UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

CLAUDIR PEREIRA DOS SANTOS

INTEGRAÇÃO DO ECLIPSE COM WINDOWBUILDER PRO COMO SOLUÇÃO PARA O DESENVOLVIMENTO DE SISTEMAS DESKTOP

MONOGRAFIA DE ESPECIALIZAÇÃO

MEDIANEIRA
2011

CLAUDIR PEREIRA DOS SANTOS

INTEGRAÇÃO DO ECLIPSE COM WINDOWBUILDER PRO COMO SOLUÇÃO PARA O DESENVOLVIMENTO DE SISTEMAS DESKTOP

Monografia apresentada como requisito parcial à obtenção do título de Especialista na Pós Graduação em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná – UTFPR – Campus Medianeira.

Orientador: Prof. Msc. Everton Coimbra de Araújo.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Especialização em Engenharia de Software

TERMO DE APROVAÇÃO

INTEGRAÇÃO DO ECLIPSE COM WINDOWBUILDER PRO COMO SOLUÇÃO PARA O DESENVOLVIMENTO DE SISTEMAS DESKTOP

Por

Claudir Pereira dos Santos

Esta monografia foi apresentada às 18:45 h do dia 06 de dezembro de 2011 como requisito parcial para a obtenção do título de Especialista no curso de Especialização em Engenharia de *Software*, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. O acadêmico foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado com louvor e mérito.

____
Prof. Msc. Everton Coimbra de Araújo
UTFPR – Campus Medianeira
(orientador)

____
Prof. Msc. Fernando Schütz
UTFPR – Campus Medianeira

____
Prof. Msc. Alan Gavioli
UTFPR – Campus Medianeira

Dedico à minha família.

AGRADECIMENTOS

A todos aqueles que de uma forma ou de outra contribuíram para a execução deste trabalho.

“A sabedoria tem os seus excessos e não é menos necessário moderá-la do que à loucura.”

(MICHEL DE MONTAIGNE)

RESUMO

SANTOS, Claudir Pereira dos. INTEGRAÇÃO DO ECLIPSE COM WINDOWBUILDER PRO COMO SOLUÇÃO PARA O DESENVOLVIMENTO DE SISTEMAS DESKTOP. 2011. 76 f. Monografia (Especialização em Engenharia de Software). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

Este trabalho tratou sobre o uso de *frameworks* no desenvolvimento Java para *desktop.* Como embasamento teórico, foi referenciado assuntos ligados a produtividade no desenvolvimento de sistemas, técnicas para desenvolvimento produtivo, métricas para monitorar o desenvolvimento e aspectos sobre algumas ferramentas que podem ser usadas. A coleta de dados foi iniciada executando um aplicativo para criação da base dados que permitiu trabalhar visualmente e depois replicar as mudanças ao banco de dados. Durante a geração das entidades, também coletou-se dados ligados a agilidade para criação das mesmas. Na elaboração das janelas do sistema e na vinculação por meio de *bindings,* foi analisada a facilidade para desenho, vinculação dos componente e auxílio para trabalhar com o código. Em seguida, foi evidenciado como criar um relatório para Java e adicioná-lo ao projeto principal. Por fim, foi demonstrado como gerar um arquivo executável Java para ser distribuído para usuários finais. Como auxílio na exposição dos dados, foi usado recorte de tela e cópia do código fonte. Em seguida, foi feita a análise dos dados onde se verificou o impacto positivo do uso dos *frameworks*. No final foi comentado os resultados obtidos com o estudo e apresentadas as considerações finais.

Palavras-chave: *Frameworks, Java, software;*

ABSTRACT

SANTOS, Claudir Pereira dos. INTEGRATION OF ECLIPSE WITH WINDOWBUILDER PRO AS SOLUTION FOR DEVELOPMENT DESKTOP SYSTEMS. 2011. 76 f. Monografia (Especialização em Engenharia de Software). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

This paper deals with the use of frameworks for developing Java Desktop.com theoretical basis, it was referred issues related to productivity in systems development, development of production techniques, metrics to monitor progress and aspects of some tools that can be used. Data collection began running an application to create the database that lets you work visually and then replicate the changes to the generation of banco.Durante entities, also collected data related to the agility to create the drawing of windows mesmas.Na system and linking through bindings, we analyzed the facility to design, component and linking of aid to work with código.Em was then shown how to create a report for Java and add it to the main project. Finally, it was shown how to generate a Java executable file to be distributed to end users. As an aid in the exposure data, was used to screen clipping and copy the code fonte.Em next step was to analyze the data where there was the positive impact of the use of frameworks. In the end the results were discussed with the study and presents the final considerations.

Keywords: Frameworks, Java, software;

LISTA DE FIGURAS

Figura 1 - Criando relacionamento entre tabelas 28
Figura 2 - Sincronizando o modelo de dados..29
Figura 3 - Baixando e configurando EclipseLink 30
Figura 4 - Project Explorer e Data Source Explorer após criação do projeto 30
Figura 5 - Botão de navegação entre código e formulário mais algumas ferramentas do WindowBuilder Pro...32
Figura 6 - Definindo *Absolute layout* 32
Figura 7 - Paleta *Structure* e suas opções 33
Figura 8 - Tela para cadastro de pessoas criada com WindowBuilder Pro...............34
Figura 9 - Criando entidades de forma reversa..35
Figura 10 - Gerenciando criação de entidades..36
Figura 11 - Configurando registro de eventos em *persistense.xml* 37
Figura 12 - Tela para configuração das vinculações..38
Figura 13 - Iniciando vinculação de *JTable*...43
Figura 14 - Vinculando *jTextField* a *jTable*...46
Figura 15 - Criando vinculação entre botões e tabela...52
Figura 16 - Resultado das vinculações entre *Jtable* e *JButton*..................................52
Figura 17 - Resultado da vinculação de campos a tabela..53
Figura 19 - Ativando janelas no *iReport*...60
Figura 20 - Janelas do *iReport*...61
Figura 21 - Criando campos através de código SQL no *iReport*..............................62
Figura 22 - Adicionando campo ao formulário e removendo bandas desnecessárias 63
Figura 23 - Adicionando parâmetro ao consulta Sql do *iReport*...............................64
Figura 24 - Compilando relatório..65
Figura 25 - Instalando fonte para o *iReport*...65
Figura 26 - Bibliotecas necessárias para *jasper*...66
Figura 27 - Gerando arquivo *jar* executável do projeto 68

LISTA DE QUADROS

Quadro 1 - Código fonte de *personTableModel* 42
Quadro 2 - Declarando objetos para vinculação do formulário principal 42
Quadro 3 - Código fonte de *PersonDao* 45
Quadro 4 - Carregando dados na *jTable*..46
Quadro 5 - Código fonte de CityDao 48
Quadro 6 - Declarando Lista e DAO das cidades...48
Quadro 8 - Código fonte do método *toString* da entidade *City* sobrescrito 49
Quadro 9 - Métodos *Sets* originas da entidade *Person*..50
Quadro 10 - Código fonte de *AbstractModelObject*..50
Quadro 12 - Código do método do botão *new*...53
Quadro 14 - Código do método do botão *change* 55
Quadro 15 - Código do método do botão *delete* 55
Quadro 16 - Atualizando campo *id* da entidade *Person* nos componentes visuais...56
Quadro 17-Código do método para verificação do formulário...................................56
Quadro 18 - Código do método do botão *save*..57
Quadro 19 - Código do método do botão *search* 57
Quadro 21 - Código do método do botão *Print*..67

SUMÁRIO

1 INTRODUÇÃO 12
1.1 OBJETIVO GERAL 14
1.2 OBJETIVOS ESPECÍFICOS 14
1.3 JUSTIFICATIVA 14
1.4 ORGANIZAÇÃO DO TRABALHO 16
2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 A PRODUTIVIDADE EM DESENVOLVIMENTO DE SISTEMAS 17
2.2 TÉCNICAS PARA DESENVOLVIMENTO PRODUTIVO 17
2.3 MÉTRICAS PARA ACOMPANHAMENTO DO DESENVOLVIMENTO 19
2.3.1 Linha de Código (LOC) 20
2.3.2 Pontos de Função (FP) 20
2.3.3 Modelo de Custo Construtivo (COCOMO) 21
2.4 ASPECTOS GERAIS DO ECLIPSE E OUTRAS FERRAMENTAS 22
2.4.1 Histórico do Eclipse e sua Comunidade 22
2.4.2 MarketPlace 23
2.4.3 Data Tools Platform (DTP) 23
2.4.4 EclipseLink 24
2.4.5 WindowBuilder Pro 24
2.4.6 BeansBinding 25
2.4.7 Mysql 26
2.4.8 IReport 26
3 PROCEDIMENTOS METODOLÓGICOS DA PESQUISA 28
3.1 BANCO DE DADOS 28
3.2 ENTIDADES 29
3.3 TELAS E VÍNCULOS *BINDING* 31
3.3.1 Instalando os *Frameworks* 31
3.3.2 Tela Principal 31
3.3.3 Criação das Entidades 34
3.3.4 Arquivo *Persistence.xml* 36
3.3.5 Biblioteca *Beansbinding* e aba *Binding* 37

3.3.6 Tabela Modelo e o Vinculo ao *JTable* 38
3.3.7 Objeto de Acesso a Dados (DAO)....................43
3.3.8 JTextFields e sua Vinculação ao JTable 46
3.3.9 *JComboBox* Vinculado e Alimentado com Dados 46
3.3.10 Entidades Preparadas para *Beansbinding* 49
3.3.11 Controle de Ativação e Desativação dos Componentes da Tela....................51
3.3.12 Método do Botão *New*....................53
3.3.13 Método do Botão *Cancel*....................54
3.3.14 Método do Botão *Change*....................54
3.3.15 Método do Botão *Delete*....................55
3.3.16 Método do Botão S*ave*....................55
3.3.17 Método do Botão *Search*....................57
3.3.18 Método para Exibir Cidade Correta no *ComboBox* 58
3.4 RELATÓRIOS NO *IREPORT*....................59
3.4.1 *Drive* de Conexão a Base de Dados no *iReport*....................59
3.4.2 Exibição de Janelas no *iReport*....................60
3.4.3 Consulta *Sql* para Gerar os Campos do *iReport* 61
3.4.4 Adição de Campos ao Relatório....................62
3.4.5 Parâmetro para o Relatório 63
3.4.6 Compilação do Relatório 64
3.4.7 Geração de Arquivo *jar* das Fontes....................65
3.5 ADICIONAR RELATÓRIO NO PROJETO....................66
3.6 ARQUIVO EXECUTÁVEL DO PROJETO 67
3.7 ANÁLISE DOS DADOS....................68
4 RESULTADOS E DISCUSSÃO 70
5 CONSIDERAÇÕES FINAIS 71
5.1 CONCLUSÃO....................71
5.2 TRABALHOS FUTUROS 72
REFERÊNCIAS....................73

1 INTRODUÇÃO

A produtividade é um ponto que pode definir se um projeto de *software* obterá ou não sucesso. Quanto maior a trajetória a ser percorrida, maior a relevância da produtividade. Isso impactará em custo e também nas metas a serem atingidas. Um projeto que se torne muito caro em relação ao seu beneficio, possivelmente estará fadado ao fracasso. O mesmo pode ocorrer com um sistema que não seja entregue no tempo esperado.Sem produtividade ou sem a eficiência do processo produtivo, dificilmente uma empresa vai ser bem-sucedida ou até mesmo sobreviver no mercado (MACEDO, 2002).

São vários os fatores que podem implicar na redução da produtividade na criação ou manutenção de um produto. A função de um engenheiro de *software* é visualizar e diminuir esses riscos. Cada projeto tem seus complicadores, mas provavelmente haverão elementos comuns. Entre eles, pode estar o *framework* usado para desenvolvimento.

Quanto a produtividade, a sua gestão está se tornando um dos quesitos essenciais na formulação das estratégias de competitividade das empresas mas as relações entre produtividade e lucratividade dificilmente podem ser diretamente estabelecidas (MACEDO, 2002).

Com isso, pode se entender que mesmo uma empresa sendo produtiva não haverá garantias que a mesma obterá bons resultados, pois este é apenas um dos fatores para o sucesso.

A medição da produtividade, que é a capacidade da empresa gerar produto, é feita principalmente por indicadores de natureza físico operacional, por exemplo, X unidades de bens e serviços por unidade de tempo, produção física por número de horas trabalhadas, entre outros. Quando os bens e serviços produzidos pela empresa não são homogêneos, esse método de medição possui limitações, pois, nessas condições, é impossível a agregação desses bens e serviço sem uma única quantidade física total de bens e serviços produzidos pela empresa (MACEDO, 2002).

No caso de um projeto de *software,* os entregáveis são os produtos produzidos em determinado tempo. Nesse caso também pode existir a falta de homogeneidade de alguns dos produtos como fator negativo.

Para otimizar a produtividade e minimizar o impacto na produção de produtos diferentes, este trabalho sugere o uso de *frameworks* para desenvolvimento. O reuso de *software* tem sido um dos principais objetivos da engenharia de *software* e com o surgimento do paradigma da orientação a objetos, a tecnologia adequada para reuso de grandes componentes tornou-se disponível e resultou na definição de *frameworks* orientados a objetos. As principais vantagens de um *framework* são o aumento do reuso e a redução do tempo para desenvolvimento (CÔRTES, 2004).

De maneira sintetizada, pode definir-se *frameworks* como um conjunto de classes-base (esqueleto) sobre o qual uma ferramenta é construída (REINALDO, 2003).

A engenharia de *software,* objetiva desenvolver sistemas com melhor qualidade, a custa de menor tempo e esforço. Estes requisitos são necessários na construção de qualquer programa. Usando Padrões de Projeto, constroem-se *frameworks* que favoreçam o usuário no reuso de análise, projeto e código (REINALDO, 2003).

Neste trabalho é proposto o uso de apenas ferramentas ligadas ao escopo da pesquisa. Para desenvolvimento das telas será utilizado o *WindowBuilder Pro* combinado com *Binding* e como base de dados será usado o *MySql* sendo que para manipulação da mesma será empregado o *MySql Workbench*.

Para desenvolvimento dos relatórios será usada a última versão disponível o *iReport*.

Também será usado o Eclipse combinado com *Java Persistence API* (JPA) juntamente com *EclipseLink* para persistir e manipular os dados.

Com a finalidade de demonstrar o uso do *framework*, serão aplicadas as ferramentas em um protótipo simples que armazenará dados básicos de pessoas.

Com isso, o trabalho buscará expor como usar de forma produtiva para desenvolvimento de aplicações *Java* para *desktop* o Eclipse com agregação de componentes.

1.1 OBJETIVO GERAL

Propor uma combinação de *frameworks* para agilizar o desenvolvimento Java para *desktop*.

1.2 OBJETIVOS ESPECÍFICOS

- Levantar ferramentas necessárias para otimizar o Eclipse para desenvolvimento *desktop*;
- Configurar o ambiente com as ferramentas para desenvolvimento em *Java*; e
- Demonstrar a aplicação das ferramentas do *framework*.

1.3 JUSTIFICATIVA

A pesquisa tem como tema a integração de *framework* para uso de forma produtiva, pois no atual contexto de mercado, existem diversos fatores que levam ao desenvolvimento de *software* com maior agilidade e reuso de objetos. O uso de um *framework* pode auxiliar significativamente na redução de custos e tempo de entrega do produto final.

A necessidade de um *framework* produtivo para desenvolvimento é uma realidade cada vez mais presente. O desenvolvedor precisa otimizar seu trabalho para que seja possível concentrar-se nas regras de negócio, não dedicando tempo, por exemplo, para resolver problemas relacionados ao ambiente de desenvolvimento, como a integração de ferramentas, atualização de telas, criação de objetos entre outros.

Um fator de grande importância na área de desenvolvimento de sistemas é a produtividade. Os desenvolvedores já se preocuparam muito em proteger seus códigos fonte, mas agora existem várias metodologias de desenvolvimento de

sistemas que pregam que o mais importante é conseguir entregar um *software* que atenda todas as funcionalidades requisitadas pelo cliente num prazo menor sem detrimento da qualidade (JUNIOR, 2007).

Essa exposição demonstra uma das preocupações no desenvolvimento de um sistema informatizado, que é a agilidade na entrega do produto final com a qualidade esperada. Para auxiliar, pode ser empregada a reutilização de componentes.

A reutilização de objetos surgiu no final dos anos 60 como uma alternativa para superar a crise do *software*, e tem como objetivo principal desenvolver sistemas com qualidade e economicamente viável utilizando artefatos já especificados, implementados e testados (CARROMEU, 2007).

Um dos principais desafios da Engenharia de *Software* é criar mecanismos para que o processo de desenvolvimento se dê com qualidade e produtividade. Isso pode ser minimizado com um *framework* adequado (BERTOLLO, RUY, *et al.*, 2011).

Para cada situação pode existir um *framework* específico especializado para resolver problemas de forma pontual ou então ser mais generalista e servir de base para integração de outros para atender as necessidades de desenvolvimento.

São várias as justificativas destacando-se entre as principais a necessidade de se otimizar a produtividade durante o desenvolvimento de *software*, devido à grande competitividade existente no mercado, onde as empresas requisitam aos programadores soluções sempre urgentes, sem que isso afete a qualidade dos *softwares* requisitados (JUNIOR, 2007).

A opção por Java nesse trabalho deve-se ao fato de ser uma plataforma bem difundida com uma infinidade de documentação disponível além da liberdade em desenvolver para diversos sistemas operacionais. O Java é atualmente a linguagem mais utilizada em todo o mundo, e ainda em crescimento nas empresas, através de novas adoções. Uma coisa que se deve frisar é que hoje o Java não é apenas uma linguagem, mas sim uma plataforma de desenvolvimento. Permite desenvolver em qualquer sistema operacional para qualquer sistema operacional e oferece um grande número de *frameworks* (SANTANA, 2011).

A aplicação do *framework* será feito apenas para *desktop* para que o escopo definido seja passível de execução no tempo proposto. Também foi levado em conta o conhecimento do autor em relação a outras plataformas e linguagens, sendo este um dos pontos mais proeminentes.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em cinco capítulos. No capítulo 1, será tratado sobre os objetivos, justificativa do trabalho e sua organização. No capítulo 2, será feita a fundamentação teórica sobre produtividade na criação de sistemas, técnicas de desenvolvimento produtivo, métricas para acompanhamento e aspectos gerais das ferramentas que serão usadas. No capítulo 3, será feito uso das ferramentas para que seja possível coletar e analisar os dados. No capítulo 4, será discutido o resultado obtido. Por fim, o capítulo 5 apresentará as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 A PRODUTIVIDADE EM DESENVOLVIMENTO DE SISTEMAS

Ao se falar em desenvolvimento, produção, industrialização e outros termos afins, surge a necessidade de obter resultados com o menor custo e tempo. Agregado a isso está a exigência de qualidade do produto final. Na área de desenvolvimento de sistemas informatizados não é diferente. O cliente sempre espera que o produto seja entregue no menor tempo possível e com a qualidade esperada. Para isso ele está disponível a fazer um determinado investimento que deve ser condizente com o benefício da solução que lhe será entregue.

A produtividade é a melhor determinante da competitividade e lucratividade das indústrias. É, portanto, o melhor indicador do progresso econômico de uma empresa. Através do crescimento continuado da produtividade, tem-se melhor competitividade, os produtos serão melhores e mais baratos, serviços serão bem prestados, os salários aumentarão e o nível de vida será melhor (BARBARÁN e FRANCISCHINI, 2011).

2.2 TÉCNICAS PARA DESENVOLVIMENTO PRODUTIVO

É necessário estabelecer padrões que levem a equipe a atingir os resultados esperados nos projetos desenvolvidos.

Uma melhoria no desenvolvimento de *softwares* pode ser atingida aplicando *Process Patterns* que é uma solução consolidada para organizar um conjunto de atividades para resolver um problema comum na área de Engenharia de Processos. Processos atuam sobre domínios bastante distintos, como projetos de engenharia, produção industrial, recrutamento de funcionários, entre outros, sendo que cada um apresenta necessidades e regras específicas. Por esta razão, os processos apresentam diferentes características em cada domínio em que são aplicados.

Entretanto, é possível notar que existem similaridades entre processos de domínios distintos (TAMAKI e HIRAMA, 2007).

Como pode ser visto, para cada caso pode existir uma exceção que deve ser tratada em relação a um projeto anterior. Mesmo sendo do mesmo domínio, um projeto pode necessitar de um tratamento específico.

Para auxiliar na padronização de processo existem sistemas como, por exemplo, o *Rational Unified Process* (RUP), uma estrutura de processo abrangente, que fornece as práticas da industrialização, teste e entrega de *software*, contribuindo para implementação e gestão eficaz do projeto. Também oferece orientação de melhores práticas, sendo adequado ao desenvolvimento de parte ou do projeto todo (INTERNATIONAL BUSINESS MACHINES, 2011).

No RUP o projeto de *software* é dividido ao longo do tempo em quatro fases sendo elas a iniciação, elaboração, construção e transição. Cada uma das fases é dividida em uma ou mais iterações. Uma iteração é um ciclo de desenvolvimento completo, resultando em uma versão, um subconjunto do produto final, que cresce incrementalmente de iteração a iteração para se transformar no produto final (TAMAKI e HIRAMA, 2007).

Além dessa divisão que auxilia do processo de desenvolvimento, o RUP representa o processo utilizando cinco elementos, sendo eles os (1) papéis, um conjunto de comportamentos ou responsabilidades que um indivíduo pode ter durante o processo de desenvolvimento, as (2) atividades, algo que um papel pode realizar e que produz um resultado significativo para o projeto, os (3) artefatos, um produto de trabalho do processo que pode ser utilizado ou produzido por um papel ao realizar uma atividade, (4) fluxos de trabalho, uma seqüência de atividades que produz um resultado de valor observável e as (5) disciplinas um conjunto de atividades relacionadas a uma mesma área de interesse (TAMAKI e HIRAMA, 2007).

Os padrões de processos são uma das ferramentas para obter um desenvolvimento produtivo pois com isso o engenheiro de *software* consegue estabelecer caminhos a serem seguidos pela equipe afim de que não se percam durante os processos.

2.3 MÉTRICAS PARA ACOMPANHAMENTO DO DESENVOLVIMENTO

O acompanhamento durante o desenvolvimento de um sistema é algo de estrema relevância. A empresa ou desenvolvedor precisa ter estimativas de quando será possível entregar um produto. Também deve conhecer quantas horas levará para concluir determinado projeto para poder criar orçamentos condizentes. Caso o valor orçado seja alto poderá perder o cliente. Ao mesmo tempo se for muito inferior a realidade, poderá inviabilizar o projeto.

Os modelos para estimativa de esforço para desenvolvimento de *software* são divididos em duas categorias principais sendo elas a algorítmicos e não-algorítmica (ATTARZADEH e OW, 2009).

Maior parte do trabalho no campo estimativa de custos está focada na modelagem de algoritmos para levantamentos de indicadores. Neste processo os custos são analisados utilizando fórmulas matemáticas que liga os custos ou insumos com métricas para produzir uma saída estimada. As fórmulas usadas em um modelo formal surgem a partir da análise de dados históricos. A precisão do modelo pode ser melhorada através de ajustes ao modelo para seu ambiente de desenvolvimento específico, que envolve basicamente a correção dos coeficientes das métricas. Geralmente existe uma grande inconsistência das estimativas que pode ser reduzida com o ajuste do modelo mas, que mesmo assim continuam a produzir erros de 50-100% (HACETTEPE UNIVERSITY, 1998).

No Brasil, as indústrias de sistemas informatizados sofrem a denominada crise do *software,* enfrentando alguns problemas como estimativas de prazo e de custo freqüentemente imprecisas, falta de investimento em recursos humanos na busca de melhor produtividade o que acaba reduzindo a qualidade do *software* que, apresentam altos índices de erros nos programas, causando insatisfação e desconfiança por parte do cliente e por fim a ausência de medidas consistentes aos objetivos e à estratégia competitiva da organização, o que dificulta a monitoração das atividades e a melhoria, tanto no processo, quanto no produto (BARBARÁN e FRANCISCHINI, 2011).

A capacitação de recursos humanos, não só para desenvolvimento, mas também para gerenciamento, é uma necessidade para aquelas empresas que trabalham com desenvolvimento de *softwares.* Uma base deve ser criada para servir

de referência para acompanhamento, sendo a mesma atualizada com base nas métricas obtidas a cada novo projeto. Algumas das opções para estabelecer essas referências, são as métricas de *softwares*.

2.3.1 Linha de Código (LOC)

Essa métrica mede um sistema por meio do número de linhas do código fonte do programa. Dessa forma tenta-se prever a quantidade de esforço necessário para produzir um sistema e o custo de desenvolvimento de um programa por linha de código.

Esta medida tem apresentado algumas controvérsias, não sendo universalmente aceita como a melhor maneira de se medir o processo de desenvolvimento do *software*. A maior parte gira em torno de que as medidas de LOC são dependentes da linguagem de programação, penalizando portanto, programas bem projetados que tem redução na quantidade de linhas. Outro fator é que o seu uso em estimativas requer um nível de detalhes que pode ser difícil de obter (BARBARÁN e FRANCISCHINI, 2011).

Como exemplo pode se citar um caso onde o programador tenha vasta experiência em programação orientada a objetos e faça uso de herança. De outro lado pode existir um caso onde esse benefício da programação orientada a objetos não seja empregado. Isso vai impactar diretamente no custo do sistema. A questão é se esta variação deve ou não ser empregada também no preço do produto e de quem seria a responsabilidade em absorver a diferença.

2.3.2 Pontos de Função (FP)

O Ponto de Função mede o tamanho do *software* pela quantidade de suas funcionalidades externas, tomando como base o projeto lógico ou a partir do modelo de dados (BARBARÁN e FRANCISCHINI, 2011).

A idéia fundamental da análise por ponto de função consiste em medir o tamanho do produto de *software* baseado em termos lógicos, orientados ao usuário. A análise por ponto de função não se preocupa diretamente com a plataforma tecnológica, ferramentas de desenvolvimento e linhas de código, medindo simplesmente a funcionalidade entregue ao usuário final. Ela avalia e mede o produto baseando-se em características funcionais bem definidas deste sistema (MACEDO, 2003).

2.3.3 Modelo de Custo Construtivo (COCOMO)

Este método foi desenvolvido por Barry Boehm, para estimar esforço, prazo, custo e tamanho da equipe para um projeto de *software.* Ele trabalha com três modelos sendo o COCOMO básico, intermediário e o avançado (BARBARÁN e FRANCISCHINI, 2011).

O COCOMO básico é um modelo estático de valor simples que calcula o esforço e o custo de desenvolvimento de *software* como uma função do tamanho de programa expresso em estimativas das linhas de código. O intermediário se preocupa com o esforço para desenvolvimento do *software* como uma função do tamanho do programa e de um conjunto de direcionadores de custo que incluem avaliações subjetivas do produto, do *hardware*, do pessoal e dos atributos do projeto. Por fim, o avançado incorpora todas as características do intermediário, com uma avaliação do impacto dos direcionadores de custo sobre cada passo do processo de engenharia de *software* como por exemplo a análise e o projeto (BARBARÁN e FRANCISCHINI, 2011).

Como a maioria das técnicas tradicionais, COCOMO exige um processo de estimativa a longo prazo (ATTARZADEH e OW, 2009) o que em alguns casos pode desestimular seu uso, levando os engenheiros de *software* a buscar outras soluções.

2.4 ASPECTOS GERAIS DO ECLIPSE E OUTRAS FERRAMENTAS

Os recursos e aspectos gerais do Java, Eclipse e outras ferramentas destacados aqui, são os relevantes para o estudo sendo eles o *MarketPlace, Data Tools Platform*(DTP), EclipseLink, W*indowBuilderPro*, *BeansBinding*, MySql e iReport.

2.4.1 Histórico do Eclipse e sua Comunidade

Muito conhecido no mundo de desenvolvimento o Eclipse é uma ferramenta utilizada para otimizar a produção de *softwares.* Além de uma gama de componentes já disponibilizados na sua distribuição original é possível realizar a expansão de seus recursos por meio de complementos.

Eclipse é uma comunidade de fonte aberta, cujos projetos estão focados na construção de uma plataforma de desenvolvimento aberta composta por *frameworks* extensíveis, ferramentas em tempo de execução para a construção, implementação e gerenciamento de *software* em todo o ciclo de vida. A comunidade Eclipse tem uma reputação merecida de fornecer *software* com qualidade de forma confiável e previsível. Isto se deve ao empenho do *committers* que são os desenvolvedores de código aberto e organizações que contribuem para os projetos *open source*. A Fundação Eclipse também fornece serviços e suporte para os projetos para ajudá-los alcançar estas metas (THE ECLIPSE FOUNDATION, 2011).

A história do Eclipse começou com indústrias lideres como Borland, IBM, MERANT, QNX Software Systems, Rational Software, RedHat, SuSE, Together Softe Webgain formado o Conselho de Regentes eclipse.org inicial em novembro de 2001. Até o final de 2003, este consórcio inicial tinha crescido para mais de 80 membros. Em 2 de fevereiro de 2004 o Conselho de Regentes Eclipse, anunciou a reorganização do Eclipse em uma corporação sem fins lucrativos. Originalmente um consórcio que se formou quando a IBM lançou a Plataforma Eclipse *Open Source*, tornando-se um órgão independente que iria conduzir a evolução da plataforma com vantagem para os *committers* e usuários finais. Toda a tecnologia e código fonte

fornecido e desenvolvido por esta comunidade em rápido crescimento passou a ser disponibilizada a título gratuito, através do *Eclipse Public License* (THE ECLIPSE FOUNDATION, 2011).

2.4.2 MarketPlace

O cliente *MarketPlace* permite consultar e instalar aplicações a partir do Eclipse *Marketplace* facilitando o controle de *plug-ins* (BLEWITT, 2010).

Com essa ferramenta, o usuário que estiver iniciando o uso do Eclipse terá facilidades em gerenciar e instalar novos *plug-ins* o que propiciará uma maior gama de ferramentas para experimentar e testar em seus projetos. Com isso o dinamismo e a produtividade são fortalecidos.

2.4.3 Data Tools Platform (DTP)

O *Data Tools Platform* (DTP) apresenta um conjunto bem organizado de *frameworks* e ferramentas para criação, gerenciamento e utilização de dados. Essas ferramentas são destinados ao uso em desenvolvimento e tarefas administrativas. O DTP consiste em vários projetos, incluindo um modelo de base, capacitação, conectividade e ferramentas de desenvolvimento *Structured Query Language* (SQL). Esses projetos oferecem os modelos de domínio, núcleo do *frameworks* DTP, ferramentas para fontes específicas de dados, conexão de gerenciamento de funcionalidade e ferramentas de consulta necessários para a construção, acesso e manipular dados (THE ECLIPSE FOUNDATION, 2010).

2.4.4 EclipseLink

O *EclipseLink* é um *framework* de persistência e transformação avançada de objeto que fornece ferramentas em tempo de desenvolvimento e execução que implementa *Java Persistence API* (JPA) e reduz os esforços de desenvolvimento e manutenção aumentando a funcionalidade do aplicativo da empresa. Permite integrar persistência e transformação do objeto na aplicação, mantendo o desenvolvedor focado no problema de domínio primário, tirando partido de uma solução eficiente, flexível e comprovada em campo (THE ECLIPSE FOUNDATION, 2011).

O objetivo do projeto *EclipseLink* é fornecer uma solução completa para persistência de forma abrangente. Também é desenvolvido para ser executado em qualquer ambiente Java, permitindo ler e escrever objetos para qualquer tipo de fonte de dados, incluindo bancos de dados relacionais, *eXtensible Markup Language* (XML), ou sistemas EIS. São incluídas as extensões de funcionalidades avançadas, para os padrões dominantes de persistência para cada fonte de dados de destino como JPA para bancos de dados relacionais, Java API *for* XML *Binding* (JAXB) para XML, *Java Connector Architecture* (JCA) para EIS e outros tipos de sistemas legados e *Service Data Objects* (SDO) (THE ECLIPSE FOUNDATION, 2011).

2.4.5 WindowBuilder Pro

O *WindowBuilder* é composto por SWT *Designer* e Swing *Designer* tornando a criação de aplicações Java GUI descomplicada, reduzindo com isso o tempo empregado para escrever códigos. É possível usar ferramentas visuais de *design* e *layout* para criar de formas simples, janelas complexas, sendo o código Java gerado automaticamente. Os controles podem ser arrastados e soltos para serem adicionados. Também é possível gerenciar os manipuladores de eventos dos componentes e alterar suas propriedades usando um editor visual entre outras opções (THE ECLIPSE FOUNDATION, 2011).

Essa ferramenta é construído como um *plug-in* para o Eclipse e as várias ferramentas baseadas em Eclipse. Ele constrói uma árvore sintática abstrata (AST) para navegar no código fonte e usa GEF para mostrar e gerir a apresentação visual. O código gerado não requer bibliotecas adicionais personalizados para compilar e executar, podendo ser usado sem ter *WindowBuilder Pro* instalado. Ele pode ler e escrever praticamente qualquer formato e engenharia reversa inclusive código escrito a mão, suportando a edição do código de forma livre permitindo fazer alterações em qualquer lugar e mais, é possível mover, alterar nome e subdividir os métodos sem nenhum problema (THE ECLIPSE FOUNDATION, 2011).

2.4.6 BeansBinding

Em tradução literal, *binding* pode significar ligação, amarração, vinculação entre outras palavras. Esses são significados acertados com a lógica de seu funcionamento, pois o mesmo tem a função de estabelecer vinculações de componentes.

É uma biblioteca da plataforma Java definida pela *Java Specification Request* (JSR) 295. Este projeto *open source* tenta simplificar a forma como os desenvolvedores sincronizam duas propriedades, geralmente em dois objetos diferentes (O'CONNER, 2008).

Grandes quantidades do tempo de desenvolvimento das aplicações Java com interface gráfica do usuário (GUI) envolve mover dados de objetos do domínio para componentes GUI e vice-versa (DELAP, 2006). Os *frameworks* que implementam *binding* buscam minimizar esse impacto.

A especificação de *JavaBeans* define um modelo para componentes de *software* reutilizáveis. Entre muitas outras coisas, ela especifica como os componentes podem expor suas mudanças de estados e como outros objetos interessados nessas mudanças serão informados quando elas ocorrerem. A idéia subjacente foi a de construir uma aplicação visualmente organizada e modificada por *beans*. Conseqüentemente, componentes *Swing*, sendo um dos blocos de construção de um programa desse tipo, deveriam se comportar adequadamente como *beans* (KÜNNETH, 2008).

Com as exposições, percebe-se que a finalidade principal do *binding* seria auxiliar no desenvolvimento de aplicações, controlando automaticamente as atualizações de estado dos componentes.

2.4.7 Mysql

O MySQL oferece um *software*, multi-threaded, multiusuário e servidor de banco de dados baseado em *Structured Query Language* (SQL). Destina-se a missão crítica, sistemas com carga pesada de produção, bem como para a incorporação de *software* implantado em massa. Atualmente é mantido pela Oracle Corporation e suas afiliadas. O *software* MySQL é de licença dupla, com isso os usuários podem optar por usar o *software* MySQL como um produto de código aberto sob os termos da *General Public License* (GNU) ou pode comprar uma licença comercial padrão da Oracle (ORACLE AND/OR ITS AFFILIATES, 2011).

Quanto a conectividade, existem clientes que podem se conectar usando vários protocolos. O *MySQL Connector* /J que é o cliente para Java, fornece suporte para programas que usam conexões JDBC. Os clientes podem ser executados no Windows ou Unix (ORACLE AND/OR ITS AFFILIATES, 2011).

A partir do MySql 5.5, o mecanismo padrão de armazenamento do banco de dados passou do MyISAM para o InnoDB. Esse mecanismo proporciona transações ACID, integridade referencial, recuperação de falhas do sistema e trabalha com controle sobre registros e não tabela. Para registro constantemente acessados é estabelecido um *cache* para que sejam acessados diretamente em memória (ORACLE AND/OR ITS AFFILIATES, 2011).

2.4.8 IReport

Um dos requisitos comuns em um sistema são os relatórios. *iReport* é um aplicativo que permite o desenvolvimento de relatórios oferecendo uma interface gráfica intuitiva que facilita a criação dos mesmos.

iReport é uma ferramenta de código aberto que pode criar relatórios complexos através da Biblioteca *Jasper Reports*. Ele é escrito em Java e é distribuído com o código fonte de acordo com a Licença Pública Geral (GNU). Através de uma interface gráfica intuitiva, iReport permite criar qualquer tipo de relatório facilmente. Com a versão 3.1, iReport foi quase totalmente reescrito, com a nova aplicação baseada na plataforma NetBeans (TOFFOLI, 2011).

3 PROCEDIMENTOS METODOLÓGICOS DA PESQUISA

Para realização do estudo foi empregado o uso de algumas ferramentas já disponíveis no mercado para desenvolvimento de sistemas e linguagem Java, sendo realizado um breve estudo sobre cada uma delas.

3.1 BANCO DE DADOS

A coleta de dados foi feita com base na analise dos resultados obtidos com o uso das ferramentas. Nas subseções seguintes foi demonstrado por meio de recortes de telas algumas das principais funcionalidades dos sistemas.

A criação do banco de dados pode ser feito usando uma ferramenta visual para edição ou pelo console por meio da digitação das linhas de código. Mesmo na ferramenta é possível editar o código manualmente e executar o mesmo para criação dos objetos da base de dados. Neste trabalho foi usado o MySql Workbench 5.2.34 CE que permite criar a base, adicionar as tabelas e fazer a manutenção. É disponibilizado um editor para a tabela que permite gerenciar todos os recursos por meio de abas de forma facilitada.

Com o *extendedentity-relationship* (EER) ou modelo de entidade e relacionamento oferecido pelo *framework*, foi possível criar a tabela os relacionamentos e as chaves estrangeiras de forma visual.

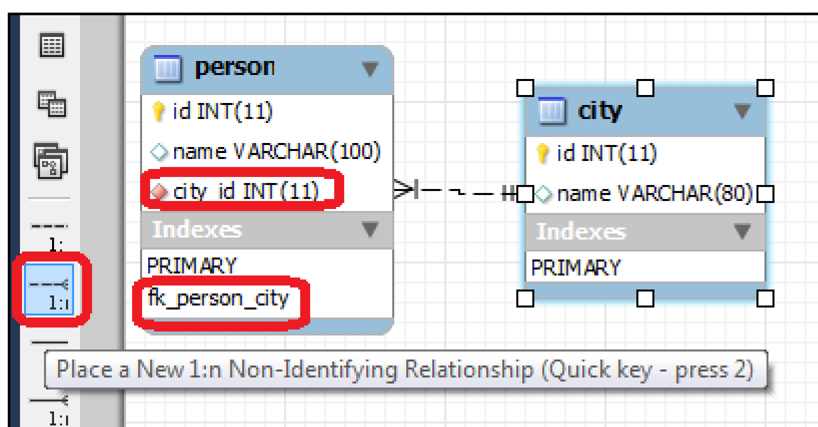

Figura 1 - Criando relacionamento entre tabelas

Na Figura 1 é exibido parte da tela para criação de diagramas. Nela estão marcados alguns itens de relevância como a chave estrangeira da tabela *person, city_id.* Esse campo foi criado automaticamente ao ser estabelecido o relacionamento de 1 para n entre as duas entidades. As alterações podem ser replicadas na base de dados criada conforme Figura 2.

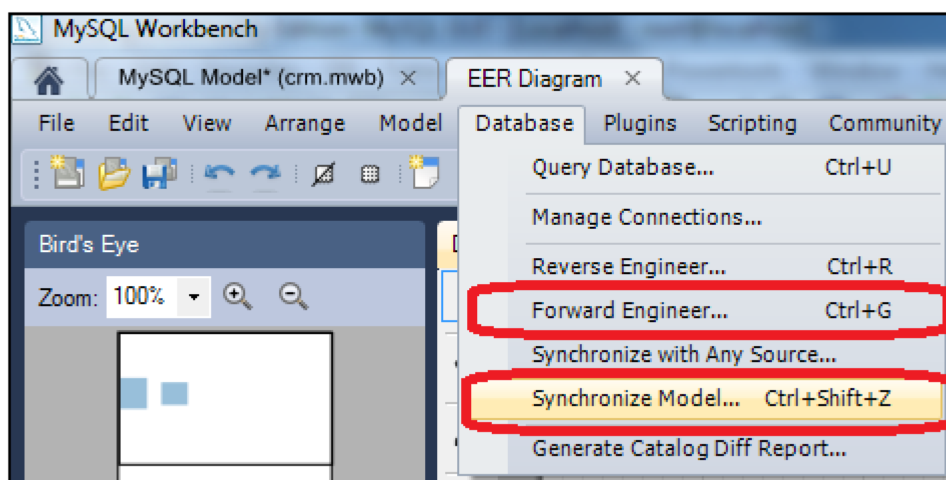

Figura 2 - Sincronizando o modelo de dados

Neste caso, onde a tabela já existe, a opção a ser usada é a da sincronização, caso a tabela também estivesse sendo criada pela tela de diagramação da base, a opção a ser usada seria de *Forward Enginner.*

3.2 ENTIDADES

A criação da entidade foi feita de forma simplifica dentro do Eclipse Indigo. Como nessa pesquisa foi usado EclipseLink, o mesmo foi obtido por meio de *download* e habilitado dentro do próprio Eclipse. A Figura 3 demonstra como essa configuração foi feita.

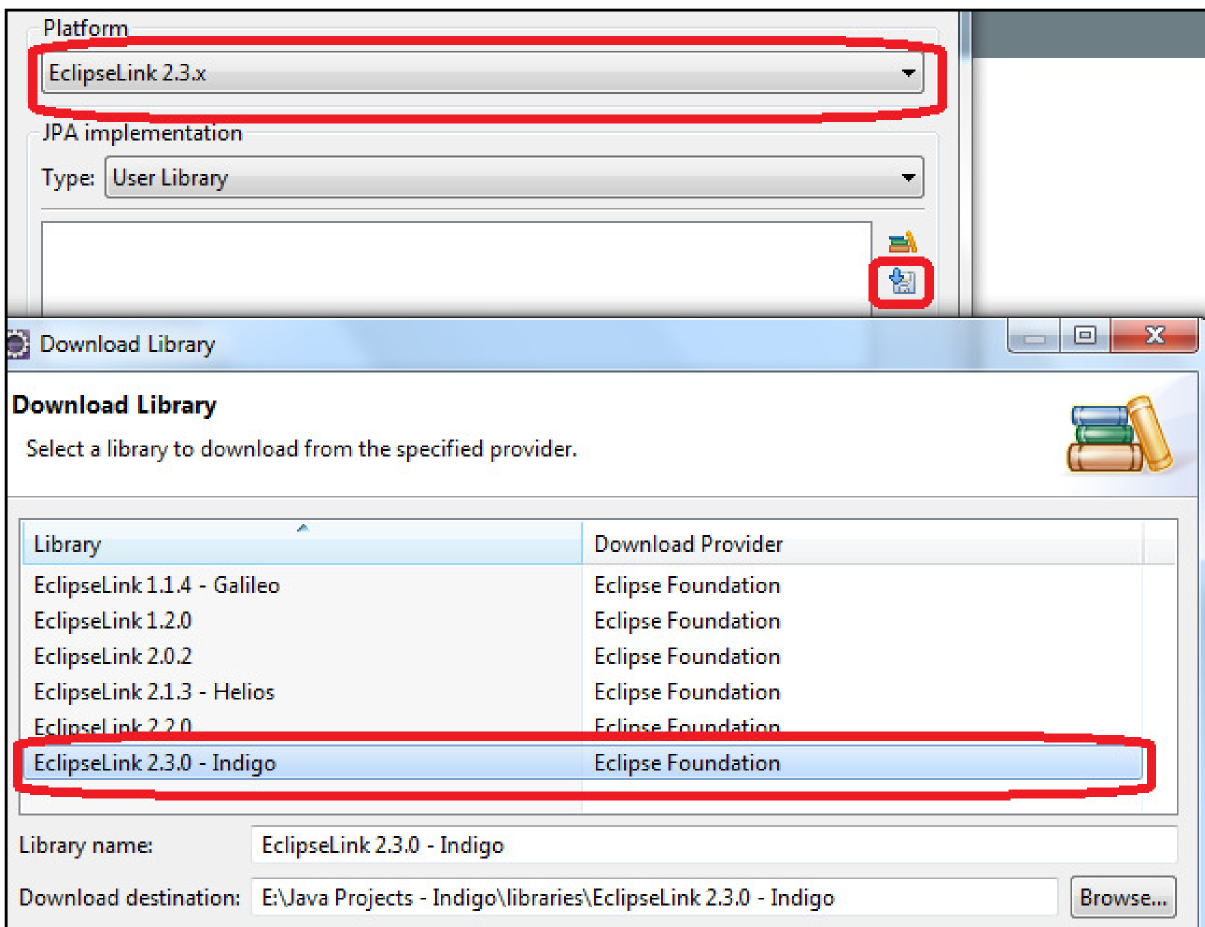

Figura 3 - Baixando e configurando EclipseLink

Nesta mesma tela foi definida a conexão com o banco de dados, um *Drive* de conexão, o MySQL Connector/J e adicionado o *jar* ao *build path*.

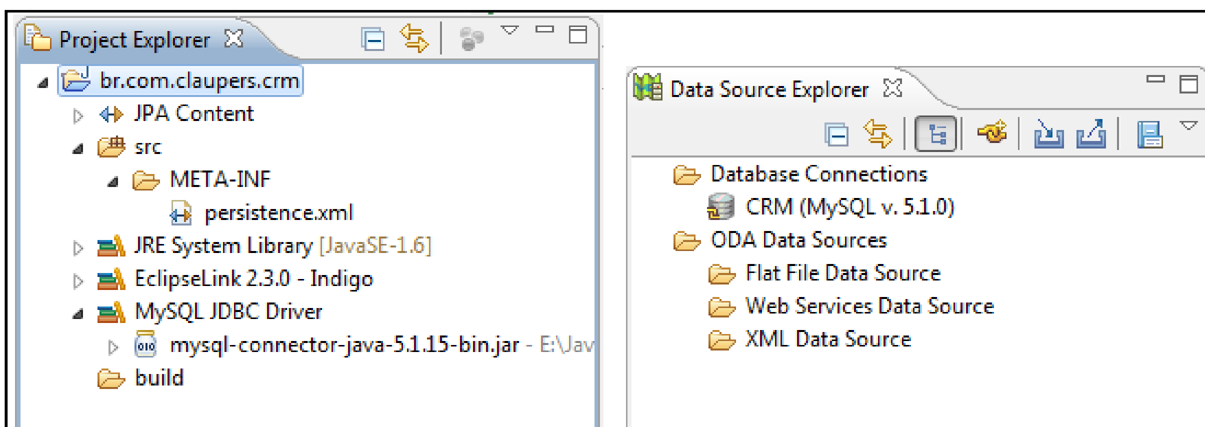

Figura 4 - Project Explorer e Data Source Explorer após criação do projeto

A Figura 4 mostra o menu de navegação do projeto e também da fonte de dados. Através do *Data Source Explorer* é possível inserir e visualizar dados na base caso necessário.

3.3 TELAS E VÍNCULOS *BINDING*

3.3.1 Instalando os *Frameworks*

Para criação de telas e vinculação foi usado o WindowBuilder Pro, instalado com auxílio *MarketPlace* do Eclispe sendo que para isso bastou buscar pela palavra chave *WindowBuilder* que o *MarketPlace* retornado o resultado para iniciar a instalação do *framework.*

Diferente de outros *plugins* que tem a instalação iniciada diretamente no *marketPlace,* o WindowBuilder Pro forneceu uma *url* para colar no instalador do Eclipse e dar seqüência ao processo.

3.3.2 Tela Principal

Com a instalação do *WindowBuilder Pro, o* passo seguinte foi criar uma janela para a aplicação usando os recursos do *framework* através de novos botões que ficaram disponíveis na tela inicial do Eclipse.

Com a criação do formulário principal o Eclipse exibiu o código fonte do mesmo. Para trabalhar em modo visual, bastou clicar na aba *design* conforme é exibido na Figura 5 que também já traz mais alguns recortes da tela de desenvolvimento.

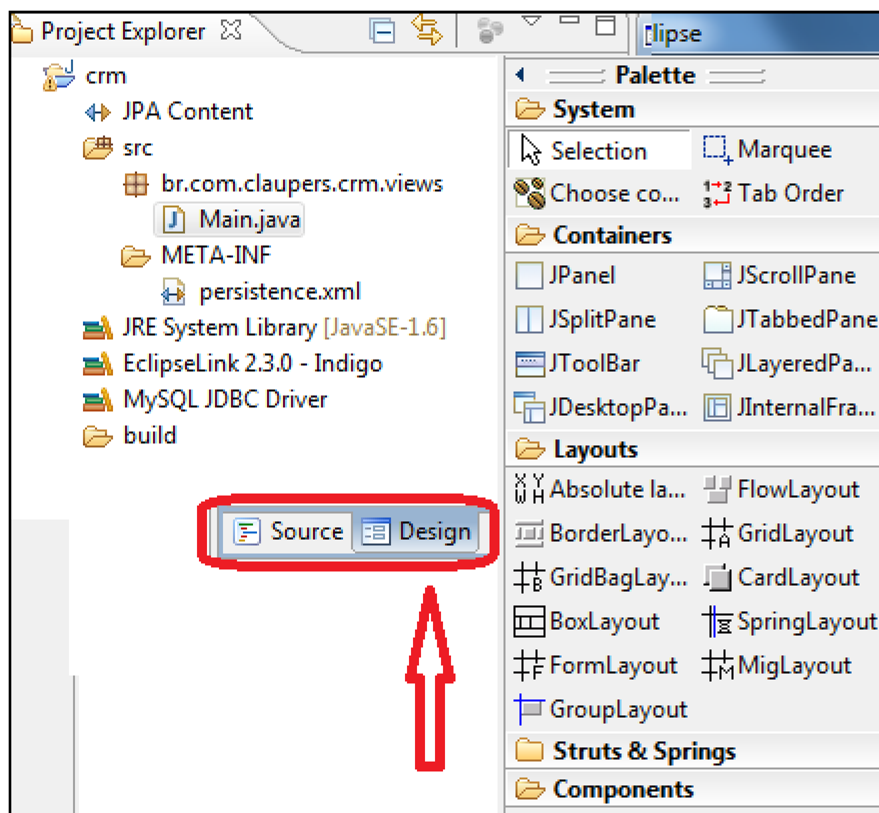

Figura 5 - Botão de navegação entre código e formulário mais algumas ferramentas do WindowBuilder Pro

O Eclipse permite varias combinações de janelas para facilitar o uso através das Perspectivas, que está ligado a forma que se vê o ambiente de desenvolvimento do projeto. Quando se trabalha com código Java, pode se optar por Java EE *perspective*, para base de dados *Database development perspective* e assim por diante.

Para criar o formulário, bastou selecionar os componentes com um clique na paleta e depois com outro clique colocar no formulário. Na Figura 6 é exibido como selecionar o *layout* dos componentes.

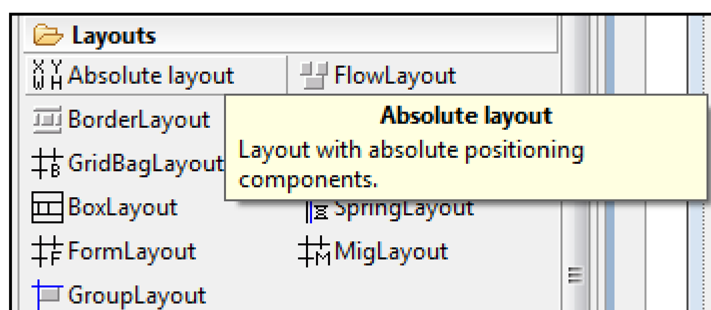

Figura 6 - Definindo *Absolute layout*

Ao adicionar, por exemplo, um JTextField, a paleta *structure* exibia as opções que podiam ser alteradas do componente. Em sua parte superior a mesma tem cinco botões sendo eles o (1) *Show Events* que exibe os eventos que podem ser ativados, (2) *Goto Definition* que leva para o código do componente, (3) *Convert Field to local/Convert Local to Field* que define se os componentes serão declarados para serem usados apenas localmente no código ou se em todo o escopo da classe, (4) *Show advanced properties* que exibe as opções avançadas do componente e por fim o botão (5) *Restore default value* que restaura os valores padrões.

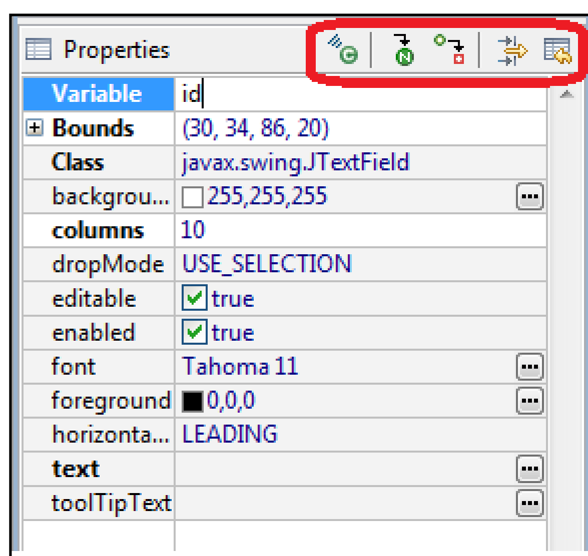

Figura 7 - Paleta *Structure* e suas opções

Na Figura 8 é exibida a prévia da janela criada com WindowBuilder Pro e do lado esquerdo todos seus componentes. Feito isso, foi possível iniciar a vinculação com *binding.*

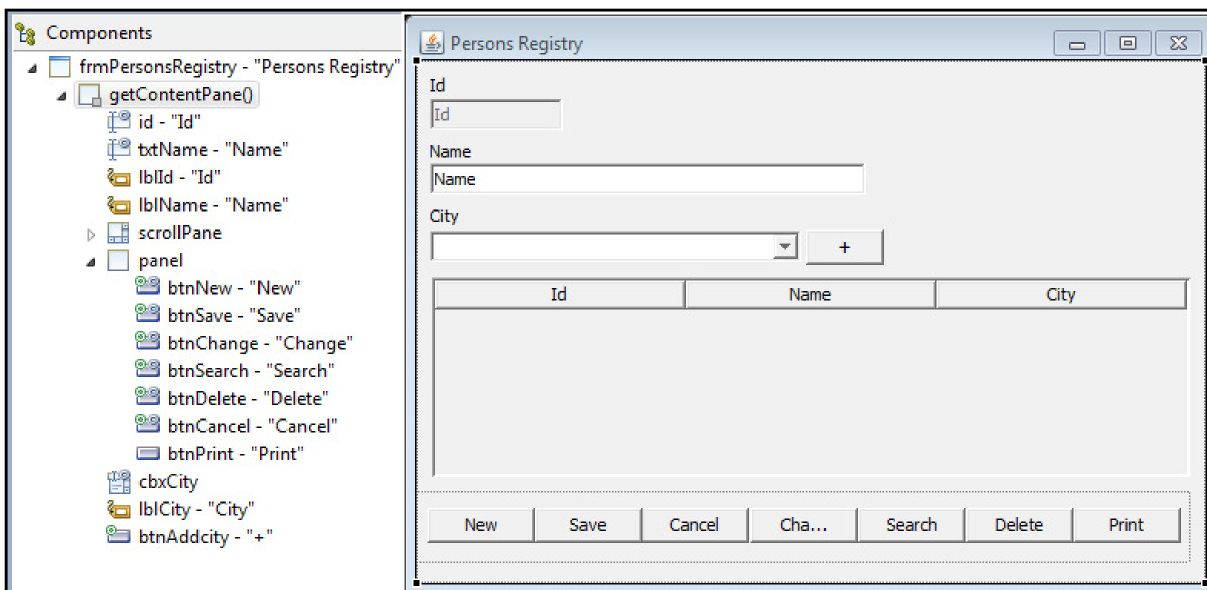

Figura 8 - Tela para cadastro de pessoas criada com WindowBuilder Pro

3.3.3 Criação das Entidades

Como a base de dados já havia sido criada, foi possível gerar as entidades de forma reversa conforme exibido na Figura 9.

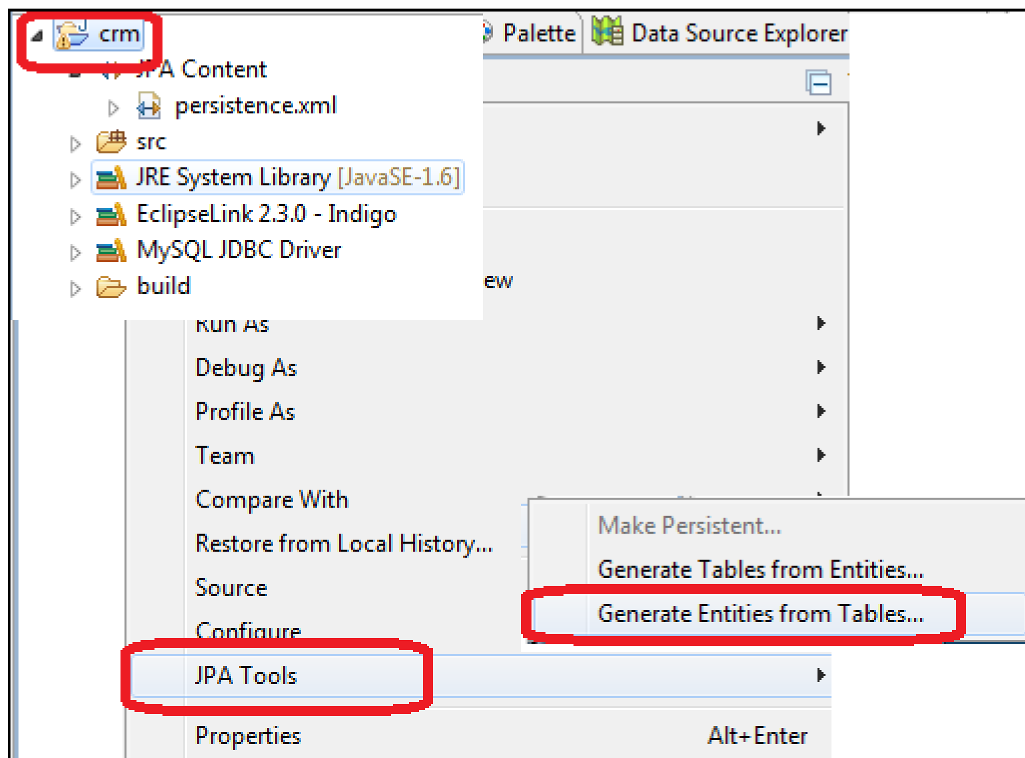

Figura 9 - Criando entidades de forma reversa

Com isso foi chamada a ferramenta de geração de entidades que permitiu controlar a conexão com o banco, quais tabelas teriam as entidades criadas, os relacionamentos das mesmas, o tipo de acesso, a forma de geração da chave primária e tipo de coleção como é visto na Figura 10.

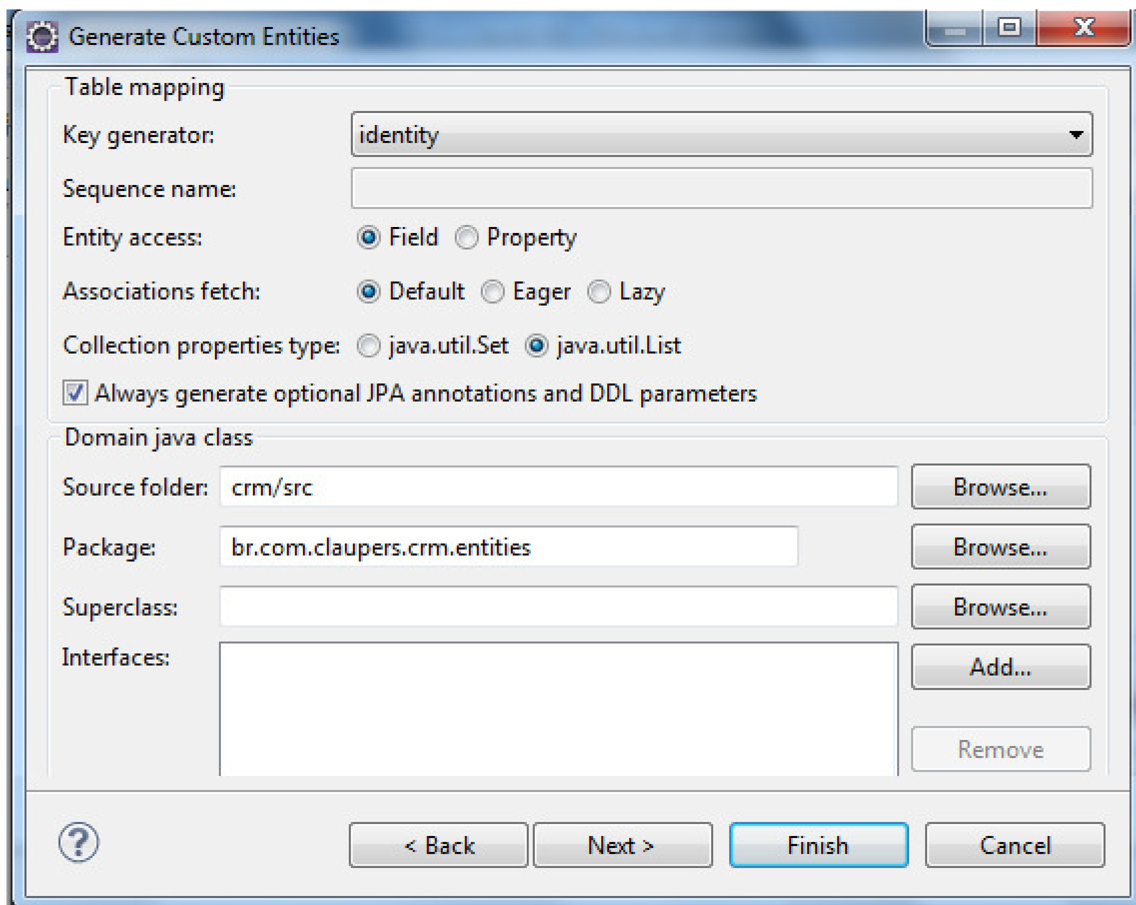

Figura 10 - Gerenciando criação de entidades

Com esse procedimento cria-se as entidades para serem usadas. Para isso, ainda foi necessário fazer alguns ajustes finais no arquivo persistence.xml.

3.3.4 Arquivo *Persistence.xml*

Como foi usado o EclipseLink, o controle do arquivo foi feito de forma visual em vários parâmetros. Com outras opções como Hibernate, isso também é possível, mas alguns parâmetros tem que ser ajustadas manualmente no código do arquivo.

No arquivo *persistense.xml* foi possível controlar qual o nível de registro de eventos do sistema. Esta opção é útil para detecção de erros. A Figura 11 ilustra como.

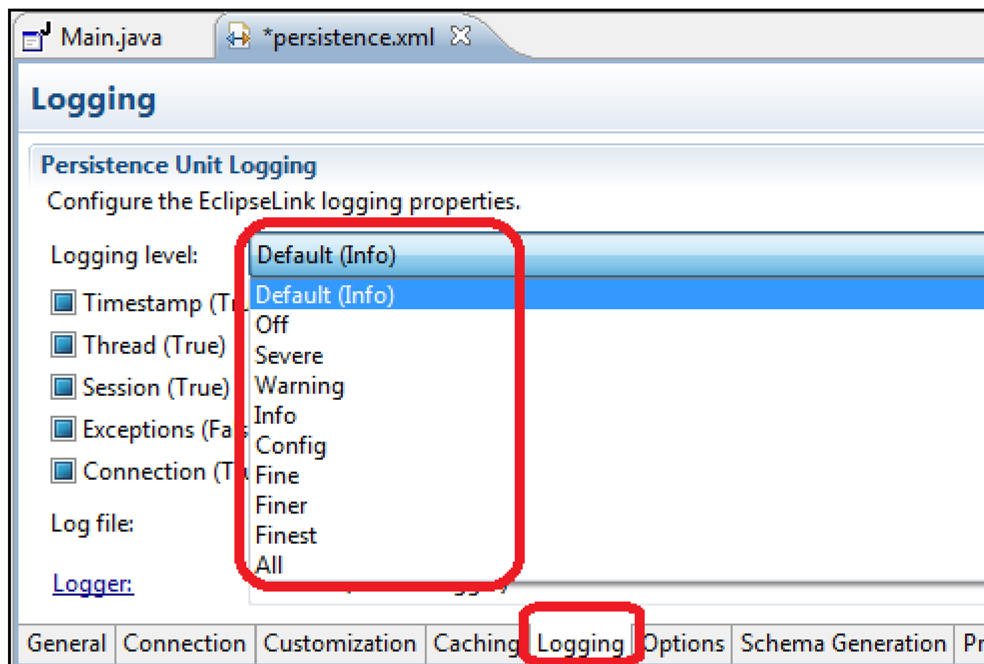

Figura 11 - Configurando registro de eventos em *persistense.xml*

Da mesma forma que gerou-se as entidades usando como referência as tabelas da base de dados, o processo também pode ser feito no outro sentido. Para isso basta criar a entidade e vinculá-la no arquivo *persistense.xml* e configurar na aba *Schema Generation* para que as tabelas sejam criadas ao se executar o sistema.

Toda alteração feita foi refletida no código do arquivo *persistense.xml* que pode ser visualizado na aba *Source*. As alterações feitas no código também refletirão nos controles visuais.

3.3.5 Biblioteca *Beansbinding* e aba *Binding*

Para efetuar a vinculação foi necessário a biblioteca *beansbinding-1.2.1*, por isso foi adicionada ao *CLASSPATH*. Após adicionar o arquivo, bastou fechar e abrir o formulário para que uma nova aba, a *Bindings,* fosse ativada. Nessa aba, ficaram disponíveis vários comandos visuais para configuração dos vínculos como pode ser observado na Figura 12.

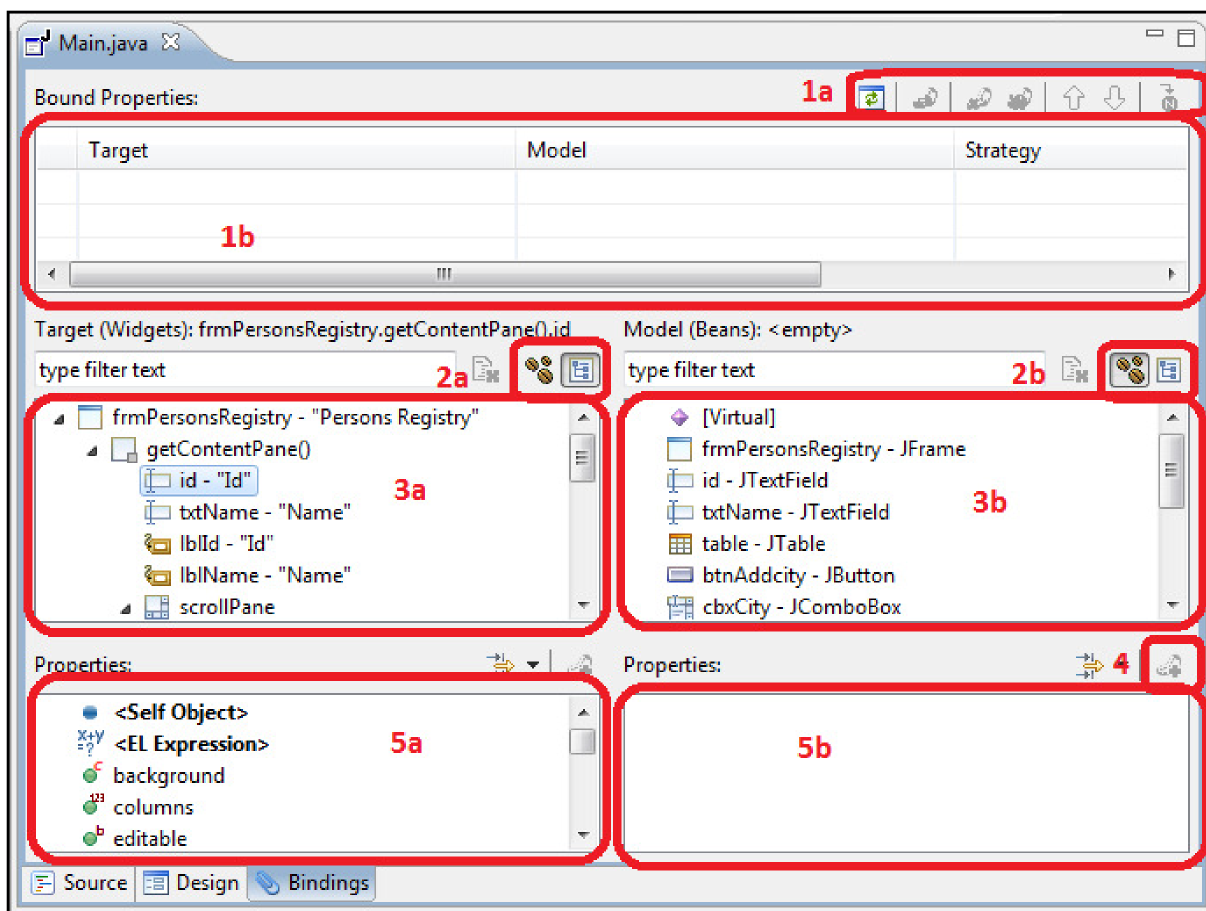

Figura 12 - Tela para configuração das vinculações

O campo 1b, exibe as vinculações criadas que, ao serem selecionadas podem ser alteradas pelos controles do campo 1a. Os controles dos campos 2a e 2b servem para definir como o objeto deve ser tratado, isso é muito importante, pois algumas opções só aparecem se for feita a definição correta. Os campos 3a e 3b estabelecem, respectivamente, qual objeto será o alvo e o modelo. O controle do campo 4 serve para criar a vinculação. Por fim, os campos 5a e 5b, permitem escolher quais propriedades serão vinculadas de ambos os objetos.

3.3.6 Tabela Modelo e o Vinculo ao *JTable*

O primeiro objeto vinculado nesta tela foi o *JTable*. Para isso, criou-se uma tabela modelo conforme código fonte do Quadro 1.

```
package br.com.claupers.crm.model;

import java.beans.PropertyChangeSupport;
import java.util.ArrayList;
import java.util.List;

import javax.swing.table.AbstractTableModel;

import br.com.claupers.crm.entities.City;
import br.com.claupers.crm.entities.Person;

public class PersonTableModel extends AbstractTableModel {

	/**
	 *Table Model from Persons
	 */
	private static final long serialVersionUID = 1L;
	private List<Person>myList = new ArrayList<Person>();

	public PersonTableModel(){
		this.myList = new ArrayList<Person>();
	}

	public PersonTableModel(List<Person> c){
		this();
		myList.addAll(c);
	}

	public void addAll(List<Person> c){
		List<Person> oldValue = this.myList;
		this.myList = new ArrayList<Person>(this.myList);
		this.myList.addAll(c);
		firePropertyChange("myList", oldValue, this.myList);
		firePropertyChange("rowCount", oldValue.size(), this.myList.size());

	}

	public void add(Person e){
		List<Person> oldValue = this.myList;
		this.myList = new ArrayList<Person>(this.myList);
		this.myList.add(e);
		firePropertyChange("myList", oldValue, this.myList);
		firePropertyChange("rowCount", oldValue.size(), this.myList.size());

	}
	public void add(int index,Person e){
		List<Person> oldValue = this.myList;
		this.myList = new ArrayList<Person>(this.myList);
		this.myList.add(index, e);
		firePropertyChange("myList", oldValue, this.myList);
		firePropertyChange("rowCount", oldValue.size(), this.myList.size());

	}

	@Override
	public int getColumnCount() {
		return 3;
	}
```

```
@Override
public int getRowCount() {
	returnmyList.size();
}

@Override
public Object getValueAt(int row, int col) {
	Person e = myList.get(row);
	if (col==0){
		return e.getId();
	}else if (col==1){
		return e.getName();
	}else if (col==2){
		return e.getCity();
	}
	return"";
}

@Override
public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
	Person e = myList.get(rowIndex);
	if (columnIndex==0){
		e.setId((Integer)aValue);
	}else if (columnIndex==1){
		e.setName(aValue.toString());
	}else if (columnIndex==2){
		e.setCity((City)aValue);
	}
	fireTableCellUpdated(rowIndex, columnIndex);
}

@Override
public String getColumnName(int column) {
	if (column==0){
		return"Id";
	}else if (column ==1){
			return"Name";
	}else if (column ==2){
			return"City";
	}
	return super.getColumnName(column);
}



@Override
public boolean isCellEditable(int rowIndex, int columnIndex) {
	return false;
}

public void remove(int index){
	List<Person> oldValue = this.myList;
	this.myList = new ArrayList<Person>(this.myList);
	this.myList.remove(index);
	firePropertyChange("myList", oldValue, this.myList);
	firePropertyChange("rowCount", oldValue.size(), this.myList.size());

}
```

```
	public void remove(Person e){
		List<Person> oldValue = this.myList;
		this.myList = new ArrayList<Person>(this.myList);
		this.myList.remove(e);
		firePropertyChange("myList", oldValue, this.myList);
		firePropertyChange("rowCount", oldValue.size(), this.myList.size());

	}

	@Override
	public Class<?> getColumnClass(int columnIndex) {
		return Object.class;
	}

	@Override
	public void fireTableCellUpdated(int row, int column) {
		super.fireTableCellUpdated(row, column);
	}

	public Person getPerson(int index) {
		return myList.get(index);
	}

	public void clear(){
		List<Person> oldValue = this.myList;
		this.myList = new ArrayList<Person>(this.myList);
		this.myList.clear();
		firePropertyChange("myList", oldValue, this.myList);
		firePropertyChange("rowCount", oldValue.size(), this.myList.size());
	}

	public List<Person>getMyList(){
		return this.myList;
	}

	private final PropertyChangeSupport propertyChangeSupport = new
PropertyChangeSupport(
			this);

	public void addPropertyChangeListener(PropertyChangeListener listener) {
		propertyChangeSupport.addPropertyChangeListener(listener);
	}

	public void addPropertyChangeListener(String propertyName,
			PropertyChangeListener listener) {
		propertyChangeSupport.addPropertyChangeListener(propertyName,
listener);
	}

	public void removePropertyChangeListener(PropertyChangeListener listener) {
		propertyChangeSupport.removePropertyChangeListener(listener);
	}

	public void removePropertyChangeListener(String propertyName,
			PropertyChangeListener listener) {
		propertyChangeSupport.removePropertyChangeListener(propertyName,
				listener);
	}
```

```
	protected void firePropertyChange(String propertyName, Object oldValue,
			Object newValue) {
		propertyChangeSupport.firePropertyChange(propertyName, oldValue,
				newValue);
	}
}
```

Quadro 1 - Código fonte de *personTableModel*

Após a criação do modelo, foi *declarado personTableModel* e um objeto *Person* no código do formulário conforme Quadro 2.

```
package br.com.claupers.crm.views;

import java.awt.EventQueue;

public class Main {

	private JFrame frmPersonsRegistry;
	private JTextField id;
	private JTextField txtName;
	private JTable table;
	private JButton btnAddcity;
	private JComboBox cbxCity;
	private JButton btnNew;
	private JButton btnSave;
	private JButton btnChange;
	private JButton btnSearch;
	private JButton btnDelete;
	private PersonTableModel personTableModel;
	private Person person;
```

Quadro 2 - Declarando objetos para vinculação do formulário principal

Feito isso, foi possível realizar a vinculação da *JTable* com o modelo de tabela conforme Figura 13.

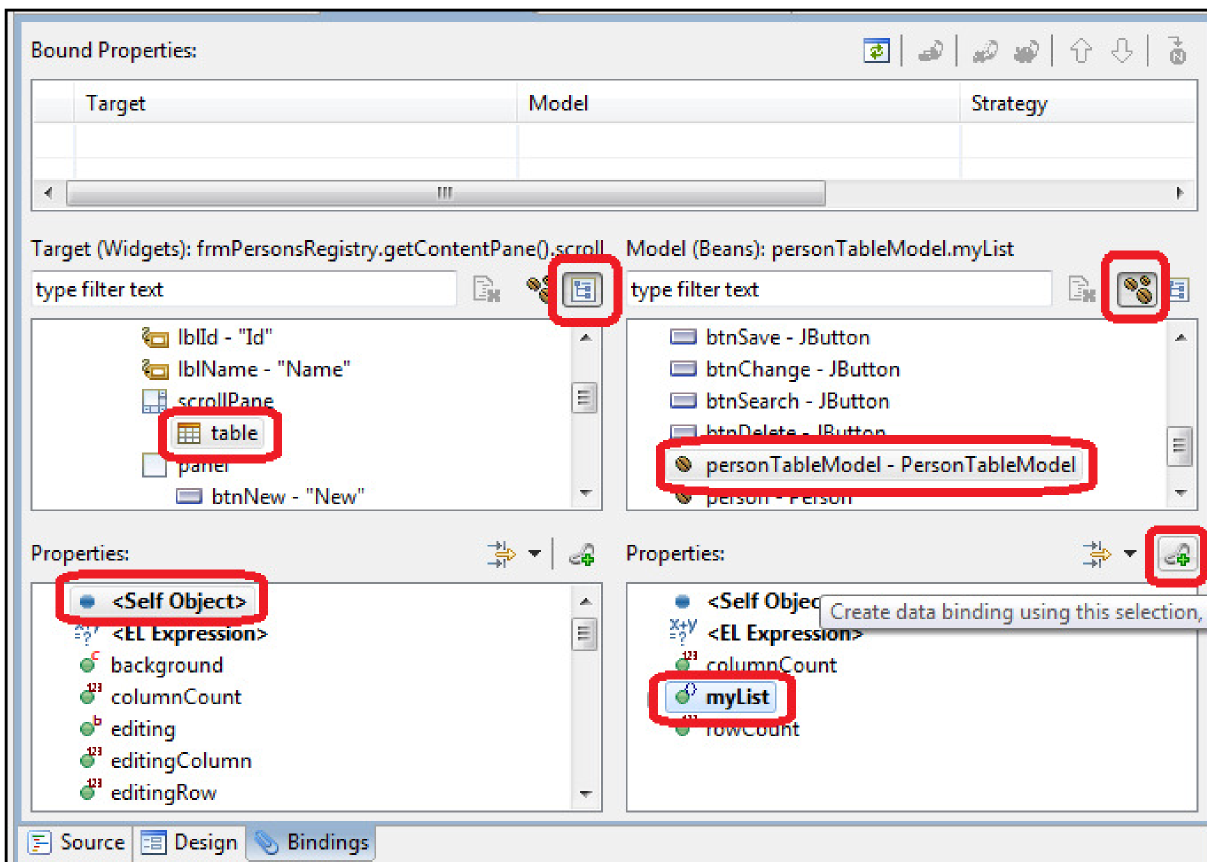

Figura 13 - Iniciando vinculação de *JTable*

A propriedade *myList* só foi exibida devido ao comando *getMyList* de *personTableModel.* Dessa forma, para usar uma determinada propriedade, um método *get* deve ser inserido. Se o nome do método fosse *getMyList2* a propriedade seria *myList2*.

3.3.7 Objeto de Acesso a Dados (DAO)

Para testar, foi inserido manualmente alguns dados em ambas as tabelas e criados objetos de acesso a dados (DAO) para popular a lista conforme Quadro 3.

```
package br.com.claupers.crm.dao;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.Query;

import br.com.claupers.crm.entities.Person;

public class PersonDao {
	private EntityManagerFactory emf;
	private List<Person>personList = new ArrayList<Person>();


	public PersonDao(EntityManagerFactory emf){
		this.emf=emf;
	}


	public void setEmf(EntityManagerFactory emf){
		this.emf=emf;
	}


	@SuppressWarnings("unchecked")
	public List<Person> getAllPersons(){
		EntityManager em = this.emf.createEntityManager();

		Query query = em.createQuery("SELECT p FROM Person p ORDER BY 
p.name");
		this.personList = query.getResultList();
		em.close();
		return this.personList;
	}

	@SuppressWarnings("unchecked")
	public List<Person> getLastFive(){
		EntityManager em = this.emf.createEntityManager();
		StringBuffer sb = new StringBuffer();
		sb.append("SELECT p FROM Person p ORDER BY p.name");
		Query query = em.createQuery(sb.toString());
		query.setMaxResults(5);
		this.personList = query.getResultList();
		em.close();
		return this.personList;
	}
	public void persist(Person person){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		em.persist(person);
		em.getTransaction().commit();
		em.close();
	}
	public void remove (int id){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		Query query = em.createQuery("SELECT p FROM Person p WHERE p.id = 
:id");
		query.setParameter("id",id);
		Person person = (Person) query.getSingleResult();
		em.remove(person);
		em.getTransaction().commit();
		em.close();
	}
```

```
	public void merge (Person person){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		em.merge(person);
		em.getTransaction().commit();
		em.close();
	}

	@SuppressWarnings("unchecked")
	public List<Person> likeName(String search){
		EntityManager em = this.emf.createEntityManager();

		Query query = em.createQuery("SELECT p FROM Person p WHERE
UPPER(p.name) LIKE :name ORDER BY p.name");
		query.setParameter("name", "%"+search.toUpperCase()+"%");
		this.personList=query.getResultList();
		em.close();
		return this.personList;
	}

}
```

Quadro 3 - Código fonte de *PersonDao*

Logo em seguida, o objeto foi declarado no código fonte do formulário *Main* juntamente com uma *EntityManagerFactory*. Em seguida instanciou-se os objetos durante a inicialização do formulário conforme Quadro 4.

```
package br.com.claupers.crm.views;

import java.awt.EventQueue;

public class Main {
	...
	private PersonTableModel personTableModel;
	private Person person;
	private PersonDao personDao;
	private EntityManagerFactory emf;
	...
	public static void main(String[] args) {
		...
	}

	public Main() {
		initialize();
	}
	private void initialize() {
		frmPersonsRegistry = new JFrame();
		...
		emf =  Persistence.createEntityManagerFactory("crm");
		personDao = new PersonDao(emf);

		...
		table = new JTable();
personTableModel = new PersonTableModel();
		table.setModel(personTableModel);
```

```
                table.setAutoCreateColumnsFromModel(false);
personTableModel.addAll(personDao.getAllPersons());
                scrollPane.setViewportView(table);
                ...
        }
```

Quadro 4 - Carregando dados na *jTable*

3.3.8 JTextFields e sua Vinculação ao JTable

Executado neste ponto, o aplicativo exibiu os registros inseridos manualmente na base. O próximo passo foi vincular os *JTextFields* ao *JTable*. A Figura 14 demonstra o processo.

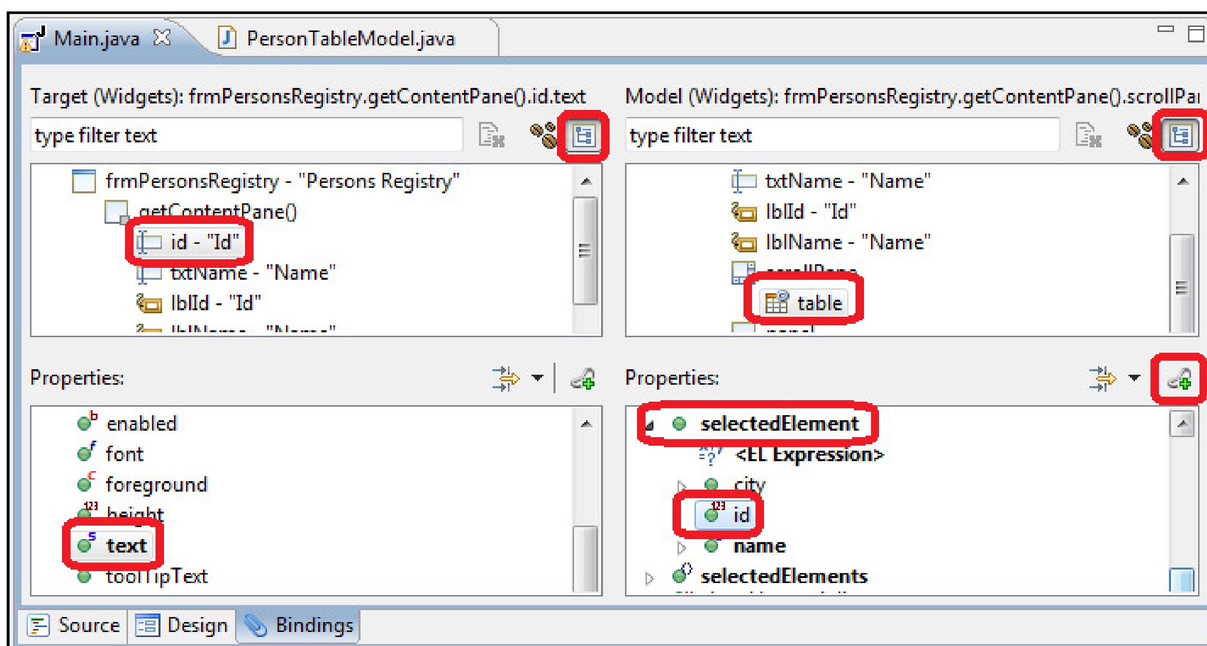

Figura 14 - Vinculando *jTextField* a *jTable*

3.3.9 *JComboBox* Vinculado e Alimentado com Dados

Para o *jComboBox* existem algumas mudanças para efetuar a vinculação em relação a *JTextField*. A propriedade a ser usada é a *selectedItem.*

Feito a vinculação do *jComboBox*, foi necessário popular o mesmo com os objetos com o uso do *For* otimizado. Para isso,criou-se um DAO para fazer acesso aos dados conforme Quadro 5.

```
package br.com.claupers.crm.dao;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;

import br.com.claupers.crm.entities.City;

public class CityDao {
private EntityManagerFactory emf;
       private List<City>cityList = new ArrayList<City>();
private List<Integer>idCityList = new ArrayList<Integer>();


      public CityDao(EntityManagerFactory emf){
            this.emf=emf;
      }

      publicvoid setEmf(EntityManagerFactory emf){
            this.emf=emf;
      }

      @SuppressWarnings("unchecked")
      public List<City> getAllCities(){
            EntityManager em = this.emf.createEntityManager();

            Query query = em.createQuery("SELECT p FROM City p ORDER BY
p.name");
            this.cityList = query.getResultList();
            em.close();
            return this.cityList;
      }

      @SuppressWarnings("unchecked")
      public List<City> getLastFive(){
            EntityManager em = this.emf.createEntityManager();
            StringBuffer sb = new StringBuffer();
            sb.append("SELECT p FROM City p ORDER BY p.name");
            Query query = em.createQuery(sb.toString());
            query.setMaxResults(5);
            this.cityList = query.getResultList();
            em.close();
            return this.cityList;
      }
@SuppressWarnings("unchecked")

      public List<Integer> getIdCityList(){
            EntityManager em = this.emf.createEntityManager();
```

```
		Query query = em.createQuery("SELECT p.id FROM City p ORDER BY 
p.name");
		this.idCityList = query.getResultList();
		em.close();
		return this.idCityList;
	}
	public void persist(City city){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		em.persist(city);
		em.getTransaction().commit();
		em.close();
	}
	public void remove (int id){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		Query query = em.createQuery("SELECT p FROM City p WHERE p.id = 
:id");
		query.setParameter("id",id);
		City city = (City) query.getSingleResult();
		em.remove(city);
		em.getTransaction().commit();
		em.close();
	}
	public void merge (City city){
		EntityManager em = this.emf.createEntityManager();
		em.getTransaction().begin();
		em.merge(city);
		em.getTransaction().commit();
		em.close();
	}

	@SuppressWarnings("unchecked")
	public List<City> likeName(String search){
		EntityManager em = this.emf.createEntityManager();
		Query query = em.createQuery("SELECT p FROM City p WHERE 
UPPER(p.name) LIKE :name ORDER BY p.name");
		query.setParameter("name", "%"+search.toUpperCase()+"%");
		this.cityList=query.getResultList();
		em.close();
		return this.cityList;
	}
}
```

Quadro 5 - Código fonte de CityDao

Em seguida, foi declarada uma lista para receber os dados e o DAO no código fonte do formulário Main como é mostrado no Quadro 6.

```
...
	private Person person;
	private PersonDao personDao;
	private EntityManagerFactory emf;
	private List<City>cityList = new ArrayList<City>();
	private CityDao cityDao;
...
```

Quadro 6 - Declarando Lista e DAO das cidades

Depois de declarados, os objetos alimentaram o *JComboBox* com os dados. O Quadro 7 demonstra esse procedimento.

```
...
            cbxCity = new JComboBox();
            cbxCity.setBounds(30, 111, 241, 20);
            frmPersonsRegistry.getContentPane().add(cbxCity);

            cityDao = new CityDao(emf);
            cityList = cityDao.getAllCities();
            for(City city: cityList){
            cbxCity.addItem(city);
            }
...
```

Quadro 7 - Populando *JComboBox*

Com isso o *JComboBox* recebeu os objetos existentes na base de dados. O que foi exibido não era o nome da cidade mas sim o do objeto que armazenava essas informações. Para exibir a informação desejada, a entidade *City* teve seu método *toString* sobrescrito. Isso foi feito de forma simplificada no Eclipse, bastando usar a opção de sobrescrever métodos. Com isso, uma tela permitiu a escolha do método a ser sobrescrito. Nesse caso a opção foi pelo *toString* e o resultado final é do Quadro 8 e a entidade passou a exibir o nome da cidade.

```
@Override
	public String toString() {
		// TODO Auto-generated method stub
		return this.getName();
	}
```

Quadro 8 - Código fonte do método *toString* da entidade *City* sobrescrito

3.3.10 Entidades Preparadas para *Beansbinding*

Para que as entidades funcionassem adequadamente com *binding,* algumas alterações em seus métodos fizeram-se necessárias. Quando elas foram geradas pelo Eclipse, ele não implementou códigos para disparar as alterações nas propriedades como pode ser visto no Quadro 9.

```
...
public Person() {
    }

      ...
      public void setId(int id) {
            this.id = id;
      }

      public void setName(String name) {
            this.name = name;
      }

      public void setCity(City city) {
            this.city = city;
      }
      ...
```
Quadro 9 - Métodos *Sets* originas da entidade *Person*

Para que funcionem corretamente, primeiramente foi necessário criar um objeto conforme Quadro 10 e depois alterar os métodos *sets* da entidade para ficar com a estrutura exposta no Quadro 11.

```
package br.com.claupers.crm.model;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public abstract class AbstractModelObject {
      protected final PropertyChangeSupport propertyChangeSupport = new
PropertyChangeSupport(
                  this);
      public void addPropertyChangeListener(PropertyChangeListener listener) {
            propertyChangeSupport.addPropertyChangeListener(listener);
      }
      public void addPropertyChangeListener(String propertyName,
                  PropertyChangeListener listener) {
            propertyChangeSupport.addPropertyChangeListener(propertyName,
listener);
      }
      public void removePropertyChangeListener(PropertyChangeListener listener) {
            propertyChangeSupport.removePropertyChangeListener(listener);
      }
      public void removePropertyChangeListener(String propertyName,
                  PropertyChangeListener listener) {
            propertyChangeSupport.removePropertyChangeListener(propertyName,
                        listener);
      }
      protected void firePropertyChange(String propertyName, Object oldValue,
                  Object newValue) {
            propertyChangeSupport.firePropertyChange(propertyName, oldValue,
                        newValue);
      }
}
```
Quadro 10 - Código fonte de *AbstractModelObject*

Agora que o objeto *AbstractModelObject* foi criado é possível estende-lo na entidade e assim herdar suas propriedades.

```
public class Person extends AbstractModelObject implements Serializable  {
...
public void setId(int id) {
		int oldValue = this.id;
		this.id = id;
		firePropertyChange("id", oldValue, this.id);
	}


	public void setName(String name) {
		String oldValue = this.name;
		this.name = name;
		firePropertyChange("name", oldValue, this.name);

	public void setCity(City city) {
		City oldValue = this.city;
		this.city = city;
		firePropertyChange("city", oldValue, this.city);
	}
```

Quadro 11 - Métodos *sets* adaptados da entidade *Person* para *binding*

Ao estender o objeto *AbstractModelObject* a entidade recebeu o suporte para disparar as alterações das propriedades.

3.3.11 Controle de Ativação e Desativação dos Componentes da Tela

Para este trabalho foi definido que a inserção, alteração ou exclusão de um registro será permitida apenas após clicar no botão que chame cada método. Para evitar que o usuário clicasse, por exemplo, no botão excluir antes de salvar uma nova inserção, foi necessário controlar o estado e ativo ou inativo dos componentes na tela, para isso usou-se o *binding d*a propriedade *enabled* dos componentes em uma *EL Expression* do *JTable* como é exibido na Figura 15.

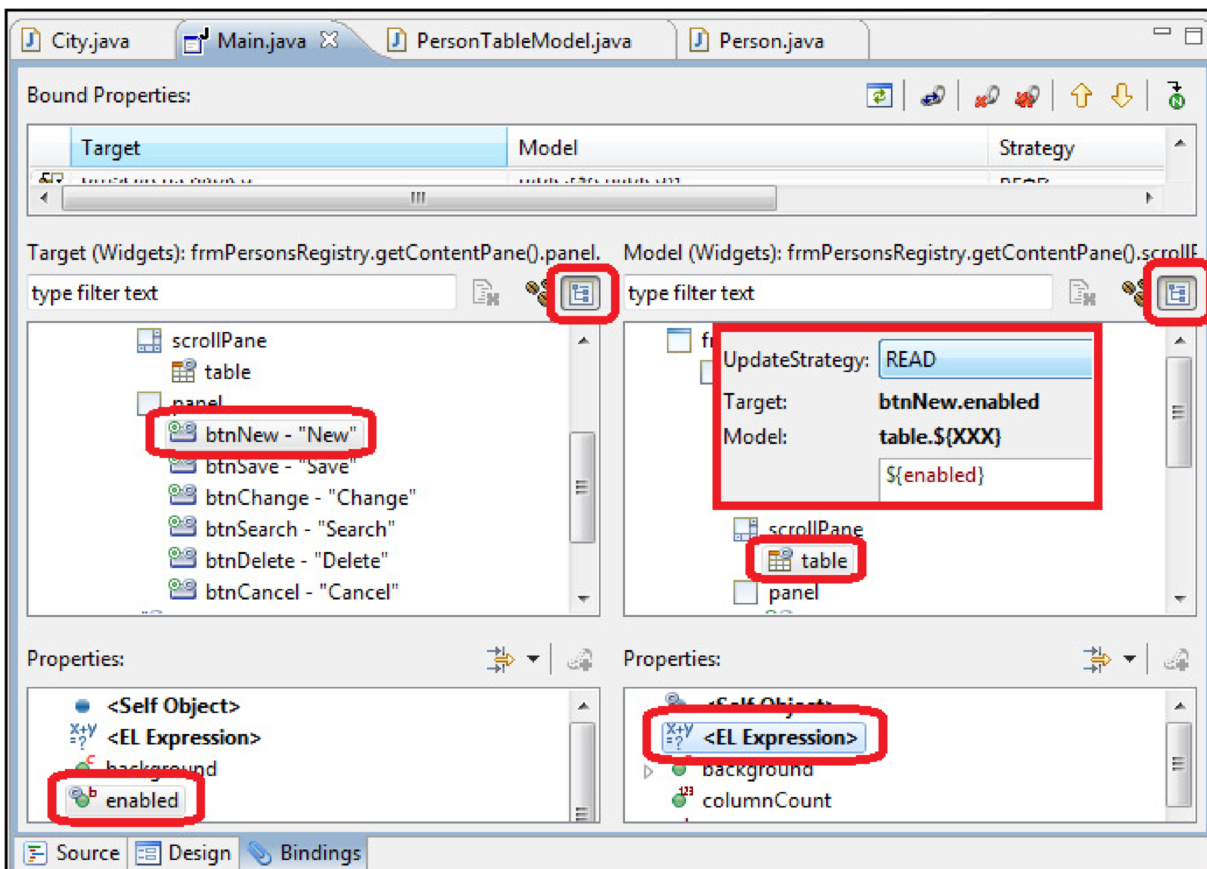

Figura 15 - Criando vinculação entre botões e tabela

Como pode ser observado, foi criado uma vinculação entre a propriedade *enabled* do *jButton* e uma *ELExpression* da *JTable.* No recorte que foi feito pode se observar que a expressão usada foi ${*enabled*}, ou seja, quanto a tabela estiver ativa o botão também estará. O resultado final para a vinculação dos botões é o exibido na Figura 16.

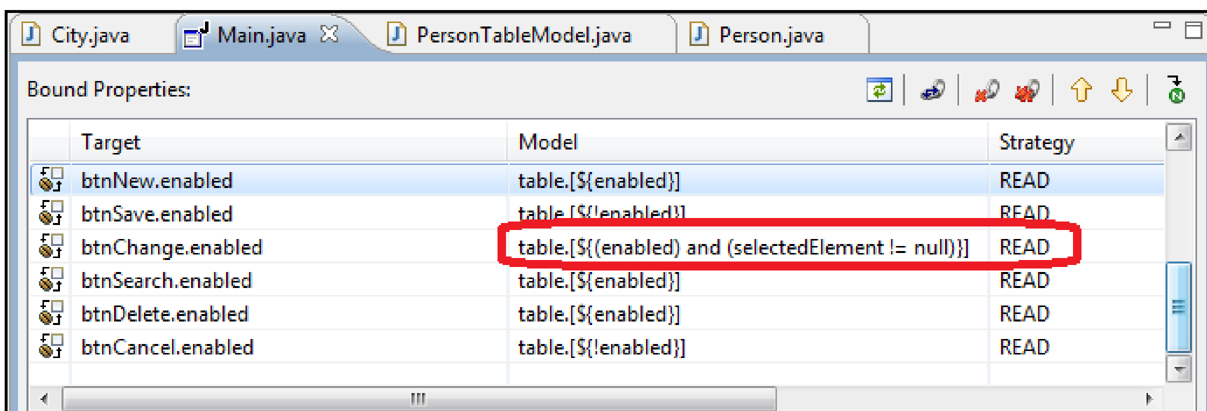

Figura 16 - Resultado das vinculações entre *Jtable* e *JButton*

Como pode ser observado, para botão *Change* além da tabela estar ativa, tinha que ter um registro selecionado.

O mesmo conceito aplicou-se para os campos, com exceção do campo *id* que sempre estaria desativado. A vinculação ficou conforme Figura 17.

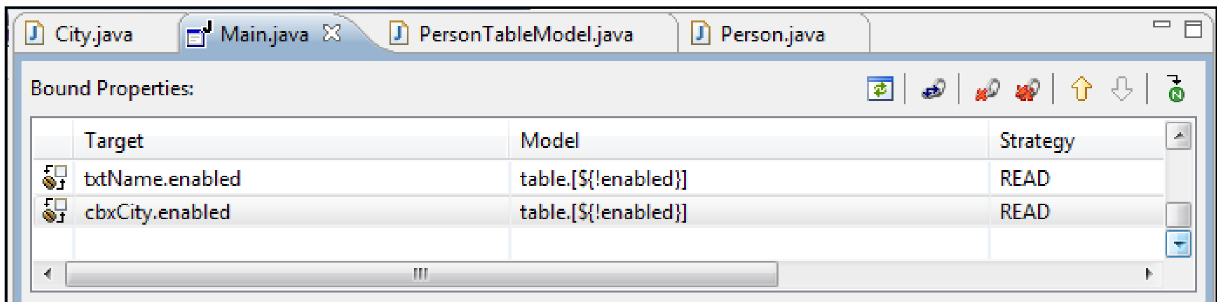

Figura 17 - Resultado da vinculação de campos a tabela

Com isso os campos só receberiam alterações quando a tabela estivesse inativa, o que podia ser controlado dentro dos métodos dos botões.

3.3.12 Método do Botão *New*

Para iniciar a implementação do método no botão *New*, bastou visualizar o formulário e dar dois cliques sobre o mesmo para que fosse gerado o código inicial. O restante do método exibido no Quadro 12 foi digitado manualmente.

```
...
private String Status="";
btnNew.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent arg0) {
     table.setEnabled(false);
     person = new Person();
     personTableModel.add(person);
     table.repaint();
     txtName.requestFocus();
     table.addRowSelectionInterval(table.getRowCount()-1, table.getRowCount()-1);
     txtName.requestFocus();
     Status="New";
  }
});
...
```

Quadro 12 - Código do método do botão *new*

No código do Quadro 12 também foi gerada uma variável do tipo *string* para armazenar o *status* da operação que pode ser *new* ou *change.* Essa informação é usada no botão salvar.

3.3.13 Método do Botão *Cancel*

O método do botão *Cancel* ativa a tabela e, caso o *status* seja *new* simplesmente remove o objeto *person* do modelo da tabela, caso contrário ele remove o objeto alterado e carrega da base o original sem mudanças e adiciona no mesmo índice que estava. O Quadro 13 mostra esse processo em detalhes.

```
btnCancel.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
	table.setEnabled(true);
int index = table.getSelectedRow();
	Person personCancel = personTableModel.getPerson(index);
	if(Status=="New"){
		personTableModel.remove(person);
	}else if (Status=="Change"){
		personTableModel.remove(index);
		person=personDao.getPerson(personCancel.getId());
		personTableModel.add(index,person);
		table.addRowSelectionInterval(index, index);
	}
}
});
```
Quadro 13 - Código do método do botão *cancel*

3.3.14 Método do Botão *Change*

O método do botão *change* também era simples. Ele apenas desativava a tabela, em seguida informava que o *status* era de alteração e solicitava o foco para o campo do nome como pode ser visto no Quadro 14.

```
btnChange.addActionListener(new ActionListener() {
			publicvoid actionPerformed(ActionEvent arg0) {
				table.setEnabled(false);
				Status="Change";
				txtName.requestFocus();
			}
		});
```

Quadro 14 - Código do método do botão *change*

3.3.15 Método do Botão *Delete*

O método *delete* carregava na linha de *personTableModel*, com base no índice, a pessoa selecionada para um novo objeto *person*. Em seguida questionava o usuário do sistema sobre a exclusão ou não. Em caso positivo *personDao* excluiria a pessoa do banco de dados com base no campo chave sendo que logo após o objeto *person* também era removido do modelo da tabela para atualizar a tela. Tudo isso é demonstrado no Quadro 15.

```
btnDelete.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent arg0) {
	Person person = personTableModel.getPerson(table.getSelectedRow());
	Integer excluir=JOptionPane.showConfirmDialog(null, "Deseja excluir a 
pessoa "+person.getName()+"?");
				if (excluir==0){
					personDao.remove(person.getId());
					personTableModel.remove(person);	}
			}
		});
```

Quadro 15 - Código do método do botão *delete*

3.3.16 Método do Botão S*ave*

O botão *Save* era responsável por persistir os dados no banco tanto na inclusão e alteração. Por isso foi necessário a criação da variável *status*, pois ao clicar no botão *save* o sistema precisava saber qual procedimento estava sendo feito para disparar o processo correto. Antes de implementar o botão *save,* foi necessário realizar uma alteração na entidade *Person* para que, quando fosse inserido um novo registro e clicado no botão *save*, atualizasse na *JTable* de forma visual o campo *id.*

Caso isso não fosse feito, ao clicar em salvar o valor de *id*, visualmente, seria 0 até que algo disparasse uma alteração de propriedade. Para que a chave primária atualizasse no componente visual foi necessário criar um método na entidade *Person* conforme Quadro 16.

```
publicvoid refreshId(){
             int oldValue = 0;
             firePropertyChange("id", oldValue, this.id);
       }
```

Quadro 16 - Atualizando campo *id* da entidade *Person* nos componentes visuais

Feito isso, bastava chamar o método *refreshId* no botão *save* após persistir a entidade na base de dados. Para evitar que o formulário fosse salvo com informações incompletas, criou-se um método para verificar se os campos foram preenchidos como exibido Quadro 17.

```
...
initDataBindings();
}

       private Boolean isCheckForm (Person person){
        if ((person.getName()== null) || (person.getName().equals(""))){
            JOptionPane.showMessageDialog(null, "Um nome deve ser informado.");
            txtName.requestFocus();
            return false;
       }else if (person.getCity()==null){
            JOptionPane.showMessageDialog(null,"Uma cidade deve ser informada.");
            cbxCity.requestFocus();
            return false;
       }else{
            Return true;
       }
}
      ...
```

Quadro 17-Código do método para verificação do formulário

Com essas alterações implementada, já era possível chamar o método do botão *save* dando dois cliques sobre ele e adicionando o código conforme Quadro 18.

```
btnSave.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent arg0) {
if (Status=="New"){
      if (isCheckForm(person)){
        table.setEnabled(true);
        personDao.persist(person);
        personTableModel.fireTableDataChanged();
        person.refreshId();
        table.setEnabled(true);
      }
}else if (Status=="Change"){
      Person person = personTableModel.getPerson(table.getSelectedRow());
      if (isCheckForm(person)){
        personDao.merge(person);
        table.setEnabled(true);
      }
}
}
});
```

Quadro 18 - Código do método do botão *save*

Por meio da variável *Status* o método tinha como saber qual processo devia executar e chamando o método *isCheckForm* verificava a validade das informações do formulário.

3.3.17 Método do Botão *Search*

O método do botão *search* exibia um diálogo de inserção para que o usuário digitasse parcialmente o nome a ser buscado como pode ser visto do Quadro 19.

```
btnSearch.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String pesq = JOptionPane.showInputDialog(null, "Digite o nome a localizar");
    if (pesq != null){
      personTableModel.clear();
      personTableModel.addAll(personDao.likeName(pesq));
    }
  }
});
```

Quadro 19 - Código do método do botão *search*

Antes de carregar a lista resultante da busca, o método limpava a atual, caso contrário o resultado seria adicionado ao final da que estava carregada da tabela modelo.

3.3.18 Método para Exibir Cidade Correta no *ComboBox*

Quando uma nova pessoa era inserida ou então alterada, automaticamente ao selecioná-la, o ComboBox exibia a cidade correta, mas quando uma nova lista era carregada o componente perdia a referencia a propriedade city da entidade person. Para resolver esse problema foi necessário criar o método do Quadro 20 para a JTable.

```
...
private List<Integer>idCityList = new ArrayList<Integer>();
...

idCityList = cityDao.getIdCityList();
table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
@Override
public void valueChanged(ListSelectionEvent e) {
// TODO Auto-generated method stub
if (table.getSelectedRow()> 0){
	int index = table.getSelectedRow();
	Person person = personTableModel.getPerson(index);
	City city = person.getCity();
	if (city!=null) {
	  cbxCity.setSelectedIndex(idCityList.indexOf(city.getId()));
	}
}
}
});
```

Quadro 20 - Adicionando ouvinte de seleção de lista a *jTable* e declarando nova lista *idCityList*

No código acima foi criada uma nova lista para armazenar os campos chaves das cidades. Em seguida foi adicionado um ouvinte de seleção de lista a *jTable*, com isso, a cada vez que uma nova linha é selecionada, esse método era disparado. Em seguida, dentro do método, era verificado se realmente uma linha foi selecionada, se sim, o índice era armazenado e usado para pegar o objeto *person* da linha. Dessa forma era possível verificar qual a cidade da pessoa e usar sua *id* para atualizar o *ComboBox.*

Para gerar esse código, foi empregado o recurso do Eclipse que gera o método com base no que já foi digitado. Isso facilita, pois mesmo sabendo apenas parte do método é possível usá-lo.

3.4 RELATÓRIOS NO *IREPORT*

Para geração dos relatórios foi usado o *iReport* 4.1.3. O *download* do instalador foi feito no *site* http://jasperforge.org/projects/ireport. Como existia instaladores para múltiplas plataformas, bastou escolher a mais adequada e efetuar a instalação.

3.4.1 *Drive* de Conexão a Base de Dados no *iReport*

Um dos primeiros passos foi adicionar o *drive* que seria usado para acesso a base de dados. Para isso, copiou-se o arquivo para pasta padrão do *iReport* e em seguida adicionado ao *classpath* do *framework.* Com isso já era possível configurar e acessar a base de dados como é mostrado na Figura 18.

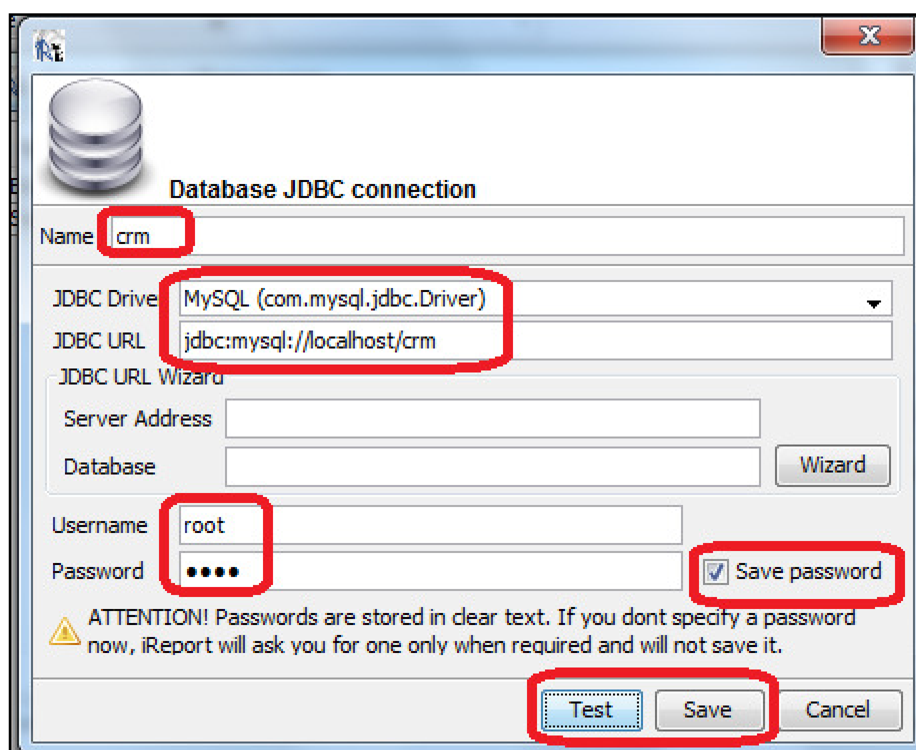

Figura 18 - Configurando a conexão no *iReport*

Após essa etapa, a conexão estava funcionando e a criação do relatório já podia ser iniciada.

Concluído a configuração do *drive,* já era possível criar o relatório. Para isso foi acessado o menu Arquivo\new abrindo dessa forma a tela para escolha de um modelo. Nesta etapa, um modelo de relatório em branco foi selecionado para iniciar os testes.

3.4.2 Exibição de Janelas no *iReport*

Foi necessário ativar algumas janelas com ferramentas, conforme é demonstrado na Figura 19, que são necessárias para auxiliar na criação do relatório.

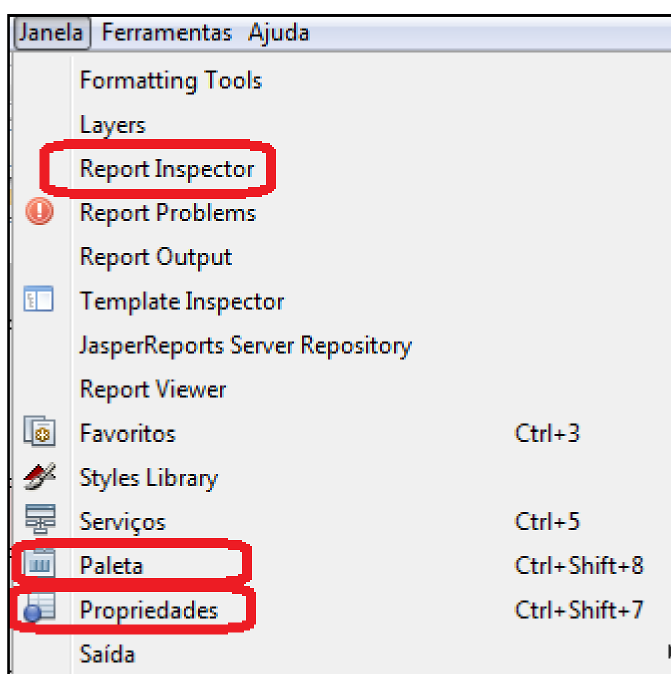

Figura 19 - Ativando janelas no *iReport*

Com essas janelas foi possível manipular o relatório. Na janela *Report Inspector* era exibido tudo que fazia parte do relatório, tornando possível realizar uma navegação de forma facilitada durante a manutenção do relatório. A janela de propriedades exibia as informações do componente selecionados no *Report*

Inspector e na Paleta, os componentes disponíveis para uso no relatório conforme é exibido na Figura 20.

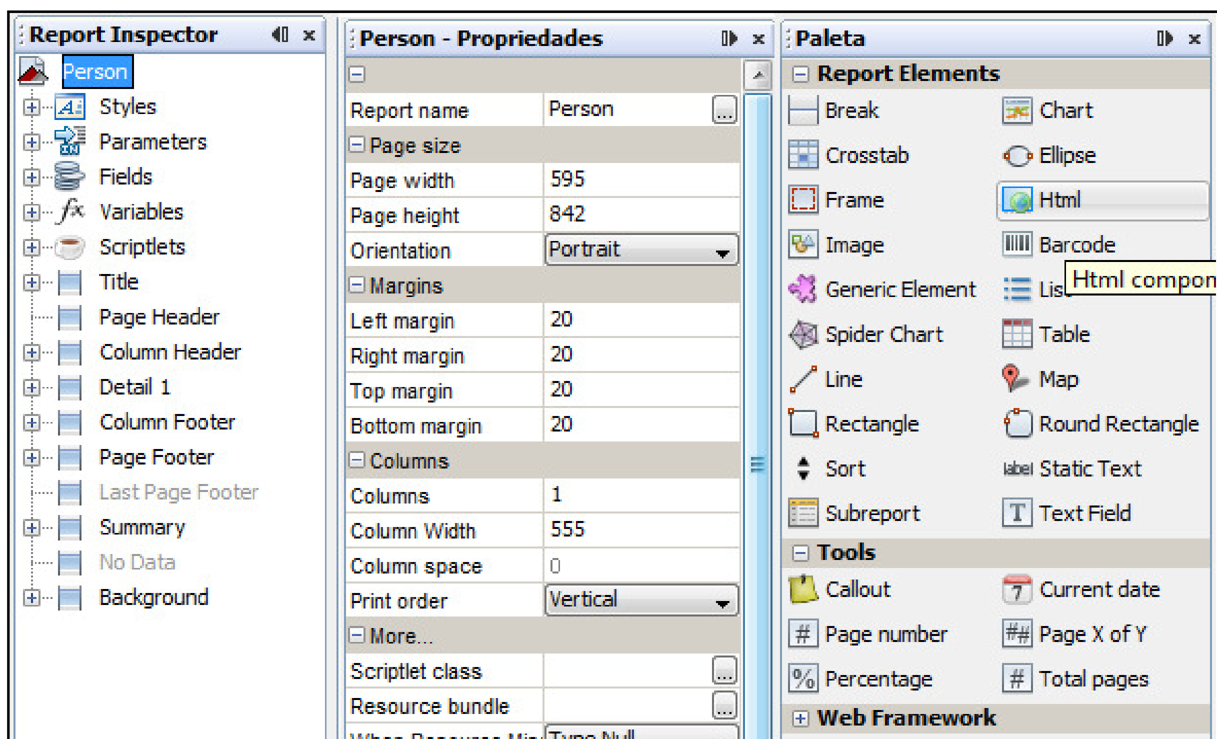

Figura 20 - Janelas do *iReport*

Essas são as janelas mais usadas no sistema durante a elaboração dos relatórios.

3.4.3 Consulta *Sql* para Gerar os Campos do *iReport*

Um dos primeiros passos foi criar uma consulta SQL para criar os campos. Isso é feito em tela específica que abre ao clicar no botão da tela inicial como é mostrado na Figura 21.

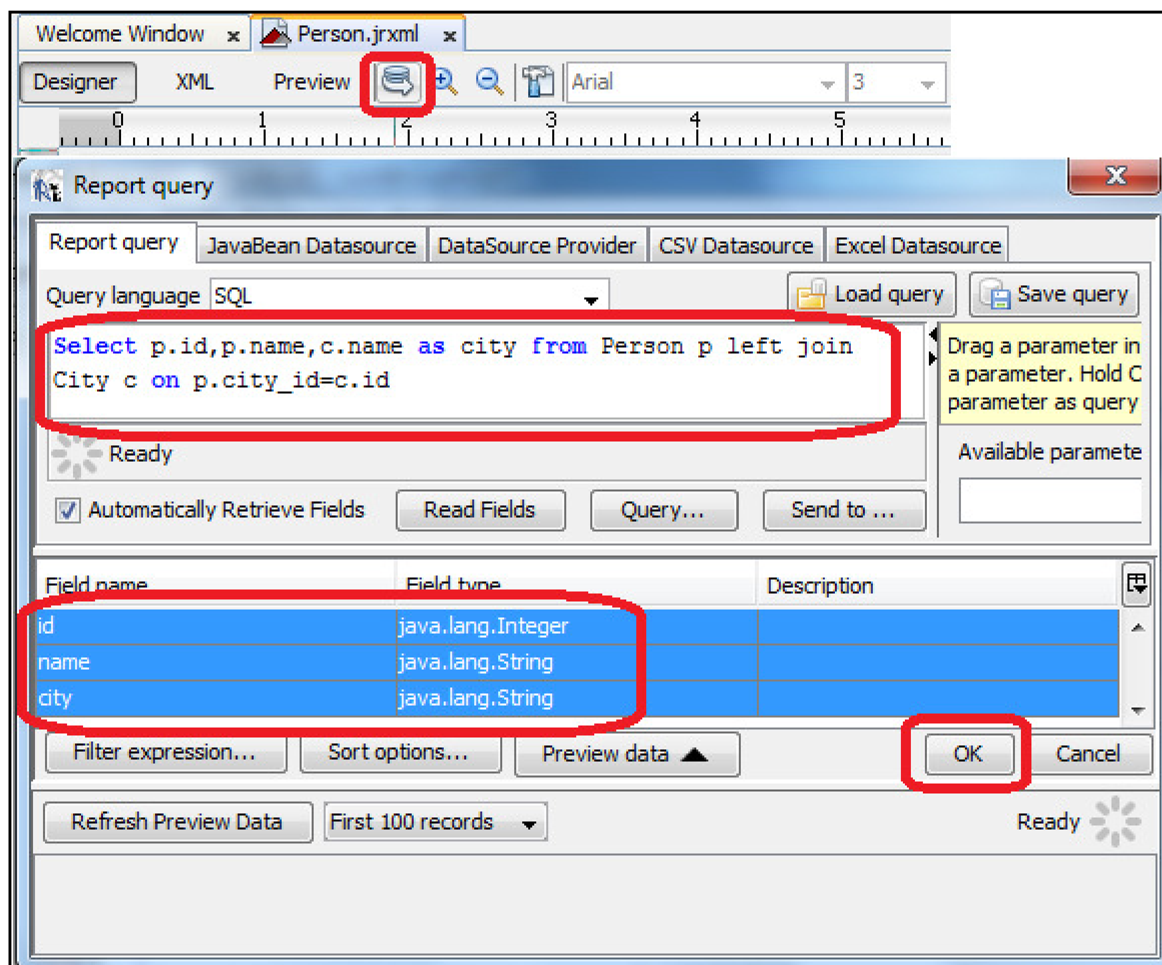

Figura 21 - Criando campos através de código SQL no *iReport*

Como pode ser visto na Figura 21, a ferramenta gerou os campos com base no código SQL. Mesmo que um determinado campo exista em outra tabela, pode ser feito um *join* que o sistema usa o resultado da consulta.

Após clicar no botão *ok* da tela acima, a ferramenta disponibilizou os campos para uso no *Report Inspector.*

3.4.4 Adição de Campos ao Relatório

Para adicionar os campos ao relatório, bastou clicar sobre os mesmos no *Report Inspector* e arrastar até a banda *detail* como é exibido na Figura 22.

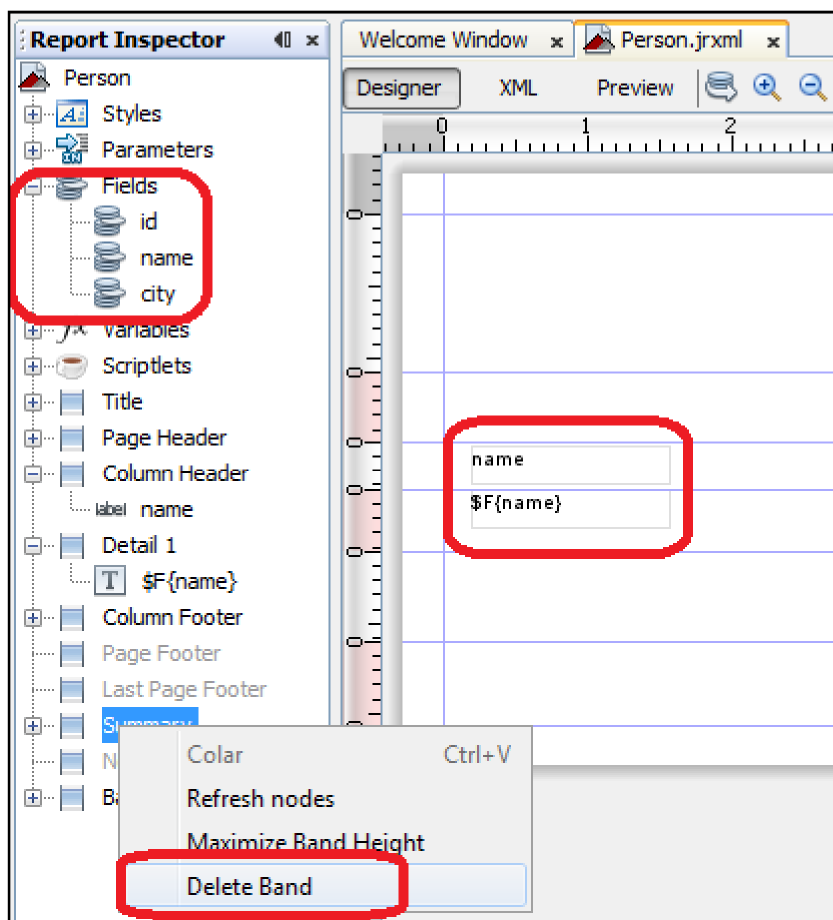

Figura 22 - Adicionando campo ao formulário e removendo bandas desnecessárias

Ao arrastar o campo para a banda *detail,* automaticamente o sistema adicionou um nome para ela na banda de cabeçalho da coluna. Esse é o processo para criação do relatório na parte visual.

3.4.5 Parâmetro para o Relatório

A casos em que, ao chamar um relatório, pode se querer que seja exibida uma informação filtrada. Para isso é necessário adicionado um parâmetro ao relatório como é exemplificado na Figura 23.

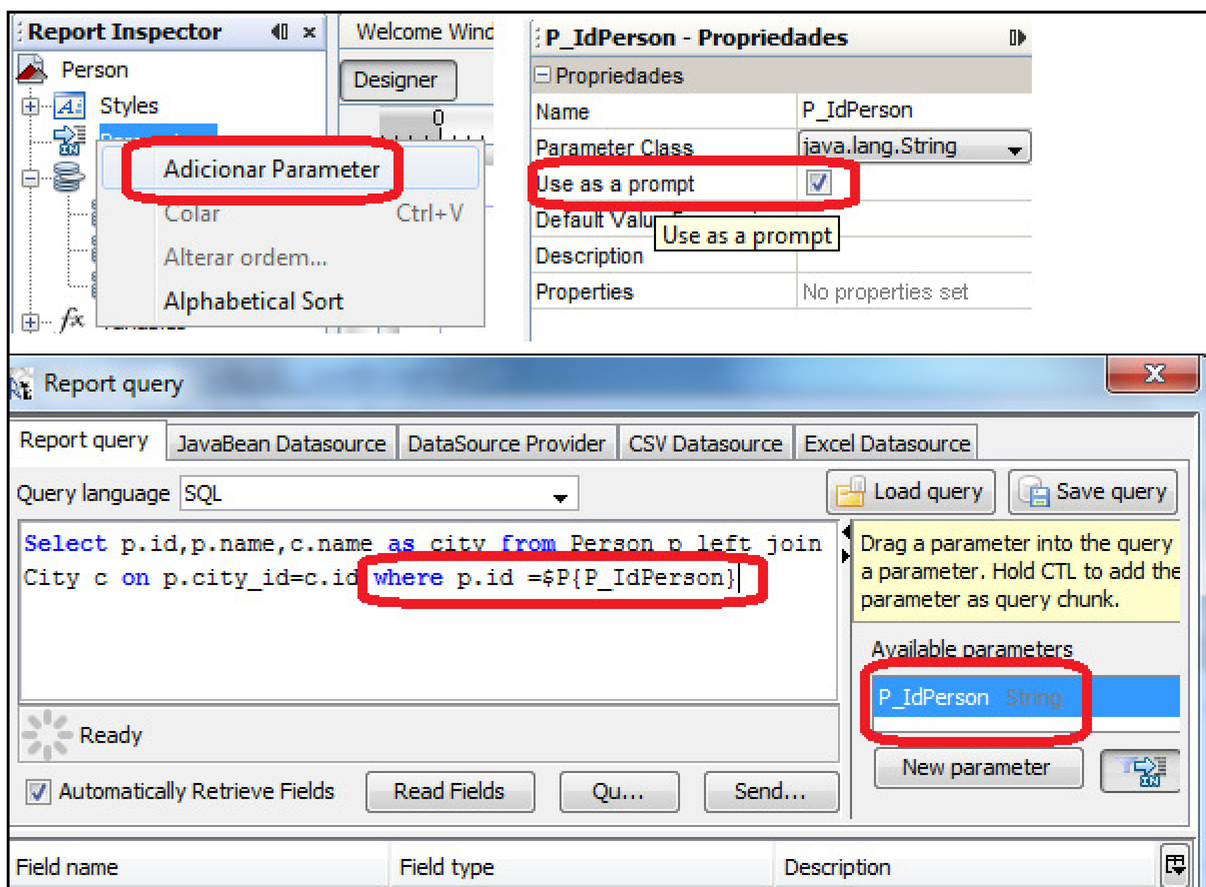

Figura 23 - Adicionando parâmetro ao consulta Sql do *iReport*

Com isso foi possível exibir apenas uma determinada pessoa no relatório. Como na propriedade do parâmetro está marcado para usar *prompt*, foi possível testar o relatório dentro do próprio *iReport*, para isso bastou clicar no botão *Preview* logo acima do relatório. Antes de exibir o relatório, era solicitado o código do usuário.

3.4.6 Compilação do Relatório

A estrutura final do relatório é a exibida abaixo na Figura 24. Após compilar o relatório, o arquivo *jasper* do mesmo foi copiado para um pacote do sistema no Eclipse.

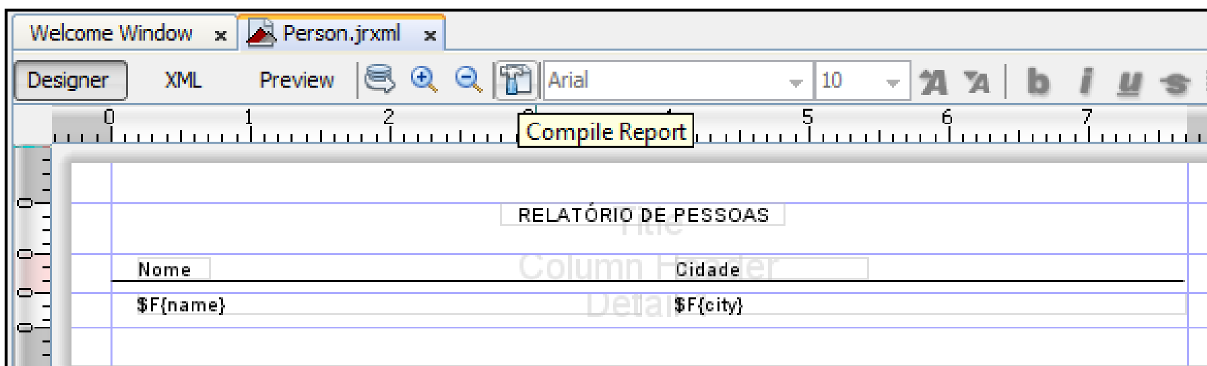

Figura 24 - Compilando relatório

3.4.7 Geração de Arquivo *jar* das Fontes

Para que a exportação do relatório em alguns formatos ocorresse corretamente, foi necessário adicionar um arquivo *jar* com as fontes usadas. Neste caso, todos os componentes do relatório tiveram sua fonte alterada para Arial sendo o único tipo necessário. O primeiro passo foi obter as fontes, em seguida seguir os passos mostrados na Figura 25.

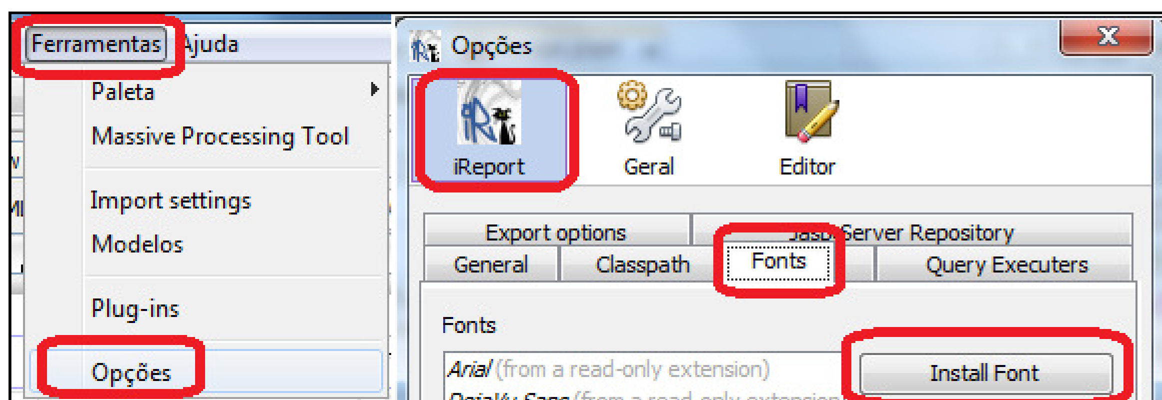

Figura 25 - Instalando fonte para o *iReport*

Ao clicar no botão *Install Font,* foi aberto um assistente que solicitou onde estava salva a fonte que, em seguida, requisitou os detalhes da fonte. O próximo passo foi definir a língua padrão dos relatórios sendo logo após finalizado o processo.

Com isso, a fonte foi adicionada ao *iReport* e os relatório executados e exportados dentro dele, exibidos corretamente.Para que funcionasse na aplicação,

foi necessário exportar a fonte para um *jar* e adicioná-la a aplicação. O processo foi feito na mesma tela onde foi iniciada a instalação da fonte.

3.5 ADICIONAR RELATÓRIO NO PROJETO

O primeiro passo para adicionar o relatório ao projeto foi copiar o arquivo *jasper e o jar* da fonte para um mesmo pacote e também alguns arquivos do *iReport* e colocá-los na pasta *lib* do projeto e adicionado ao *classpath*. Esses arquivos estavam no diretório do *iReport* e são exibidos na Figura 26.

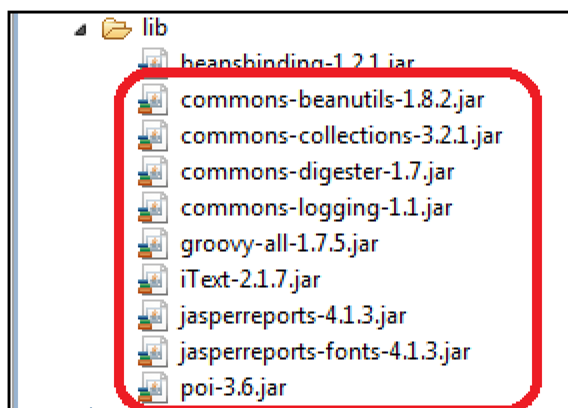

Figura 26 - Bibliotecas necessárias para *jasper*

Depois que as bibliotecas necessárias já estão disponíveis no *classpath,* foi implementado o método do botão *Print*.

```
btnPrint.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent arg0) {
	Toolkit tk = Toolkit.getDefaultToolkit();
	Dimension screenSize = tk.getScreenSize();
	try{
	URL jasper =
this.getClass().getResource("/br/com/claupers/crm/reports/Person.jasper");
	InputStream fs;
	fs = (InputStream) jasper.getContent();
	Class.forName("com.mysql.jdbc.Driver");
	Map<String, Object> map = new HashMap<String, Object>();
	Person person = personTableModel.getPerson(table.getSelectedRow());
	Integer id = person.getId();
map.put("P_IdPerson",id );
	Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/crm", "root" , "root" );
```

```
JasperPrint print = JasperFillManager.fillReport(fs,map,con);
	if (print.getPages().size()>0)
	{
		JDialog formView = new JDialog();
		formView.setModal(true);
		formView.setTitle("Visualização de relatórios");
		formView.setSize(screenSize.width, screenSize.height);
		JasperViewer viewer = new JasperViewer(print,false);
		formView.getContentPane().add(viewer.getContentPane());
		formView.setVisible(true);

	}else{ JOptionPane.showMessageDialog(null, "Nenhum registro encontrado. 
Informe outro periodo.");
	}
con.close();
	 }catch (Exception e){
	e.printStackTrace();
	}
}
});
```

Quadro 21 - Código do método do botão *Print*

Para melhorar o controle sobre a visualização foi criado um formulário próprio, dessa forma o mesmo é exibido em *modal* para que somente seja possível carregar um após fechar o outro. Também foi criado um arquivo *HashMap* para ser possível passar o parâmetro *P_IdPerson*, dessa forma ao se clicar no botão *Print* é possível gerado um relatório apenas na pessoa selecionada.

3.6 ARQUIVO EXECUTÁVEL DO PROJETO

O Eclipse permitiu gerar de forma descomplicado um arquivo *jar* executável para distribuição da aplicação. Como isso, o sistema pôde ser distribuído de forma facilitada. Para fazer esse procedimento, bastou clicar sobre o projeto com o botão direito e ir na opção *Export* e seguir os passos indicados na Figura 27.

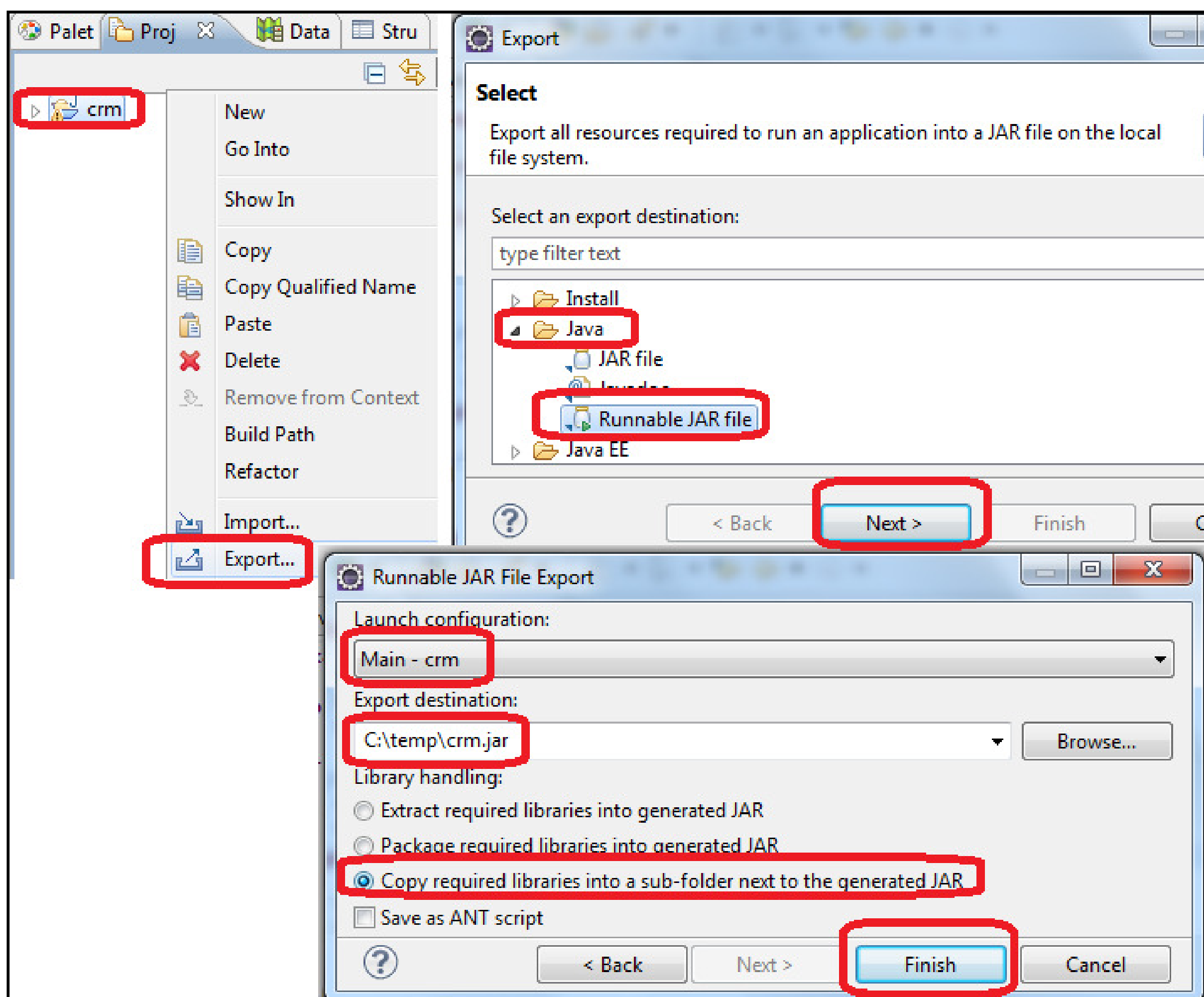

Figura 27 - Gerando arquivo *jar* executável do projeto

Com isso o arquivo *jar* ficou pronto para ser distribuído. Para testar, bastou dar dois cliques sobre o mesmo. Ao distribuir o arquivo, a pasta crm_lib que foi criada no mesmo diretório teve que ir junto pois contém as bibliotecas necessárias para o sistema.

3.7 ANÁLISE DOS DADOS

A criação da base de dados foi feito de forma simplifica com uso de *framework* sem a necessidade de uso da linguagem *SQL* pura. Com isso, mesmo não tendo conhecimento avançado de banco de dados, a manipulação pôde ser feita com mais qualidade.

Na etapa seguinte, a criação das entidades de forma reversa foi feita com auxílio do Eclipse, que gerou, com base no banco de dados, as entidades com todos os requisitos necessários para seu funcionamento. O processo não levou mais que um minuto, sendo que se fosse feito de forma manual, o tempo poderia ser dezenas de vezes maior.

Em relação a criação das telas e vinculação, o processo também foi otimizado. Para boa parte do desenvolvimento, não foi necessário trabalhar com código Java puro. O desenho, alinhamento de componentes, vinculação e chamada de métodos, foi feito com auxílio do *framework* que contribuiu com o desenvolvedor em relação a produtividade.

O relatório também pôde ser desenvolvido sem muitas complicações com um *framework* que alia produtividade e flexibilidade. Ao mesmo tempo que permite desenvolver de forma visual, também oferece a opção de testar o relatório antes mesmo de ser adicionado ao projeto. Isso diminui o tempo de desenvolvimento que o técnico levaria para implementar o relatório do projeto central para teste.

Por fim, com passos simples, foi possível usar o Eclipse e gerar um arquivo executável da aplicação para ser distribuída ao usuário final.

Todo o processo foi assistido por *frameworks* o que auxiliou de forma relevante no desenvolvimento. Com eles, mesmo um desenvolvedor sem conhecimentos avançados em Java pode dar os primeiros passos para uma boa aplicação.

4 RESULTADOS E DISCUSSÃO

Com o uso dos *frameworks* propostos, foi verificado que o tempo de desenvolvimento em plataforma Java pode ser otimizado. Procedimentos que levariam um tempo considerável para execução tiveram seu tempo reduzido consideravelmente. Um exemplo que pode ser citado é a criação das entidades de acesso a dados. Isso também auxilia o programador, pois são grandes as opções da plataforma Java e, às vezes, mesmo conhecendo-a de forma avançada, não é possível memorizar todos os padrões. Como o *framework* gera boa parte do código, ele propicia ao programador uma facilidade em analisar o código de forma reversa após a geração.

Nesta pesquisa, o Eclipse atuou como *framework* central, recebendo objetos do *iReport* e do *MySql Workbench*. Também poderia ter se optado por *plugins* diretamente no Eclipse para realização dessas tarefas.

O desenvolvimento foi assistido por *frameworks* da primeira à última etapa. No caso das ferramentas usadas, todas permitem ao usuário, caso queira ou necessite, trabalhar diretamente no código. Para que tudo funcionasse corretamente, em alguns casos seguiu-se padrões dos *frameworks*, mas nada que comprometesse o desenvolvimento.

Com o impacto na produtividade ao usar um *framework,* as empresas podem passar a se perguntar se o ideal é um profissional com conhecimento avançado na linguagem que será usado ou um conhecimento avançado nos *frameworks* disponíveis para trabalhar com ela. O ideal seria a combinação de ambos.

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

A utilização de *frameworks* para desenvolvimento é algo comum. Alguns abrangem várias linguagens ao mesmo tempo, com isso auxiliando na integração de tecnologias diferentes.

A reutilização de componentes e objetos prontos é um paradigma atual, ao invés de recriar o melhor seria utilizar algo que já está pronto. Mesmo quando o desenvolvedor faz seus próprios objetos, ele emprega o conceito de reutilização, então por que não usar o que já foi disponibilizado por outros.

Para a escolha do *framework*, o desenvolvedor deve ficar atento a seus requisitos e saber se os mesmos são condizentes com o que se espera. Se o programador quer ter acesso para alterar todo o código, deve estar atendo se isso é disponibilizado pela ferramenta ou se, por exemplo, é gerado um código com a estrutura padrão da linguagem que pode ser lido por outros *frameworks*. São detalhes que podem ter grande impacto ao se optar por mudar de ferramenta.

Também não se pode criar a falsa idéia que, para todos os problemas, o *framework* terá algo pronto. Não serão poucos os casos onde um objeto disponibilizado deverá ser adaptado as exceções de uma aplicação. Neste ponto vai pesar mais o conhecimento da linguagem que do *framework*.

Existem atualmente ferramentas que se propõe a gerar todo o código fonte sem nenhuma intervenção do profissional na parte técnica da programação, ficando o mesmo apenas com definição correta das regras de negócio (SOFTWELL SOLUTIONS, 2011).

O que deve ser pesado é, o quanto quer se aumentar a produtividade durante o desenvolvimento e qual as conseqüências e custo dessas decisões. Uma escolha incorreta, pode levar o desenvolvedor a criar uma dependência muito forte com a ferramenta usada, tornando complicada a substituição.

5.2 TRABALHOS FUTUROS

Para trabalhos futuros, sugere-se um estudo sobre o *framework* JGoodies, similar ao WindowBuilder Pro. Os pontos a serem levantados sobre esta ferramenta seriam sua usabilidade, a expectativa de vida do mesmo e a existência ou não de suporte técnico.

Também poderia ser feito um comparativo entre o WindowBuilder Pro e o JGoodies ressaltando os pontos positivos e negativos entre ambos.

Por fim, poderia ser demonstrado como usar a ferramenta empregando a mesma em um estudo de caso que abrangesse os recursos disponibilizados por ela.

REFERÊNCIAS

ATTARZADEH, I.; OW, H. S. Software Development Effort Estimation. **International Journal of Computer Theory and Engineering (IJCTE)**, Cingapura/Asia, 01 Out 2009. 1. Disponivel em: <http://www.ijcte.org/papers/077.pdf>. Acesso em: 03 Set 2011.

BARBARÁN, G. C.; FRANCISCHINI, P. G. INDICADORES DE PRODUTIVIDADE NA INDÚSTRIA DE SOFTWARE, 2011. Disponivel em: <http://www.abepro.org.br/biblioteca/ENEGEP1998_ART515.pdf>. Acesso em: 25 Julho 2011.

BERTOLLO, G. et al. **ODE – Um Ambiente de Desenvolvimento de Software**., 2011. Disponivel em: <http://www.inf.ufes.br/~falbo/download/pub/CFSbes2002.pdf>. Acesso em: 09 jul. 2011.

BEUREN, I. M. **Como Elaborar Trabalhos Monográficos em Contabilidade**. 3. ed. São Paulo: Atlas, 2008.

BLEWITT, A. Lançado Eclipse Helios. **infoQ**, 2010. Disponivel em: <http://www.infoq.com/br/news/2010/06/eclipse-helios>. Acesso em: 20 Julho 2011.

CARROMEU, C. Linha de Produtos de Software no Processo de Geração de Sistemas Web de Apoio a Gestão de Fomento de Projetos. **TEDE/UFMS**, 2007. Disponivel em: <http://www.cbc.ufms.br/tedesimplificado/tde_arquivos/1/TDE-2010-02-05T064814Z-560/Publico/Camilo.pdf>. Acesso em: 17 Julho 2011.

CÔRTES, S. D. C. Conceitos e Fundamentos. In: ______ **Um modelo de transações para integração de SGDB a um ambiente de Computação Movel**. Rio de Janeiro: [s.n.], 2004. p. 24-50. Tese (Doutorado em Informática) - Programa de Pós-graduação em Informatica da PUC-Rio. Disponivel em: <http://www2.dbd.puc-rio.br/pergamum/tesesabertas/9924918_04_cap_02.pdf>. Acesso em: 07 set. 2011.

DELAP, S. Understanding JFace data binding in Eclipse, Part 1: The pros and cons of data binding. **IBM**, 2006. Disponivel em: <http://www.ibm.com/developerworks/opensource/library/os-ecl-jfacedb1/>. Acesso em: 13 set. 2011.

HACETTEPE UNIVERSITY. Cost Estimation Models. **Sencer Sultanoglu's Page**, 1998. Disponivel em: <http://yunus.hacettepe.edu.tr/~sencer/cocomo.html#cocomo81>. Acesso em: 03 Set 2011.

INTERNATIONAL BUSINESS MACHINES. IBM Rational Unified Process (RUP). **IBM - United States**, 2011. Disponivel em: <http://www-01.ibm.com/software/awdtools/rup/>. Acesso em: 05 out. 2011.

JUNIOR, A. F. F. Biblioteca FAI. **FAI - Centro de Ensino Superior em Gestão, Tecnologia e Educação**, 2007. Disponivel em: <http://www.fai-mg.br/biblioteca/index.php?option=com_docman&task=doc_download&gid=17&Itemid=99999999>. Acesso em: 17 Julho 2011.

KÜNNETH, T. Binding Beans. **java.net**, 2008. Disponivel em: <http://today.java.net/pub/a/today/2008/05/27/binding-beans.html?force=644>. Acesso em: 14 set. 2011.

MACEDO, M. D. M. Gestão da produtividade nas empresas: A aplicação do conceito de Produtividade Sistêmica permite. **FAE BUSINESS**, Curitiba, set. 2002. Disponivel em: <http://www.fae.edu/publicacoes/pdf/revista_fae_business/n3_setembro_2002/ambiente_economico3_gestao_da_produtividade_nas_empresas.pdf>. Acesso em: 07 set. 2011.

MACEDO, M. V. L. R. Laboratório de Administração e Segurança de Sistemas. **Instituto de Computação da UNICAMP**, Campinas, Dez 2003. Disponivel em: <http://www.las.ic.unicamp.br/paulo/teses/20031211-MP-Marcus.Vinicius.La.Rocca.Macedo-Uma.proposta.de.aplicacao.da.metrica.de.pontos.de.funcao.em.aplicacoes.de.dispositivos.portateis.pdf>. Acesso em: 03 Set 2011.

O'CONNER, J. Synchronizing Properties with Beans Binding (JSR 295). **java.net**, 2008. Disponivel em: <http://today.java.net/pub/a/2008/03/20/synchronizing-properties-with-beans-binding.html?force=203>. Acesso em: 13 set. 2011.

ORACLE AND/OR ITS AFFILIATES. MySQL 5.6 Reference Manual. **MySQL**, 2011. Disponivel em: <http://dev.mysql.com/doc/refman/5.6/en/index.html>. Acesso em: 2011 set. 14.

ORACLE AND/OR ITS AFFILIATES. The InnoDB Storage Engine. **MySql**, 2011. Disponivel em: <http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html>. Acesso em: 15 set. 2011.

REINALDO, F. A. F. Definição e Aplicação de um Framework para Desenvolvimento de Redes Neurais Modulares e Heterogêneas. **Pergamum**, Florianópilis, 2003. Disponivel em: <http://www.tede.ufsc.br/teses/PGCC0469.pdf>. Acesso em: 17 Julho 2011.

SANTANA, O. G. D. Por que Java? **DevMedia**, 2011. Disponivel em: <http://www.devmedia.com.br/post-20384-Por-que-java.html>. Acesso em: 25 Julho 2011.

SOFTWELL SOLUTIONS. Maker. **Softwell**, 2011. Disponivel em: <http://www.softwell.com.br/>. Acesso em: 11 nov. 2011.

TAMAKI, P. A. O.; HIRAMA, K. Departamento de Ciências da Computação. **Universidade Federal de Lavras**, 08 Mar 2007. Disponivel em: <http://www.dcc.ufla.br/infocomp/artigos/v6.1/art09.pdf>. Acesso em: 03 Set 2011.

THE ECLIPSE FOUNDATION. Data Tools Platform User Documentation. **Eclipse**, 2010. Disponivel em: <http://help.eclipse.org/helios/index.jsp>. Acesso em: 20 Julho 2011.

THE ECLIPSE FOUNDATION. About EclipseLink. **Eclipse**, 2011. Disponivel em: <http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Introduction/About_EclipseLink>. Acesso em: 09 set. 2011.

THE ECLIPSE FOUNDATION. About the Eclipse Foundation. **Eclipse**, 2011. Disponivel em: <http://www.eclipse.org/org/>. Acesso em: 20 Julho 2011.

THE ECLIPSE FOUNDATION. EclipseLink Project. **Eclipse**, 2011. Disponivel em: <http://www.eclipse.org/projects/project_summary.php?projectid=rt.eclipselink>. Acesso em: 09 set. 2011.

THE ECLIPSE FOUNDATION. WindowBuilder - is a powerful and easy to use bi-directional Java GUI designer. **Eclipse**, 2011. Disponivel em: <http://www.eclipse.org/windowbuilder/>. Acesso em: 09 set. 2011.

TOFFOLI, G. **JasperSoft Ultimate Guide**., 2011. Disponivel em: <https://www.jaspersoft.com/sites/default/files/sample_ireport_ultimate_guide_3_pdf_14156.pdf>. Acesso em: 15 set. 2011.