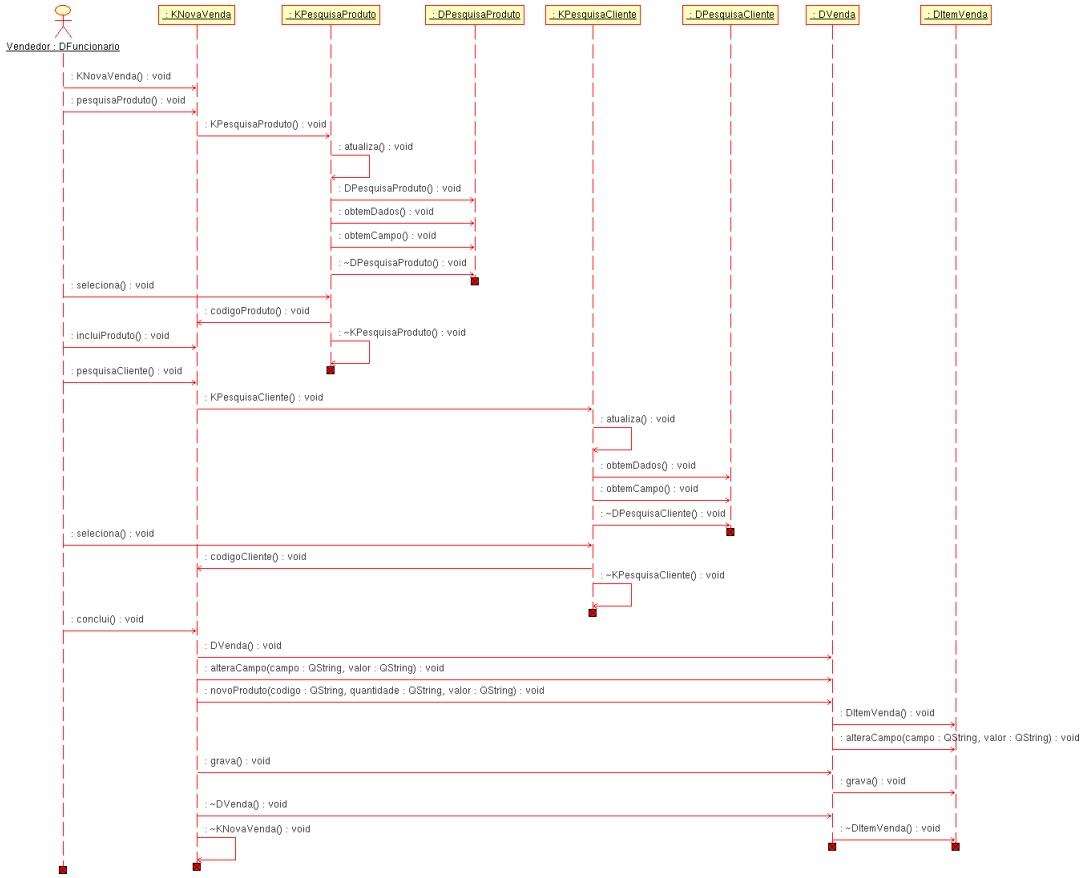

Figura A-3. Negociação da venda com cliente

Apêndice B. Preparando o banco de dados MySQL

Este apêndice tem a finalidade de mostrar como preparar o banco de dados para funcionar junto com o sistema desenvolvido neste trabalho. Será mostrado como instalar o MySQL no Conectiva Linux 9 e como gerar as tabelas de acordo com a especificação do modelo de dados.

B.1. Instalando o MySQL

Todo o desenvolvimento deste trabalho foi feito em uma distribuição padrão, no caso o Conectiva Linux 9. Todas as ferramentas utilizadas foram escolhidas tentando obedecer ao que é disponibilizado pela distribuição. O MySQL é fornecido na distribuição e a partir de agora será mostrado como instalar o sistema gerenciador de banco de dados a partir dos pacotes `RPM` da Conectiva.

O MySQL pode também ser baixado do site oficial do MySQL (http://www.mysql.com/), porém ao escolher esta opção, os arquivos podem ser armazenados em locais que não são o padrão da Conectiva e assim pode ser necessário fazer algumas alterações no sistema para que os programas do MySQL bem como as bibliotecas sejam encontradas.

Para o projeto deste trabalho, devem ser instalados pelo menos dois pacotes: o MySQL e o respectivo pacote de desenvolvimento (MySQL-devel). A maneira mais fácil de instalar estes pacotes no Conectiva Linux 9 é através do Synaptic (Figura B-1), uma ferramenta de gerenciamento de pacotes. Ao entrar no Synaptic, ele pede a senha do usuário *root* e depois mostra uma lista de pacotes. Neste momento é necessário estar de posse dos CD's do Conectiva ou estar conectado à internet para que o Synaptic possa fazer o *download* dos pacotes.

Na lista que o Synaptic exibe, devem ser escolhidos os pacotes MySQL, MySQL-client e MySQL-devel. Se for de interesse instalar a documentação para consulta e aprender mais sobre o MySQL, o pacote MySQL-doc também deve ser instalado. Para instalar um pacote, deve-se selecioná-lo e clicar no botão Instalar, à direita da lista. Depois dos pacotes selecionados e marcados para a instalação, deve-se dar um clique no botão Proceder localizado na barra de ferramentas do Synaptic.

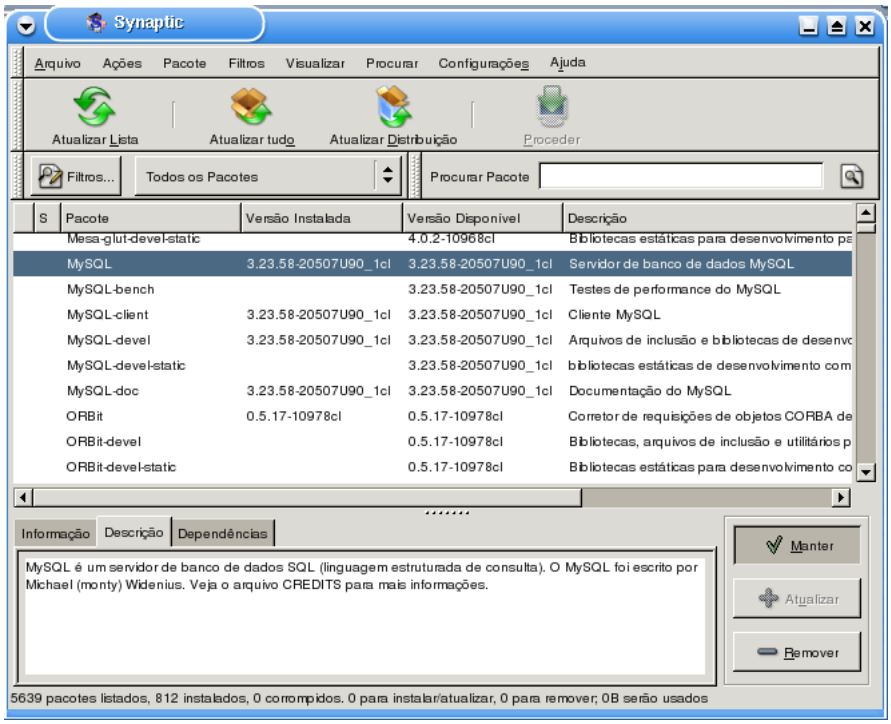

Figura B-1. Synaptic - ferramenta de gerenciamento de pacotes

Depois que o Synaptic concluir sua tarefa, o MySQL estará instalado no sistema, mas ainda não é possível trabalhar com o servidor do banco de dados. Antes de iniciar o servidor, deve-se abrir uma janela de console, logar no sistema como usuário *root* e digitar os comandos mostrados na Figura B-2 para preparar o servidor MySQL.

```
# mysql_install_db
# mysql_createdb
```

Figura B-2. Preparando o servidor MySQL

O primeiro comando vai inicializar o repositório do MySQL criando o banco de dados que gerencia o controle de acesso e suas respectivas

tabelas. O segundo comando vai criar a senha do usuário *root* do MySQL. Este usuário *root* não tem nada a ver com o *root* do Linux. Ele é o superusuário do banco de dados. Se possível ele deve ter uma senha diferente da senha do *root* do Linux.

Depois do sistema estar preparado, pode-se iniciar o servidor conforme mostrado na Figura B-3 e assim o sistema está pronto para criar a base de dados de acordo com o modelo de dados.

```
# service mysql start
```

Figura B-3. Iniciando o servidor MySQL

B.2. Criando o banco de dados

Como foi visto no Capítulo 5, as tabelas do banco de dados podem ser criadas usando um *script* SQL. Porém, antes de executar o *script*, o banco de dados deve ser criado para receber as tabelas. Depois de criado o banco de dados, deve-se informar ao MySQL qual o usuário que terá permissão de acesso ao banco de dados recém criado.

A maneira mais fácil de se criar o banco de dados e dar permissão de acesso a um usuário é usando o programa `mysql`, que é um *shell* para enviar comandos ao servidor MySQL. Para executar o programa, deve-se abrir uma janela de terminal proceder como mostrado na Figura B-4.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.58-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figura B-4. Chamando o programa `mysql`

Quando o programa `mysql` é invocado ele vai pedir a senha de acesso e neste ponto deve ser digitada a senha que foi criada para o usuário *root*. Informada a senha, o *shell* vai mudar e fica pronto para receber comandos SQL.

O primeiro comando SQL que será digitado é o que vai criar o banco de dados no qual as tabelas serão armazenadas. Logo em seguida, será criado o usuário e respectiva senha para usar o banco de dados criado como mostrado na Figura B-5.

```
mysql> CREATE DATABASE kvendas;
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON kvendas.* TO kvendas@localhost
    -> IDENTIFIED BY 'kvendas' WITH GRANT OPTION;
Query OK, 0 rows affected (0.02 sec)

mysql> GRANT ALL PRIVILEGES ON kvendas.* TO kvendas@'%'
    -> IDENTIFIED BY 'kvendas' WITH GRANT OPTION;
Query OK, 0 rows affected (0.02 sec)

mysql>
```

Figura B-5. Criando o banco de dados e o usuário

A primeira instrução SQL cria o banco de dados 'kvendas' no MySQL. Note que o nome do banco de dados está em minúsculo. Deve-se tomar muito cuidado com os nomes dos bancos de dados e de tabelas, pois o MySQL faz distinção de letras maiúsculas e minúsculas para os nomes do banco de dados das tabelas.

Os dois próximos comandos criam um usuário para acesso local e outro para acesso via rede. Os comandos mostrados criam usuário com acesso completo e dessa forma ele vai poder fazer qualquer coisa no banco de dados, inclusive alterar a estrutura das tabelas ou mesmo apagá-las. Para o propósito deste trabalho esse modo de criar usuários é satisfatório, mas em um ambiente de trabalho real, deve-se preocupar mais com a segurança, criando um banco de dados reservado ao desenvolvimento e outro para

a produção. Da mesma forma deve-se restringir o acesso aos usuários de forma que eles possam fazer apenas o necessário para seu trabalho. É extremamente recomendada a instalação do pacote MySQL-doc e a sua leitura para aprender como trabalhar com o seu sistema de privilégios e controle de acesso.

Criado o banco de dados e os usuários que vão utilizá-lo, pode-se criar as tabelas que foram concebidas na análise.

B.3. Elaborando o *script* para criar as tabelas

No Capítulo 5, foi mostrado um trecho de um *script* SQL para criar a tabela de forma de pagamento. Nesta seção será apresentado o *script* completo para criação de todas as tabelas do banco de dados. O *script* cria não só as tabelas, mas também os índices que melhoram a performance de consultas e dos relacionamentos.

Outra coisa que pode ser notado é que pode-se inserir comentários para explicar alguma coisa no *script*. Este recurso é muito interessante, pois é possível tornar o *script* auto-explicativo, ou seja, ao ler o seu conteúdo consegue-se entender o que ele está fazendo.

O *script* mostrado na Figura B-6 deve ser salvo em um arquivo texto puro, sem nenhuma formatação. Ele foi preparado especificamente para funcionar com o MySQL. Para fazê-lo funcionar em outro sistema, deve-se efetuar algumas alterações.

```
-- ************************************************
-- Tabela.........: USUARIO
-- Elemento do DER.: Entidade Usuario
-- Funï¿½o.........: Usuarios do sistema.
-- ************************************************
CREATE TABLE USUARIO (
  CODIGO    TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  NOME      VARCHAR(50) NOT NULL,
  VENDEDOR  VARCHAR(1) NOT NULL DEFAULT 'S',
  TELEFONE  VARCHAR(10) NOT NULL,
```

```
  SENHA     VARCHAR(32),
  SITUACAO  VARCHAR(1) NOT NULL DEFAULT 'A'
);
CREATE INDEX USNOME ON USUARIO (NOME);

-- ***************************************************
-- Tabela.........: GRUPOPRODUTO
-- Elemento do DER.: Entidade Grupo Produto
-- Funï¿½o.........: Agrupar produtos semelhantes. Tem
--                  ligaï¿½o direta com a tabela
--                  PRODUTO.
-- ***************************************************
CREATE TABLE GRUPOPRODUTO (
  CODIGO    TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  DESCRICAO VARCHAR(40) NOT NULL,
  COMISSAO  DECIMAL(3,2) NOT NULL DEFAULT 0.0,
  GRUPOPAI  TINYINT REFERENCES GRUPOPRODUTO (CODIGO)
            MATCH FULL ON UPDATE CASCADE
            ON DELETE RESTRICT
);
CREATE INDEX GPDESCRICAO ON GRUPOPRODUTO (DESCRICAO);
CREATE INDEX GPGRUPOPAI ON GRUPOPRODUTO (GRUPOPAI);

-- ***************************************************
-- Tabela.........: PRODUTO
-- Elemento do DER.: Entidade Produto
-- Funï¿½o.........: Cadastro dos produtos que serï¿½
--                  vendidos pelo sistema
-- ***************************************************
CREATE TABLE PRODUTO (
  CODIGO    INT NOT NULL PRIMARY KEY,
  GRUPO     TINYINT NOT NULL
            REFERENCES GRUPOPRODUTO (CODIGO) MATCH FULL
            ON DELETE RESTRICT ON UPDATE CASCADE,
  DESCRICAO VARCHAR(80) NOT NULL,
  APELIDO   VARCHAR(20) NOT NULL,
  UNIDADE   VARCHAR(3) NOT NULL DEFAULT 'UNI',
  FRACAO    VARCHAR(1) NOT NULL DEFAULT 'N',
  DESCONTO  DECIMAL(10,2) NOT NULL DEFAULT 0.0,
  COMISSAO  DECIMAL(5,2) NOT NULL DEFAULT 0.0,
```

```
  SITUACAO  VARCHAR(1) NOT NULL DEFAULT 'A'
);
CREATE INDEX PRGRUPO ON PRODUTO (GRUPO);
CREATE INDEX PRDESCRICAO ON PRODUTO (DESCRICAO);
CREATE INDEX PRAPELIDO ON PRODUTO (APELIDO);

-- ************************************************
-- Tabela.....: CLIENTE
-- Elemento do DER.: Entidade Cliente
-- Funï¿½o.....: Clientes da empresa - parq quem
--                    serï¿½emitida a venda.
-- ************************************************
CREATE TABLE CLIENTE (
  CODIGO        SMALLINT NOT NULL AUTO_INCREMENT
                PRIMARY KEY,
  NOME          VARCHAR(80) NOT NULL,
  APELIDO       VARCHAR(50) NOT NULL,
  ENDERECO      VARCHAR(80),
  BAIRRO        VARCHAR(40),
  CIDADE        VARCHAR(30),
  ESTADO        VARCHAR(2),
  CEP           VARCHAR(8),
  TEL1          VARCHAR(10),
  TEL2          VARCHAR(10),
  FAX           VARCHAR(10),
  EMAIL         VARCHAR(80),
  CPF           VARCHAR(11),
  IDENTIDADE    VARCHAR(20),
  CNPJ          VARCHAR(14),
  INSCEST       VARCHAR(20),
  DATACADASTRO  TIMESTAMP NOT NULL,
  TIPO          VARCHAR(20),
  CADASTRO      TINYINT NOT NULL
                REFERENCES USUARIO (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE RESTRICT,
  VENDEDOR      TINYINT NOT NULL
                REFERENCES USUARIO (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE RESTRICT,
  SITUACAO      VARCHAR(1) NOT NULL DEFAULT 'V'
);
```

```
CREATE INDEX CLNOME ON CLIENTE (NOME);
CREATE INDEX CLAPELIDO ON CLIENTE (APELIDO);
CREATE INDEX CLCPF ON CLIENTE (CPF);
CREATE INDEX CLCNPJ ON CLIENTE (CNPJ);
CREATE INDEX CLCADASTRO ON CLIENTE (CADASTRO);
CREATE INDEX CLVENDEDOR ON CLIENTE (VENDEDOR);

-- ************************************************
-- Tabela.........: REFERENCIA
-- Elemento do DER.: Entidade Referencia
-- Funï¿½o.........: Referencias de clientes
-- ************************************************
CREATE TABLE REFERENCIA (
  CODIGO          SMALLINT NOT NULL
                  REFERENCES CLIENTE (CODIGO) MATCH FULL
                  ON UPDATE CASCADE ON DELETE CASCADE,
  NUMERO          TINYINT NOT NULL AUTO_INCREMENT,
  NOME            VARCHAR(40) NOT NULL,
  TELEFONE        VARCHAR(10) NOT NULL,
  CONTATO         VARCHAR(20) NOT NULL,
  DTULTIMACOMPRA  DATE NOT NULL,
  VLULTIMACOMPRA  DECIMAL(10,2) NOT NULL,
  DTMAIORCOMPRA   DATE NOT NULL,
  CLMAIORCOMPRA   DECIMAL(10,2) NOT NULL,
  ANOCADASTRO     SMALLINT,
  CONCEITO        VARCHAR(20),
  OBSERVACAO      VARCHAR(200),
  PRIMARY KEY (CODIGO,NUMERO)
);

-- ************************************************
-- Tabela.........: FORMAPAGTO
-- Elemento do DER.: Entidade Forma Pagto
-- Funï¿½o.........: Formas de pagamento aceitas na
--                  empresa
-- ************************************************
CREATE TABLE FORMAPAGTO (
  CODIGO     TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  DESCRICAO  VARCHAR(30) NOT NULL,
  PRAZO      VARCHAR(1) NOT NULL DEFAULT 'S',
```

```
  PESOCOM    DECIMAL(5,2) NOT NULL DEFAULT 1,
  PESODESC   DECIMAL(5,2) NOT NULL DEFAULT 1,
  SITUACAO   VARCHAR(1) DEFAULT 'A'
);
CREATE INDEX FPDESCRICAO ON FORMAPAGTO (DESCRICAO);

-- ************************************************
-- Tabela.....: VENDA
-- Elemento do DER.: Entidade Venda
-- Funï¿½o.....: Registro de vendas de produtos
-- ************************************************
CREATE TABLE VENDA (
  NUMERO        INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  DATA          TIMESTAMP NOT NULL,
  VENDEDOR      TINYINT NOT NULL
                REFERENCES USUARIO (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE RESTRICT,
  CLIENTE       SMALLINT NOT NULL
                REFERENCES CLIENTE (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE RESTRICT,
  TIPOPAGTO     VARCHAR(1) NOT NULL DEFAULT 'V',
  TIPODOCUMENTO VARCHAR(1) NOT NULL DEFAULT 'O',
  INFORMACOES   VARCHAR(100),
  VENDAFINAL    INT REFERENCES VENDA (NUMERO) MATCH FULL
                ON UPDATE CASCADE ON DELETE SET NULL,
  USREGISTRO    TINYINT NOT NULL
                REFERENCES USUARIO (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE RESTRICT,
  DTESTORNO     DATETIME,
  USESTORNO     TINYINT
                REFERENCES USUARIO (CODIGO) MATCH FULL
                ON UPDATE CASCADE ON DELETE SET NULL,
  MOTIVOESTORNO VARCHAR(100)
);
CREATE INDEX VEDATA ON VENDA (DATA);
CREATE INDEX VEVENDEDOR ON VENDA (VENDEDOR);
CREATE INDEX VECLIENTE ON VENDA (CLIENTE);
CREATE INDEX VEVENDAFINAL ON VENDA (VENDAFINAL);
CREATE INDEX VEUSREGISTRO ON VENDA (USREGISTRO);
CREATE INDEX VEDTESTORNO ON VENDA (DTESTORNO);
```

```
CREATE INDEX VEUSESTORNO ON VENDA (USESTORNO);

-- ************************************************
-- Tabela.........: ITEMVENDA
-- Elemento do DER.: Relacionamento Estï¿½entre as
--                   entidades Venda e Produto.
-- Funï¿½o.........: Itens vendidos
-- ************************************************
CREATE TABLE ITEMVENDA (
  VENDA       INT NOT NULL
              REFERENCES VENDA (NUMERO) MATCH FULL
              ON UPDATE CASCADE ON DELETE CASCADE,
  SEQUENCIA   TINYINT NOT NULL AUTO_INCREMENT,
  PRODUTO     INT NOT NULL
              REFERENCES PRODUTO (CODIGO) MATCH FULL
              ON UPDATE CASCADE ON DELETE RESTRICT,
  QUANTIDADE  DECIMAL(10,2) NOT NULL DEFAULT 1.0,
  PRECOVENDA  DECIMAL(10,2) NOT NULL,
  PRECOLISTA  DECIMAL(10,2) NOT NULL,
  PRIMARY KEY (VENDA,SEQUENCIA)
);
CREATE INDEX IVPRODUTO ON ITEMVENDA (PRODUTO);

-- ************************************************
-- Tabela.........: PAGAMENTO
-- Elemento do DER.: Relacionamento Paga entre Venda e
--                   Forma Pagto
-- Funï¿½o.........: Registra o pagamento da venda
-- ************************************************
CREATE TABLE PAGAMENTO (
  VENDA       INT NOT NULL
              REFERENCES VENDA (NUMERO) MATCH FULL
              ON UPDATE CASCADE ON DELETE CASCADE,
  SEQUENCIA   TINYINT NOT NULL AUTO_INCREMENT,
  FORMAPAGTO  TINYINT NOT NULL
              REFERENCES FORMAPAGTO (CODIGO) MATCH FULL
              ON UPDATE CASCADE ON DELETE RESTRICT,
  VENCIMENTO  DATE NOT NULL,
  VALOR       DECIMAL(10,2) NOT NULL,
  PRIMARY KEY (VENDA,SEQUENCIA)
```

```
);
CREATE INDEX PGFORMAPAGTO ON PAGAMENTO (FORMAPAGTO);
CREATE INDEX PGVENCIMENTO ON PAGAMENTO (VENCIMENTO);
```

Figura B-6. O *script* SQL para criar as tabelas do banco de dados

B.4. Executando o *script*

Para enviar o *script* so servidor a fim de que ele crie as tabelas, pode-se usar o mesmo *shell* usado para criar o banco de dados e o usuário, ou seja, o programa `mysql`. A diferença é que ao invés de digitar cada um dos comandos, será usado o recurso de redirecionamento de entrada do Linux para que o *shell* receba os comandos de um arquivo, no caso o *script*.

O primeiro passo para criar as tabelas no banco de dados então será abrir uma janela de terminal. Supondo que o nome do arquivo contendo os comandos SQL se chama `kvendas.sql`, deve-se chamar o programa `mysql` da forma mostrada na Figura B-7.

```
$ mysql -u kvendas -p kvendas <kvendas.sql
```

Figura B-7. Executando o *script* SQL

O comando mostrado diz ao *shell* do MySQL que será usado o usuário `kvendas` (-u kvendas), que ele precisa pedir a senha do usuário (-p), que ele deverá usar o banco de dados `kvendas` e que o conteúdo do arquivo `kvendas.sql` deve ser usado.

O programa então vai pedir a senha do usuário e depois vai executar todos os comandos contidos no arquivo `kvendas.sql`. Depois de executar os comandos, o programa vai finalizar e a partir desse momento as tabelas estarão criadas no banco de dados.

Para se certificar de que as tabelas realmente foram criadas, deve-se chamar o programa `mysql` novamente e digitar o comando para listar as tabelas do banco de dados como mostrado na Figura B-8.

```
$ mysql -u kvendas -p kvendas
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 3.23.58-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW TABLES;
+-------------------+
| Tables_in_kvendas |
+-------------------+
| CLIENTE           |
| FORMAPAGTO        |
| GRUPOPRODUTO      |
| ITEMVENDA         |
| PAGAMENTO         |
| PRODUTO           |
| REFERENCIA        |
| USUARIO           |
| VENDA             |
+-------------------+
9 rows in set (0.00 sec)

mysql>
```

Figura B-8. Listando as tabelas do banco de dados

Assim o banco de dados está pronto para uso. Mais uma vez, é bom lembrar que é extremamente recomendáda a instalação do pacote MySQL-doc que contém a documentação do MySQL. O que foi mostrado neste apêndice sobre o MySQL é apenas o necessário para conseguir fazer o sistema desenvolvido neste trabalho trabalhar com dados do banco de dados. Num ambiente de produção serão necessárias informações extra acerca de segurança, bem como aprender mais sobre os comandos SQL suportados pelo MySQL e também aprender mais sobre a API C usada para implementar um sistema que acessa um servidor MySQL.

Referências bibliográficas

[SILBERSCHATZ] Henry F. Korth e Abraham Silberschatz, *Sistemas de banco de dados*, 1989, McGraw-Hill.

[FURLAN] José Davi Furlan, *Modelagem de objetos através da UML*, 1998, Makron Books.

[TCMMANUAL] Frank Dehne, Roel J. Wieringa, e Henk R. van de Zandschulp, *Toolkit for Conceptual Modeling (TCM) - User's Guide and Reference*, 2003, Disponível na Internet em http://wwwhome.cs.utwente.nl/~tcm/.

[QTMANUAL] Trolltech team, *Qt Reference Documentation*, 2002, Disponível na Internet em http://doc.trolltech.com/3.1/index.html.

[KDEVMANUAL] Bernd Gehrmann, Caleb Tennis, e Bernd Pol, *KDevelop User Manual*, 2004, Disponível na Internet em http://docs.kde.org/en/3.2/kdevelop/kdevelop/.

[DOCBOOK] Norman Walsh e Leonard Muellner, *DocBook: The Definitive Guide*, 2002, O'Reilly & Associates, Inc.

[XSLT] Bob Stayton, *DocBook XSL: The Complete Guide*, 2003, Sagehill Enterprises.

[DSSSL] Norman Walsh, *The Modular DocBook Stylesheets*, 2000, Disponível na Internet em http://docbook.sourceforge.net/release/dsssl/current/doc/.

[DOXYGEN] Dimitri van Heesch, *Doxygen Manual*, 2004, Disponível na Internet em http://www.stack.nl/~dimitri/doxygen/manual.html.